Symmetry Breaking Ordering Constraints

by

ZEYNEP KIZILTAN

A dissertation presented at Uppsala University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Uppsala University
Department of Information Science
Computer Science Division

5 March 2004

UPPSALA
UNIVERSITET

Dissertation for the Degree of Doctor of Philosophy in Computer Science presented at Uppsala University in 2004.

ABSTRACT

Kiziltan, Z. 2004. Symmetry Breaking Ordering Constraints. 250 pp. Uppsala. ISBN 91-506-1737-0

Many problems in business, industry, and academia can be modelled as constraint programs consisting of matrices of decision variables. Such "matrix models" often have symmetry. In particular, they often have row and column symmetry as the rows and columns can freely be permuted without affecting the satisfiability of assignments. Existing methods have difficulty in dealing with the super-exponential number of symmetries in a problem with row and column symmetry. We therefore propose some ordering constraints which can effectively break such symmetries. To use these constraints in practice, we develop some efficient linear time propagators. We demonstrate the effectiveness of these symmetry breaking ordering constraints on a wide range of problems. We also show how such ordering constraints can be used to deal with partial symmetries, as well as value symmetries.

Keywords : Constraint satisfaction, constraint programming, modelling, symmetry breaking, global constraints

Zeynep Kiziltan, Department of Information Science, Computer Science, Kyrkogårdsg. 10, Box 513, Uppsala University, SE-751 20 Uppsala, Sweden

ISBN 91-506-1737-0
urn:nbn:se:uu:diva-3991 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-3991)

Printed in Sweden by University Printers, Uppsala 2004.
Distributor: Department of Information Science, Uppsala University.

Sevgili anneannem Gülsüm Aydan (1910–2003)
ve teyzem Gülşen Aydın'ın (1936–2004) anısına

Acknowledgements

I started my doctoral studies at Uppsala University in September 1999. I should like to thank Pierre Flener for encouraging me to pursue a Ph.D. degree, for taking the initiative in me coming to Uppsala, and for his financial support. I am forever grateful to Andreas Hamfelt who gave me the opportunity of working in Uppsala, and who has been a very concerned supervisor and a bracing chair, commanding anything I need to carry out my research at high standards.

This dissertation would not have been completed without the guidance and help of some leading researchers in constraint programming. First and foremost, my sincere thanks go to my external supervisor Toby Walsh. I have learnt a lot from his deep knowledge in computer science and mathematics, his experience in doing research, and his talent in simplifying anything that looks complicated. His never ending energy has been a source of inspiration and encouragement, and his confidence in my abilities has been a stimulating challenge for me. It never mattered whether Toby was sitting next door or in Australia or changing planes in between his trips. He was always ready for an advice, help, and intellectual and emotional support. Thank you Toby for all these. I know that our fruitful collaboration is not over: it just started!

The person who has not only been a fellow Ph.D. student but also an informal supervisor and an intimate friend is Brahim Hnich. I wonder how things would turn up if I never met Brahim. The years we spent learning constraint programming together, the time we spent humorously in Uppsala and in Cork, and his words of "Kız, don't worry so much" are a valuable gift to my life. My special acknowledgements are to my co-authors Alan Frisch and Ian Miguel. It has been a great pleasure to work with you. Our "fights" in meeting rooms have resulted in good research results and papers.

I was fortunate to have the opportunity of visiting several research labs during the time of my studies. My first research visit was in the summer of 2000 to the University of Essex in England. My future research position began to shape up when Edward Tsang asked me "Zeynep, what is your research agenda?". I am very thankful to him for asking me this question. My next visit was in the fall of 2001 to the University of York in England, which allowed me to start working closely with Toby, Alan, and Ian. It is because of my several visits in 2002 to the University of Bologna in Italy that I obtained a new perspective in constraint programming and in doing research in general from Michela Milano and Andrea Lodi. Joining to their meetings was both fun and an amazing experience. Toby and Eugene Freuder were very kind to invite me to Ireland to visit Cork Constraint Computation Centre in the second half of 2002. The six-month period in Cork had been very productive, during which I worked closely with Toby and Brahim, as well as maintained the collaboration with Ian, Chris, and Alan in York. I would like to thank you all for your contribution to my research, for your professional advice and your hospitality.

Being a member of the APES multi-site research group, I have kept close contacts with its members and their collaborators, to whom I send my appreciation and regards.

I benefitted a lot from Chris Beck, Ian Miguel, and Chris Jefferson when learning how to program in Solver. I remember how they took every frustrating bug or problem very seriously and spent days debugging together with me. May your constraint programs be bug free ;-)

The days spent in Uppsala and abroad during the past 4 years have always been joyful because I have worked in a sociable atmosphere and made close friends all over the world. Gunilla Klaar, Eva Enefjord, Ingrid Lukell, Lars-Göran Svensk, Davoud Emamjomeh, and Pierre Hjälm have been very helpful to me with all sorts of administrative and technical issues. It is nearly impossible to list all the people at the department here who were always ready for help. However, the following people deserve a special mention. Thank you Torsten Palm for acting as my "swedish father" and being a devoted mate at gympa classes. I have warm thoughts of Sweden because of Anne Håkansson, Anneli Edman, Isabella Scandurra, Maria Hägglund, Vladislav Valkovsky, Thomas Snäckerström, Therese Monstad, and Jenny Eriksson Lundström. *Ett stort tack, jag kommer att sakna er allihop!*

My heart-felt wishes are to Gordon Kelly, Milena Koparanova, Timour Katchaounov and Adela Katchaounova, Maria Muskos, Monica Tavanti, Gustaf, Mehtap and İsmail Kurtoğlu, Halit Koşmaz, Hediye Güzel, the Moradi family, Demet Tezcan, and Seval Çelikel together with whom I have spent all those unforgettable times in Sweden. Thank you Elçim Olgaç, Richard Bradwell, Filiz Çaycı, Simon Colton, Sandra Ashworth, Angela Miguel, Julian Richardson, Therry and Jim Kelly for hosting me on my several trips to England and Scotland; thank you Steve Prestwich, Barbara Doyle, James Little, Chris Beck, Angela Glover, Martina Connolly, Barry and Linda O'Sullivan, Cormac Gebruers and Suzanne, Tom Carchare for your warm friendship, and Peter MacHale and Christina Offutt for your endless technical support in Ireland. Words are not enough to show my appreciation to Karin Peer for taking me around in Vienna, to Sophie for her hospitality in Paris, to Angelica Reyes and Peppe for being a great guide in Barcelona and Madrid, and to Alessandro Arcangeli, the Torroni family, Luigi Marvasi, Ombretta Conti, Francesco Panciroli, Simonetta, Federica, and İstem and Emin for making me feel at home in Italy.

I have often felt homesick because I have been far away from my family and friends (too many to name) in Turkey. On the other hand, I have always felt their love and support with a precious intimacy. *Bugünlere gelmemde bana en büyük desteği sağlayan annem Hatice Kızıltan, babam Süleyman Kızıltan, abim Kaan Kızıltan, yengem Sevim Kızıltan'a, ve tüm akraba ve arkadaşlarıma candan ve sonsuz teşekkürler!*

No Paolo, I have not forgotten you. I just do not know how to thank you for following me wherever I am, for all your support, patience and love, especially during the stressful months of writing this dissertation. *Insieme a te tutto diventa bellissimo!*

This dissertation is dedicated to the memories of my grand mother Gülsüm Aydan (1910–2003) and my aunt Gülşen Aydın (1936–2004).

Zeynep Kızıltan
January 22, 2004

Declarations

I hereby declare that I composed this dissertation entirely myself. Parts of the dissertation have appeared in the following publications which have been subject to peer review.

1. B. Hnich, Z. Kiziltan, and T. Walsh. Combining symmetry breaking with other constraints: lexicographic ordering with sums. In *Proceedings of the 8th International Symposium on Artificial Intelligence and Mathematics (AMAI-04)*, 2004. A longer version appears in Notes of the 3rd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-03), CP-03 Post-conference Workshop.

2. A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Multiset ordering constraints. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 221–226. Morgan Kaufmann, 2003. A longer version is invited to the journal *Artificial Intelligence*.

3. A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2002.

4. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetry in matrix models. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2002.

5. Z. Kiziltan and B. M. Smith. Symmetry breaking constraints for matrix models. In Notes of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02), CP-02 Post-conference Workshop, 2002.

6. Z. Kiziltan and T. Walsh. Constraint programming with multisets. In Notes of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02), CP-02 Post-conference Workshop, 2002.

7. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in matrix models. Technical Report APES-30-2001, APES Research Group, 2001. Also in Notes of the 1sth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-01), CP-01 Post-conference Workshop.

8. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling. Technical Report APES-36-2001, APES Research Group, 2001. Also in Notes of the CP-01 Post-conference Workshop on Modelling and Problem Formulation (Formul-01).

However, the contents of the work in these papers are presented in more detail and mistakes (if any) are corrected in this dissertation. Moreover, this dissertation contains a lot of new material which have not yet been published anywhere but which will soon appear in some journals.

My contribution to each technical chapter is as follows.

Chapter 3. This chapter is a result of collaboration with Pierre Flener, Alan Frisch, Brahim Hnich, Ian Miguel, and Toby Walsh. I was involved in the study of all the matrix models presented and I formalised the definitions of symmetry.

Chapter 4. This chapter is a result of collaboration with Pierre Flener, Alan Frisch, Brahim Hnich, Ian Miguel, Justin Pearson, and Toby Walsh. Whilst it is difficult to identify exactly the individual contributions, I declare that the study of the ordering constraints in Sections 4.2 and 4.3 are mainly my own work. The theoretical results given without proofs in Sections 4.4 and 4.5 are not my own work.

Chapter 5. All the theoretical and experimental work are done by me. The algorithm arose as a result of discussions with Alan Frisch, Brahim Hnich, Ian Miguel, and Toby Walsh. Ian Miguel implemented the incremental algorithm. I studied different implementations of the algorithm together with Ian Miguel.

Chapter 6. All the theoretical and experimental work are done by me. The algorithm as well as the conclusions of Section 6.9 arose as a result of discussions with Brahim Hnich and Toby Walsh.

Chapter 7. All the theoretical and experimental work are done by me. The two algorithms arose as a result of discussions with Alan Frisch, Brahim Hnich, Ian Miguel, and Toby Walsh. However, I reconstructed the algorithms.

Chapter 8. This chapter is my own work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis defended in this dissertation is that:

> *Row and column symmetry is a common type of symmetry in constraint programming. Ordering constraints can effectively break this symmetry. Efficient global constraints can be designed for propagating such ordering constraints.*

This chapter introduces the research presented in this dissertation and is organised as follows. In Section 1.1, we briefly introduce constraint programming. In Section 1.2, we discuss the motivations, and then in Section 1.3 we define the goals of our work. We give an overview of the dissertation in Section 1.4 and summarise the major contributions in Section 1.5. Finally, we present the organisation of the dissertation in Section 1.6.

1.1 Constraint Programming

Giving decisions in the presence of some constraints is an important part of our life. For instance, we want to organise a meeting in one of a choice of places within the following three months, and thus we need to decide when and where to hold this meeting. There are some constraints which prevent us from giving a quick decision. First, each participant of the meeting has a wish list specifying which days and places are suitable for her. Second, each place has restricted availability, meaning that it may not be possible to hold the meeting at a place on any day we want. Third, we cannot hold a meeting on a holiday. Now, is it possible to find a place and day in a such a way that the day is not a holiday, the place is available on the day, and every participant is happy about the choice?

Many decision problems, like deciding a meeting place and time, can be formulated as constraint satisfaction problems (CSPs). A CSP has a set of decision variables and a set of constraints. Each variable is associated with a finite set of values, called the domain of the variable, giving the possible values that the variable can take. Each constraint is defined on a subset of the variables restricting the possible values that the variables can simultaneously take. An example is the map colouring problem. We have a map of regions and a set of colours. We would like to colour the regions in such a way that no neighbouring regions have the same colour. Assume that the regions we have are some countries in western Europe: Belgium, Denmark, France, Germany, Netherlands, and Luxembourg, and assume that the set of available colours is $\{red, blue, white, green\}$. We can formulate this problem as a CSP using a set of variables $\{B, D, F, G, N, L\}$, each corresponding to a country in the map:

$$\begin{array}{rcl} \text{Belgium} & \dashrightarrow & B \\ \text{Denmark} & \dashrightarrow & D \\ \text{France} & \dashrightarrow & F \\ \text{Germany} & \dashrightarrow & G \\ \text{Netherlands} & \dashrightarrow & N \\ \text{Luxembourg} & \dashrightarrow & L \end{array}$$

The domain of each variable is the set of colours $\{red, blue, white, green\}$. For each pair of neighbouring countries, there is a constraint on the corresponding variables which disallows them to take the same value:

$$\begin{array}{ccc} F \neq B & F \neq L & F \neq G \\ L \neq G & L \neq B & B \neq N \\ B \neq G & G \neq N & G \neq D \end{array}$$

A solution to a CSP is an assignment of values to the variables such that all constraints are satisfied simultaneously. For instance, the following is a solution to our map colouring problem:

$$\begin{array}{ll} B \leftarrow blue & D \leftarrow blue \\ F \leftarrow white & G \leftarrow red \\ N \leftarrow white & L \leftarrow green \end{array}$$

CSPs are ubiquitous in various fields as diverse as artificial intelligence (e.g. temporal reasoning), control theory (e.g. design of controllers for sensory based robots), concurrency (e.g. process communication and synchronisation), computer graphics (e.g. geometric coherence), database systems (e.g. constraint databases), operations research (e.g. optimisation problems) [vHS97], bioinformatics (e.g. sequence alignment) [GBY01], and business applications (e.g. combinatorial auctions) [GM03]. In general, solving CSPs is NP-hard and so is computationally intractable [Mac77]. That is, no polynomial algorithm for solving CSPs is known to exist.

Constraint programming (CP) provides a platform for solving CSPs [MS98][Apt03] and has proven successful in many real-life applications [Wal96][Ros00] despite this intractability. The techniques used to solve CSPs originate from various fields of computer science, such as artificial intelligence, operations research, computational logic, combinatorial algorithms, discrete mathematics, and programming languages. The core of CP is a combination of search and inference. Solutions for CSPs are searched in the space of possible assignments using a search algorithm. Inference methods maintain various forms of local consistency on the constraints and reduce the domains of the variables. The inference can propagate across the constraints and can greatly reduce the effort involved in searching for solutions. This helps to make CP a powerful technology to solve CSPs, in some cases, more efficient than integer linear programming (ILP) methods (see for instance [SBHW96][DL98]).

To solve a problem using CP methods, we need first to formulate it as a CSP by declaring the variables, their domains, and the constraints on the variables. This part of the problem solving is called modelling. There are often many alternatives for each modelling decision such as the choice of the decision variables and how we state the constraints. For instance, we can just as well model the map colouring problem with a variable for each colour. Each variable then takes as a value a set of countries. To disallow neighbouring countries having the same colour, we can enforce that no pair of countries in the set assigned to a variable are neighbours. Different models have different properties. Often, alternative models need to be tried because a small modification to a model can have a huge impact on the efficiency of the solution method [Bor98][BT01][Hni03]. Hence, formulating an effective model for a given problem requires considerable skills in modelling and remains a challenging task even for modelling experts [Fre98].

1.2 Motivations

There are many recurring patterns in constraint programs [Wal03]. Identification of these patterns has two important benefits. First, patterns can be used to pass on modelling expertise. Second, special-purpose methods to support the patterns can be devised for use by a (non-expert) modeller. This helps tackle the difficulty of effective modelling and makes CP reachable to a wider audience.

One common pattern in constraint programs is a matrix model. Any formulation of a problem as a CSP which employs one or more matrices of decision variables is a matrix model. As an example, consider the sport scheduling problem which is about scheduling games between n teams over $n-1$ weeks (prob026 in CSPLib [CSP]). Each week is divided into $n/2$ periods, and each period is divided into two slots. The team in the first slot plays at home, while the team in the second slot plays away. The goal is to find a schedule such that every team plays exactly once a week, every team plays against every other team, and every team plays at most twice in the same period over the tournament.

As we need a table of meetings, a natural way to model this problem is to use a 2-dimensional matrix of variables, $S_{w,p}$, each of which is assigned a value corresponding to the match played in a given week w and period p [vHMPR99]. To visualise, we have the following matrix of variables for $n = 8$:

$S_{w,p}$	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	$S_{1,1}$	$S_{2,1}$	$S_{3,1}$	$S_{4,1}$	$S_{5,1}$	$S_{6,1}$	$S_{7,1}$
Period 2	$S_{1,2}$	$S_{2,2}$	$S_{3,2}$	$S_{4,2}$	$S_{5,2}$	$S_{6,2}$	$S_{7,2}$
Period 3	$S_{1,3}$	$S_{2,3}$	$S_{3,3}$	$S_{4,3}$	$S_{5,3}$	$S_{6,3}$	$S_{7,3}$
Period 4	$S_{1,4}$	$S_{2,4}$	$S_{3,4}$	$S_{4,4}$	$S_{5,4}$	$S_{6,4}$	$S_{7,4}$

Matrix models have been long advocated in 0/1 ILP [Sch86] and are commonly used in CP. Of the 38 problems of CSPLib [CSP] on September 22, 2003, at least 33 of the 38 have matrix models, most of them already published and proved successful [FFH$^+$01b].

An important aspect of modelling is symmetry. A symmetry is a transformation of an entity which preserves the properties of the entity. For instance, rotating a chess board 90° gives us a board which is indistinguishable from the original one. A problem exhibits symmetry when there are identical objects that cannot be distinguished. For instance, the weeks, periods, and teams in the sport scheduling problem are essentially indistinguishable and symmetric. When there is symmetry in a problem, it can be exploited in a powerful way to reuse what we know about one object for its symmetrically equivalent objects. This was first discussed in [BFP88] in the context of the search algorithms for problems with symmetry, and has been considered in diverse areas including planning [FL99][FL02], SAT [CGLR96][ARMS02], ILP [Mar02][Mar03], and model checking [ES93][ID93].

Regarding CSPs, it is possible that a CSP has symmetries in the variables or domains or both which preserve satisfiability. In the presence of symmetry, any assignment of values to the variables can be transformed into a set of symmetrically equivalent assignments without affecting whether or not the original assignment satisfies the constraints. As an example, consider the map colouring problem. As long as the neighbouring countries are coloured with a different colour, it does not matter how the countries are coloured. Hence, the colours are indistinguishable, and the initial formulation of the problem has symmetry in the domains. What does this mean? By changing the roles of the colours, we can obtain symmetrically equivalent assignments to a given assignment. For instance, given the solution:

$$\begin{array}{ll} B \leftarrow \mathit{blue} & D \leftarrow \mathit{blue} \\ F \leftarrow \mathit{white} & G \leftarrow \mathit{red} \\ N \leftarrow \mathit{white} & L \leftarrow \mathit{green} \end{array}$$

we can rename the colours as:

$$\begin{array}{rcl} \mathit{red} & \dashrightarrow & \mathit{green} \\ \mathit{blue} & \dashrightarrow & \mathit{white} \\ \mathit{white} & \dashrightarrow & \mathit{red} \\ \mathit{green} & \dashrightarrow & \mathit{blue} \end{array}$$

and obtain another assignment which is also a solution:

$$\begin{array}{ll} B \leftarrow \mathit{white} & D \leftarrow \mathit{white} \\ F \leftarrow \mathit{red} & G \leftarrow \mathit{green} \\ N \leftarrow \mathit{red} & L \leftarrow \mathit{blue} \end{array}$$

These two solutions are not fundamentally different. In each of them, B and D are assigned the same colour, F and N are assigned the same colour, and these two colours together with those assigned to G and L are all different.

As solutions to CSPs are found by searching through the space of possible assignments, symmetry in a CSP creates symmetric but essentially equivalent states in its search space. Visiting symmetric states can significantly slow down the search process. If a state leads to a solution then all its symmetric states also do, and these symmetric solutions are indistinguishable from each other. This could be a big problem when, for instance, proving optimality because it is worthless to traverse repeatedly the states which do not lead to any improvement in the objective function. Even if we are interested in one solution, we may explore many failed and symmetrically equivalent states before finding a solution. Hence, pruning the symmetric parts of the search space, which is often referred to as symmetry breaking, is essential to speed up the search efficiency. This has attracted many researchers in recent years, and several symmetry breaking methods such as SES [BW99][BW02], SBDS [GS00], and SBDD [FM01][FSS01] have been devised (see Chapter 2 for a detailed discussion of methods for breaking symmetry in CSPs).

A common pattern in matrix models is row and column symmetry [FFH^{+}01b]. A 2-dimensional matrix has row and column symmetry if its rows and columns represent indistinguishable objects and are therefore symmetric. We can permute any two rows as well as two columns of a matrix with row and column symmetry without affecting the satisfiability of any assignments. For example, consider the matrix model of the sport scheduling problem. The columns represent the weeks. As the order of the weeks is not important in a schedule, we can freely swap the assignments of two different weeks. We can visualise this on the following assignment which is a solution to the problem when $n = 8$:

$S_{w,p}$	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	0 vs 1	0 vs 2	4 vs 7	3 vs 6	3 vs 7	1 vs 5	2 vs 4
Period 2	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
Period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
Period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

By swapping the assignments of, say weeks 2 and 5, we get:

$S_{w,p}$	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	0 vs 1	3 vs 7	4 vs 7	3 vs 6	0 vs 2	1 vs 5	2 vs 4
Period 2	2 vs 3	1 vs 4	0 vs 3	5 vs 7	1 vs 7	0 vs 6	5 vs 6
Period 3	4 vs 5	2 vs 6	1 vs 6	0 vs 4	3 vs 5	2 vs 7	0 vs 7
Period 4	6 vs 7	0 vs 5	2 vs 5	1 vs 2	4 vs 6	3 vs 4	1 vs 3

This new assignment is another solution as the problem constraints are still satisfied. Let us now consider the rows which represent the periods. As the periods are indistinguishable, we can freely interchange the assignments of two different periods. By swapping the assignments of, say periods 1 and 2, in the initial solution, we get:

$S_{w,p}$	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
Period 2	0 vs 1	0 vs 2	4 vs 7	3 vs 6	3 vs 7	1 vs 5	2 vs 4
Period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
Period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

which is another solution. We have obtained two new solutions from the initial solution. However, there is no fundamental difference between these three solutions. We can obtain further symmetric solutions by permuting any two rows, any two columns, as well as by combining a row and a column permutation. This gives us in total 7!4! symmetrically equivalent solutions.

An $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries, which increase super-exponentially. Thus, it can be very costly to visit all the symmetric branches in a tree search. Ideally, we would like to cut off all the symmetric parts of the search tree. Symmetry breaking methods such as SES [BW99][BW02], SBDS [GS00], and SBDD [FM01][FSS01] achieve this goal by not exploring the parts of the search tree which are symmetric to the those that are already considered. Even though these methods are applicable to any class of symmetries, they may not be a good way of dealing with row and column symmetry for the following reasons. SES and SBDS treat each symmetry individually, which is impractical when the number of symmetries is large. Similarly, the dominance checks of SBDD can be very expensive in the presence of many symmetries. We therefore need special techniques to deal with row and column symmetry effectively.

Another common pattern in matrix models is value symmetry. A matrix has value symmetry if the values in the domain of the variables are indistinguishable and are therefore symmetric. We can then permute the values that the variables take without affecting any assignments. For example, consider the assignment which is a solution to the sport scheduling problem when $n = 8$:

$S_{w,p}$	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	0 vs 1	0 vs 2	4 vs 7	3 vs 6	3 vs 7	1 vs 5	2 vs 4
Period 2	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
Period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
Period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

As the teams are indistinguishable, we can permute the teams as:

$$\begin{array}{cccccc} 0 & \dashrightarrow & 1 & 4 & \dashrightarrow & 5 \\ 1 & \dashrightarrow & 2 & 5 & \dashrightarrow & 6 \\ 2 & \dashrightarrow & 3 & 6 & \dashrightarrow & 7 \\ 3 & \dashrightarrow & 4 & 7 & \dashrightarrow & 0 \end{array}$$

and get another assignment which is also a solution:

$S_{w,p}$	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	1 vs 2	1 vs 3	5 vs 0	4 vs 7	4 vs 0	2 vs 6	3 vs 5
Period 2	3 vs 4	2 vs 0	1 vs 4	6 vs 0	2 vs 5	1 vs 7	6 vs 7
Period 3	5 vs 6	4 vs 6	2 vs 7	1 vs 5	3 vs 7	3 vs 0	1 vs 8
Period 4	7 vs 0	5 vs 7	3 vs 6	2 vs 3	1 vs 6	4 vs 5	2 vs 4

We have again obtained a new solution from a given solution, and again there is no fundamental difference between these two solutions. By permuting the values in 8! different ways, we can get in total 8! symmetrically equivalent solutions.

A variable V of an n dimensional matrix that takes a value from a domain of indistinguishable values $\{v_1, \ldots, v_m\}$ can be replaced by a vector $\langle V_1, \ldots, V_m \rangle$ of 0/1 variables, with the semantics $V_i = 1 \leftrightarrow V = v_i$. Now, instead of the values, the new variables are indistinguishable. This converts value symmetry into row or column symmetry. Therefore, any effective technique for dealing with row and column symmetry can also deal with value symmetry in a matrix.

1.3 Goals

Row and column symmetry is an important class of symmetries, as it is very common and symmetry breaking methods like SES, SBDS, and SBDD have difficulty in dealing with the super-exponential number of symmetries in a problem with row and column symmetry. In many cases, large problems are intractable unless such symmetries are significantly reduced. Value symmetry in a matrix can easily be transformed to, for instance, row symmetry. Our main objective in this dissertation is therefore to develop special techniques to deal with row and column symmetry effectively.

One of the easiest and most efficient ways of symmetry breaking is adding extra constraints to the model of our problem [Pug93]. These constraints impose an ordering on the symmetric objects. For instance, assume that we have a set of symmetric variables $\{X, Y, Z\}$ each associated with the domain $\{1, 2, 3\}$. To break the symmetry, we can post the constraints:

$$X \leq Y \ \wedge \ Y \leq Z$$

As a consequence, among the set of symmetric assignments, only those that satisfy the ordering constraints are chosen for consideration during search.

To achieve our objective, the approach adopted in this research is to attempt to impose ordering constraints on the rows and columns of a matrix. This approach to breaking row and column symmetry has at least two important benefits over using methods like SES, SBDS, and SBDD. First, unlike SES and SBDS, we do not deal with each symmetry individually. An $n \times m$ matrix has n columns and m rows. As ordering relations are transitive, we can impose the ordering constraints between the adjacent rows and columns. Even if the symmetries grow super-exponentially, we only need $O(n + m)$ constraints to deal with the symmetries. Second, in the presence of many symmetries, a subset of the symmetries can be used to break some but not necessarily all symmetries using SES and SBDS. This requires writing problem-specific search methods. Similarly, we need problem-specific dominance checks when symmetry breaking using SBDD. Ordering constraint, however, can be used to break the row and column symmetries of any problem modelled using a matrix. Indeed, they can be used for matrices of arbitrary dimension.

Our approach to breaking row and column symmetry raises very important research questions that need to be addressed. First, what are those ordering constraints that we can impose? Care needs to be taken when posting ordering constraints. As the ordering constraints remove some assignments from a set of symmetrically equivalent assignments, it is important that at least one assignment remains in the set. Otherwise, we may lose solutions by symmetry breaking. Second, how can we use these constraints in practice? Global constraints play a central role in CP, as they facilitate problem formulation, and encapsulate powerful propagation mechanisms to detect and remove from the domains those values that cannot be a part of any solution (see Chapter 2 for a detailed discussion on global constraints). Can we devise global constraints to post and propagate the ordering constraints effectively and efficiently? Third, are the ordering constraints effective in breaking row and column symmetries? Even though our approach has important benefits over using methods like SES, SBDS, and SBDD, there are some drawbacks. First, posting ordering constraints on the rows and columns may not eliminate all symmetries. Second, the ordering imposed by these constraints may conflict with the way we conduct search, resulting in larger search space and longer run-times. For instance, consider the symmetric variables X, Y, and Z. By posting the ordering constraints $X \leq Y \ \wedge \ Y \leq Z$, we do not admit an assignment like:

$$X \leftarrow 3 \quad Y \leftarrow 1 \quad Z \leftarrow 2$$

as a solution, though it is a solution in the absence of the ordering constraints. During search, if this assignment is found then it will be rejected as a solution, and alternative parts of the space will be searched to find an assignment satisfying both the problem constraints and the ordering constraints. Hence, we might decrease the search efficiency while trying to increase it by symmetry breaking with ordering constraints.

Consequently, the goals of the research presented in this dissertation are:

1. to investigate the types of ordering constraints that can be posted on a matrix to break row and column symmetries;

2. to devise global constraints to post and propagate the ordering constraints effectively and efficiently;

3. to show the effectiveness of the ordering constraints in breaking row and column symmetries.

Our motivations and goals have been used to form a central thesis which will be defended in this dissertation:

> *Row and column symmetry is a common type of symmetry in constraint programming. Ordering constraints can effectively break this symmetry. Efficient global constraints can be designed for propagating such ordering constraints.*

It is not the goal of this research to study how many symmetries are eliminated by posting ordering constraints. While this is a worthy area of study, we are concerned only with how effective the ordering constraints are in breaking row and column symmetries. We judge this by looking at how we reduce the size of the search space and the time to solve the problem. We believe that noteworthy reductions in the search space and time are significant steps towards the tractability of CSPs.

We do not compare our approach directly with the symmetry breaking methods SES, SBDS, and SBDD for the following reasons. First, such a comparison is outside the scope of the thesis we wish to defend. Second, not all the methods are incorporated into the publicly available CP toolkits. Third, we tackle larger problems than those reported by using SBDS and SBDD. Fourth, using SBDD and a subset of the symmetries in SES and SBDS require writing problem-specific methods. Fifth, some of the recent experimental results [Pug02a][Pug03a] depend on the symmetry breaking ordering constraints presented in this dissertation.

Another way of symmetry breaking is the heuristic method adopted by Meseguer and Torras [MT99][MT01], which has been successfully applied to solving large instances of a problem modelled using a matrix with row and column symmetry. Comparing our approach and this heuristic approach on a wide range of problems with row and column symmetry is a research area of its own, and thus we discuss this only in Chapter 9 as future work.

1.4 Overview of the Dissertation

We start our research by exploring a wide range of problems originating from combinatorics, design, configuration, scheduling, timetabling, bioinformatics, and code generation. We observe that matrix modelling provides an effective way of representing these diverse problems for the following reasons. First, the problem constraints can then be expressed in terms of the well-known global constraints, such as *all-different*, global cardinality (*gcc*), sum, and scalar product. There are powerful propagation algorithms for these constraints and they are available in many constraint toolkits. Second, messy side constraints can be effectively represented by using multiple matrices and chanelling between the different matrices. This can, for instance, eliminate the need of posting large-arity constraints that can only be efficiently implemented by means of a complex daemon. Many of the studied matrix models were previously proposed and shown to be effective.

Having recognised the central role played in many constraint programs by matrix models, we identify two patterns that commonly arise in matrix models: row and column symmetry, and value symmetry. A 2-dimensional matrix has row (resp. column) symmetry iff its rows (resp. columns) represent indistinguishable objects and are therefore symmetric. If, however, only a strict subset(s) of the rows (resp. columns) are indistinguishable, then the matrix has partial row (resp. column) symmetry. In each of the 12 matrix models we have studied, there is at least one matrix with (partial) row and/or

$$\longrightarrow \text{column permutations} \longrightarrow$$

$$\begin{array}{rcccccc}
 & \overset{*}{\begin{pmatrix}0&1&0\\0&2&3\\1&0&1\end{pmatrix}} & \overset{\star}{\begin{pmatrix}0&0&1\\0&3&2\\1&1&0\end{pmatrix}} & \begin{pmatrix}1&0&0\\2&0&3\\0&1&1\end{pmatrix} & \begin{pmatrix}1&0&0\\2&3&0\\0&1&1\end{pmatrix} & \begin{pmatrix}0&0&1\\3&0&2\\1&1&0\end{pmatrix} & \begin{pmatrix}0&1&0\\3&2&0\\1&0&1\end{pmatrix} \\
 & \overset{\dagger}{\begin{pmatrix}0&1&0\\1&0&1\\0&2&3\end{pmatrix}} & \overset{\bullet}{\begin{pmatrix}0&0&1\\1&1&0\\0&3&2\end{pmatrix}} & \begin{pmatrix}1&0&0\\0&1&1\\2&0&3\end{pmatrix} & \begin{pmatrix}1&0&0\\0&1&1\\2&3&0\end{pmatrix} & \begin{pmatrix}0&0&1\\1&1&0\\3&0&2\end{pmatrix} & \begin{pmatrix}0&1&0\\1&0&1\\3&2&0\end{pmatrix} \\
\begin{matrix}\downarrow\\ \text{row}\end{matrix} & \begin{pmatrix}0&2&3\\0&1&0\\1&0&1\end{pmatrix} & \begin{pmatrix}0&3&2\\0&0&1\\1&1&0\end{pmatrix} & \begin{pmatrix}2&0&3\\1&0&0\\0&1&1\end{pmatrix} & \begin{pmatrix}2&3&0\\1&0&0\\0&1&1\end{pmatrix} & \begin{pmatrix}3&0&2\\0&0&1\\1&1&0\end{pmatrix} & \begin{pmatrix}3&2&0\\0&1&0\\1&0&1\end{pmatrix} \\
\begin{matrix}\text{permutations}\\ \downarrow\end{matrix} & \begin{pmatrix}0&2&3\\1&0&1\\0&1&0\end{pmatrix} & \begin{pmatrix}0&3&2\\1&1&0\\0&0&1\end{pmatrix} & \begin{pmatrix}2&0&3\\0&1&1\\1&0&0\end{pmatrix} & \begin{pmatrix}2&3&0\\0&1&1\\1&0&0\end{pmatrix} & \begin{pmatrix}3&0&2\\1&1&0\\0&0&1\end{pmatrix} & \begin{pmatrix}3&2&0\\1&0&1\\0&1&0\end{pmatrix} \\
 & \begin{pmatrix}1&0&1\\0&1&0\\0&2&3\end{pmatrix} & \begin{pmatrix}1&1&0\\0&0&1\\0&3&2\end{pmatrix} & \overset{\star}{\begin{pmatrix}0&1&1\\1&0&0\\2&0&3\end{pmatrix}} & \begin{pmatrix}0&1&1\\1&0&0\\2&3&0\end{pmatrix} & \begin{pmatrix}1&1&0\\0&0&1\\3&0&2\end{pmatrix} & \begin{pmatrix}1&0&1\\0&1&0\\3&2&0\end{pmatrix} \\
 & \begin{pmatrix}1&0&1\\0&2&3\\0&1&0\end{pmatrix} & \begin{pmatrix}1&1&0\\0&3&2\\0&0&1\end{pmatrix} & \begin{pmatrix}1&0&1\\3&2&0\\0&1&0\end{pmatrix} & \begin{pmatrix}1&1&0\\3&0&2\\0&0&1\end{pmatrix} & \begin{pmatrix}0&1&1\\2&0&3\\1&0&0\end{pmatrix} & \begin{pmatrix}0&1&1\\2&3&0\\1&0&0\end{pmatrix}
\end{array}$$

Figure 1.1: A set of symmetric assignments.

(partial) column symmetry. A matrix has value symmetry if all the values in the domain of the variables are indistinguishable. If, however, only a strict subset(s) of the values are indistinguishable then the matrix has partial value symmetry. Of the 12 matrix models we have studied, 4 of them have (partial) value symmetry. Unlike row and column symmetry, value symmetry is not confined to matrix models. For instance, the original formulation of the map colouring problem has value symmetry. Moreover, value symmetry does not occur as often as row and column symmetry. On the other hand, value symmetry in a matrix can be transformed to, for instance, row symmetry. This is another advantage of developing effective techniques for dealing with row and column symmetries. Such techniques can deal with problems that naturally have row and column symmetry, as well as with problems that have value symmetry.

An $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries. This means that, given an assignment, we can permute its rows and columns in $n!m!$ different ways, and obtain a set of symmetric assignments. We show an example of this in Figure 1.1 on an assignment $\begin{pmatrix}0&1&0\\0&2&3\\1&0&1\end{pmatrix}$ to a 3×3 matrix of variables. By permuting its rows and columns, we obtain $3!3! - 1 = 35$ more assignments which are symmetric to the original assignment. These 36 assignments form an equivalence class. We note that the size of an equivalence class is not necessarily $n!m!$ because different permutations may give identical assignments due to the repeating values in the assignment.

The assignments in an equivalence class are indistinguishable from each other in terms of satisfiability. One of them is a solution iff the remaining assignments are all solutions. In search for solutions, we want to discard the symmetric assignments. So, how can we distinguish between these indistinguishable assignments? As the rows and columns of a matrix are vectors, we can characterise the assignments according to how their row and column vectors are ordered. One ordering of vectors is lexicographic ordering, which is also used to order the words in a dictionary. A vector $\vec{x} = \langle x_0, \ldots, x_{n-1} \rangle$ is lexicographically less than a vector $\vec{y} = \langle y_0, \ldots, y_{n-1} \rangle$ iff there is an index k above which the subvectors are equal, and $x_k < y_k$. For instance, $\langle 0, 2, 3 \rangle$ is less than $\langle 0, 3, 1 \rangle$ in the lexicographic order. We say that the rows (resp. columns) of a matrix of values are lexicographically ordered if each row (resp. column) is no greater than the rows (resp. columns) below (resp. to the right of) it.

Lexicographic ordering is very focused on positions and it ignores values beneath the position where the vectors differ. Multiset ordering, on the other hand, ignores positions but focuses on values. A multiset is a set in which repetition is allowed. For instance, $\{\!\{1, 2, 2, 2, 3, 3\}\!\}$ is a multiset because 2 and 3 occurs more than once. A multiset $\{\!\{\mathbf{x}\}\!\}$ is less than a multiset $\{\!\{\mathbf{y}\}\!\}$ iff $\{\!\{\mathbf{x}\}\!\}$ is empty and $\{\!\{\mathbf{y}\}\!\}$ is not, or the largest value in $\{\!\{\mathbf{x}\}\!\}$ is less than the largest value in $\{\!\{\mathbf{y}\}\!\}$, or the largest values are the same and, if we eliminate one occurrence of the largest value from both $\{\!\{\mathbf{x}\}\!\}$ and $\{\!\{\mathbf{y}\}\!\}$, the resulting two multisets are ordered. For instance, $\{\!\{1, 1, 1\}\!\}$ is less than $\{\!\{0, 0, 2\}\!\}$ in the multiset order. Even though the rows and columns of a matrix are vectors, it may be useful to ignore the positions but rather concentrate on the values by treating the vectors as multisets. We say that the rows (resp. columns) of a matrix of values are multiset ordered if each row (resp. column), as a multiset, is no greater than the rows (resp. columns) below (resp. to the right of) it.

Let us now analyse the assignments of the equivalence class shown in Figure 1.1. There are exactly two assignments in this class, marked as $\star$, where the rows and columns are lexicographically ordered. On the other hand, there is exactly one assignment, marked as $\dagger$, where the rows and columns are multiset ordered. Similarly, there is exactly one assignment, marked as $*$, where the rows are lexicographically ordered and the columns are multiset ordered; and exactly one assignment, marked as $\bullet$, where the rows are multiset ordered and the columns are lexicographically ordered. Hence, we are able to distinguish the assignments marked as $\star$, $\dagger$, $*$, and $\bullet$ from the rest of the assignments.

An important result in this dissertation is that in any equivalence class of assignments, there is at least one assignment satisfying the properties of those marked as $\star$, $\dagger$, $*$, or $\bullet$ in Figure 1.1. A consequence of this is that we can add extra constraints to the model of our problem which enforce either that the rows and the columns are lexicographically ordered, or that the rows and the columns are multiset ordered, or that the rows are lexicographically ordered and the columns are multiset ordered, or that the rows are multiset ordered and the columns are lexicographically ordered. In this way, among the set of symmetric assignments, only those that satisfy the ordering constraints are chosen for consideration during search for solutions.

The utility of the lexicographic ordering and multiset ordering constraints in breaking row and column symmetries motivates us to devise global constraints to propagate these constraints effectively and efficiently. Propagating a constraint involves examining the domains of the variables participating in the constraint, and removing from the domains those values that cannot be a part of an assignment satisfying the constraint (i.e. pruning). As we can enforce the rows (resp. columns) to be in lexicographic or multiset order by imposing the ordering between the adjacent or all pairs of row (resp. column) vectors, we focus on propagating the ordering constraint posted on a pair of vectors. We devise efficient linear time propagation algorithms for the lexicographic ordering and the multiset ordering constraints by exploiting the semantics of the constraints. We provide theoretical and experimental evidence of the value of the algorithms.

We are now in the position of revisiting the matrix models that we have studied at the beginning of our research, and posting ordering constraints on the rows and columns to break the row and column symmetries. By adding ordering constraints to the matrix models, we identify a new pattern in constraint programs: the lexicographic ordering constraint on a pair of vectors of 0/1 variables together with a sum constraint on each vector. We frequently encounter this pattern in problems involving demand, capacity or partitioning that are modelled using matrices with row and/or column symmetry. A propagation algorithm which exploits the semantics of the lexicographic ordering together with sums can lead to more pruning than the total pruning obtained by the propagation algorithms of the lexicographic ordering constraint and the sum constraint. This motivates us to introduce a new global constraint which combines the lexicographic ordering constraint with two sum constraints. We devise an efficient linear time algorithm to propagate this combination of constraints. Our experimental results show that this new constraint is very useful when the lexicographic ordering constraints conflict with the way we explore the search space. Combining constraints is a step towards tackling one of the drawbacks of using additional constraints to break symmetry.

As theory can only go part of the way in judging the effectiveness of these ordering constraints in breaking row and column symmetries, we finish our research with an empirical study. We perform a wide range of experiments using some of the matrix models we have studied. In each experiment, we have a matrix of decision variables where the rows and/or columns are (partially) symmetric. To break the symmetry, we post ordering constraints on the rows and/or columns, and search for one solution or the best solution according to some criterion. Our results show that these ordering constraints are effective in breaking row and column symmetries as they significantly reduce the size of the search space and the time to solve the problem.

1.5 Summary of Contributions

The contributions of this dissertation to advancing research in CP can be grouped into three categories: identification of common constraint patterns, breaking row and column symmetry, and design and implementation of algorithms for new global constraints.

Identification of Common Constraint Patterns We observe that one common pattern in constraint programs is a matrix model. A wide range of problems can be effectively represented and efficiently solved using a matrix model. This observation leads us to identify two patterns that commonly arise in matrix models: row and column symmetry, and value symmetry.

Row and column symmetry is an important class of symmetries. In a problem with row and column symmetry, the number of symmetries grows super-exponentially and existing methods have difficulty in dealing with large number of symmetries effectively. Therefore, identification of this pattern raises an important research question: how can we tackle row and column symmetries effectively? By showing that value symmetry can be transformed to row/column symmetry, we strengthen the need for developing effective techniques for dealing with row and column symmetries.

Using our approach to breaking row and column symmetry, we identify a new pattern in constraint programs: lexicographic ordering constraints together with sum constraints.

This combination of constraints frequently occur in problems involving demand, capacity or partitioning that are modelled using 0/1 matrices with row and column symmetry.

Identification of common patterns is an important contribution, as it enables modellers to share their expertise and it initiates the development of special techniques to support the patterns. This not only strengthens the power of CP, but also promotes the reach of CP to a wider user base.

Breaking Row and Column Symmetry We propose some ordering constraints which can effectively break the row and column symmetries of a 2-dimensional matrix. In particular, we show that we can enforce the rows and the columns to be lexicographically ordered, or the rows and the columns to be multiset ordered, or the rows to be lexicographically ordered and the columns to be multiset ordered, or the rows to be multiset ordered and the columns to be lexicographically ordered. Whilst adding extra constraints to a model to break symmetry has been previously proposed [Pug93], our novelty is in the investigation of ordering constraints that can consistently be added to a matrix model to break row and column symmetries.

We extend our results to deal with matrices of arbitrary number of dimensions, as well as with matrices that contain partial symmetry or value symmetry. We identify special and useful cases where all row and column symmetries can easily be broken.

We show that the ordering constraints are incomparable both in theory and in practice. We study the effectiveness of the ordering constraints in breaking row and column symmetries both theoretically and experimentally. We show that in theory the ordering constraints may not eliminate all symmetries. We also argue that it is difficult to assess theoretically whether we can significantly reduce the search effort by imposing ordering constraints. Our experimental results reveal that ordering constraints are effective in breaking row and column symmetries as noteworthy reductions are obtained in the search space created and the time taken to solve the problems.

Design and Implementation of Algorithms for New Global Constraints To use the ordering constraints in practice and to support the frequent use of lexicographic ordering constraints together with sum constraints, we devise three new global constraints: lexicographic ordering constraint, lexicographic ordering with sum constraints, and multiset ordering constraint. Each of the global constraints encapsulate a novel linear time algorithm to propagate the constraint. The algorithms of the lexicographic ordering constraint and the lexicographic ordering with sum constraints are optimal. The algorithm of the multiset ordering constraint is optimal provided that the domain size of the variables is less than the vectors' size, which is often the case.

We compare the algorithm of each global constraint with alternative ways of propagating the constraint both theoretically and experimentally. We show that in theory each algorithm is the preferred choice of propagating the associated constraint. In practice, the algorithms of the lexicographic ordering constraint and the multiset ordering constraint propagate effectively and efficiently. The algorithm of the lexicographic ordering with sum constraints is most useful when there is a very large space to explore, such as when the problem is unsatisfiable, or when the way we explore the search space is poor or conflicts with the lexicographic ordering constraints. With the latter result, we tackle one of the drawbacks of using constraints to break symmetry. We show that a constraint which combines together symmetry breaking and problem constraints can give additional pruning and this can help compensate for the search heuristic trying to push the search in a different direction to the symmetry breaking constraints.

1.6 Organisation of the Dissertation

This dissertation is organised as follows.

Chapter 1, Introduction. We introduce our research. We first briefly introduce constraint programming. Then, we describe the motivations and the goals, present an overview, and finally summarise the major contributions of the dissertation.

Chapter 2, Formal Background. We report on the background knowledge necessary to read the dissertation. We briefly present constraint satisfaction problems, some techniques to solve such problems, global constraints, symmetry in constraint satisfaction problems, and ordering relations. We also present the notations we use throughout the dissertation.

Chapter 3, Matrix Models. We recognise the central role played in constraint programs by matrix models. We identify that row and column symmetry, and value symmetry are two patterns that frequently arise in matrix models. We discuss the importance of tackling row and column symmetry.

Chapter 4, Breaking Row and Column Symmetry. We tackle row and column symmetry. We investigate what ordering constraints we can post on the rows and columns of a matrix to break row and column symmetries. We compare the ordering constraints that we put forward and discuss their effectiveness in breaking row and column symmetries from a theoretical view point. We extend our results in a number of ways. We also identify special and useful cases where all row and column symmetries can easily be broken. Finally, we compare with related work.

Chapter 5, Lexicographic Ordering Constraint. We propose an optimal linear time algorithm to propagate the lexicographic ordering constraint. We extend the algorithm in a number of ways. We discuss alternative ways of propagating the constraint, compare with related work, and provide both theoretical and experimental evidence of the value of the algorithm.

Chapter 6, Lexicographic Ordering with Sum Constraints. We introduce a new global constraint which combines the lexicographic ordering constraint with two sum constraints. We show that this combination of constraints is another common pattern in constraints programs, and provide theoretical evidence of the value of combining the constraints. We propose an optimal linear time algorithm to propagate this new constraint. We extend the algorithm in a number of ways, and compare with related work. We experimentally evaluate the algorithm and discuss when it is most useful. 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.

Chapter 7, Multiset Ordering Constraint. We propose an efficient linear time algorithm to propagate the multiset ordering constraint. We extend the algorithm in a number of ways. We propose an alternative algorithm which is useful when the domains are large. We discuss alternative ways of propagating the constraint, and provide both theoretical and experimental evidence of the value of the algorithm.

Chapter 8, Symmetry Breaking with Ordering Constraints. We carry out an experimental study. We show that the ordering constraints are effective in breaking row and column symmetries in practice. The results of the theoretical comparison of the ordering constraints are confirmed by the experimental results.

Chapter 9, Conclusions and Future Work. We bring the dissertation to a conclusion. We present our contributions, discuss the lessons learnt, summarise the limitations of our work, present our directions for future research, and finally conclude.

Chapter 2

Formal Background

In this chapter, we familiarise the reader with the concepts and notations used throughout the dissertation. In particular, we introduce constraint satisfaction problems in Section 2.1, and briefly explain in Section 2.2 how such problems are solved. Then, in Section 2.3, we introduce global constraints and point out the major approaches to designing algorithms for global constraints. We define symmetry in constraint satisfaction problems and also discuss the various approaches to breaking symmetry in Section 2.4. Finally, before giving our notations in Section 2.6, we present ordering relations in Section 2.5.

2.1 Constraint Satisfaction Problems

A *constraint* is a relation among several unknowns (or variables), each taking a value in a given domain. Constraints thus restrict the possible values that the variables can simultaneously take. A constraint can be defined on any number of variables. If a constraint affects only one variable then the constraint is *unary*, and if it affects two variables then the constraint is *binary*. A constraint which is defined on $n > 2$ variables is a *non-binary* or n-ary constraint.

Constraint satisfaction problems play a central role in various fields of computer science [Tsa93] and are ubiquitous in many real-life application areas such as production planning, staff scheduling, resource allocation, circuit design, option trading, and DNA sequencing.

Definition 1 *A* constraint satisfaction problem *(CSP) consists of:*

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

A CSP is thus described by $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{D} = \{\mathcal{D}(X_1), \ldots, \mathcal{D}(X_n)\}$. A variable *assignment* or *instantiation* is an assignment to a variable X_i of one of the values from $\mathcal{D}(X_i)$. Whilst a *partial assignment* A to $\mathcal{X}$ is an assignment to some but not all $X_i \in \mathcal{X}$, a *total assignment*[1] A to $\mathcal{X}$ is an assignment to every $X_i \in \mathcal{X}$. We use the notation $A[\mathcal{S}]$ to

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

denote the projection of A on to the set of variables $\mathcal{S}$. A (partial) assignment A to the set of variables $\mathcal{T} \subseteq \mathcal{X}$ is *consistent* iff for all constraints $c(X_i, \ldots, X_j) \in \mathcal{C}$ such that $\{X_i, \ldots, X_j\} \subseteq \mathcal{T}$, we have $A[\{X_i, \ldots, X_j\}] \in c(X_i, \ldots, X_j)$. A *solution* to the CSP is a consistent assignment to $\mathcal{X}$. Typically, we are interested in finding one or all solutions, or an optimal solution given some objective function. In the presence of an objective function, a CSP is a *constraint optimisation problem* (COP) .

In general, solving CSPs is NP-hard and so is computationally intractable [Mac77]. *Constraint programming* (CP) provides a platform for solving CSPs [MS98][Apt03] and has proven successful in many real-life applications [Wal96][Ros00] despite this intractability. To solve a problem using CP methods, we need first to formulate it as a CSP by declaring the variables, their domains, as well as the constraints on the variables. This part of the problem solving is called *modelling*.

2.2 Search, Local Consistency and Propagation

Solutions to CSPs can be found by *searching* systematically through the possible assignment of values to variables. A search algorithm traverses the space of partial assignments and attempts to build up a solution. A common strategy for exploring the search space is *backtracking search* (see [KvB97] for a detailed presentation). A backtracking search traverses the space of partial assignments in a depth-first manner, and at each step it extends a partial assignment by assigning a value to one more variable. If the extended assignment is consistent then one more variable is instantiated and so on. Otherwise, the variable is re-instantiated with another value. If none of the values in the domain of the variable is consistent with the current partial assignment then one of the previous variable assignments is reconsidered. This process is called *backtracking*. The simplest backtracking search algorithm is the *chronological backtracking*, which backtracks to the most recently instantiated variable [BR75].

A backtrack search may be seen as a *search tree* traversal. In this approach, each node defines a partial assignment and each branch defines a variable assignment. A partial assignment is extended by branching from the corresponding node to one of its subtrees by assigning a value j to the next variable X_i from the current $\mathcal{D}(X_i)$. Upon backtracking, j is removed from $\mathcal{D}(X_i)$. This process is often called *labelling*. The order of the variables and values chosen for consideration can have a profound effect on the size of the search tree [HE80]. The order can be determined before search starts, in which case the labelling heuristic is *static*. If the next variable and/or value are determined during search then the labelling heuristic is *dynamic*.

The size of the search tree of a CSP is the product of the domain sizes of all variables and is thus too big in general to enumerate all possible assignments using a naive backtracking algorithm. Therefore, many CP solution methods are based on *inference* which reduces the problem to an equivalent (i.e. with the same solution set) but smaller problem by making implicit constraint information explicit. If the inference is complete then satisfiability is determined immediately. Since complete inference is computationally expensive to be used in practice, inference methods are often incomplete. This requires further search. On the other hand, the reduced problem is presumably easier to solve and has a smaller search tree. Incomplete inference is often referred to as *local consistency*.

Local consistencies are properties of CSPs defined over "local" parts of the CSP, in other words properties defined over subsets of the variables and constraints of the CSP. Let $\mathcal{S}$ and $\mathcal{T}$ be two distinct sets of variables $\mathcal{S}, \mathcal{T} \subset \mathcal{X}$ in a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, and A be a consistent assignment to $\mathcal{T}$. A *support* A' of A for $\mathcal{S}$ is an assignment A' to $\mathcal{S}$ such that $A \wedge A'$ is a consistent assignment. If A is not supported then A is *inconsistent* and does not belong to any solution. The main idea behind the local consistency techniques is to infer such inconsistent partial assignments to $\mathcal{X}$ and exclude them from the search tree.

Even though there exist many local consistency notions in the literature (see [Apt03] for a list of references), in this dissertation we restrict our attention to node consistency, (generalised) arc-consistency, and bounds consistency. These consistencies are concerned with the support of partial assignments having only one variable. Problem reduction can thus be performed by removing from the domains those values which lack support and are thus inconsistent.

Definition 2 *A unary constraint is* node consistent *iff its variable has a non-empty domain and every value in the domain satisfies the constraint.*

Definition 3 ([Fre85]) *A binary constraint is* (i, j)-consistent *iff its variables have non-empty domains and any consistent assignment of i variables can be extended to a consistent assignment involving j additional variables.*

One of the most widely used local consistency techniques for binary constraints is arc-consistency (AC) and is defined as follows.

Definition 4 *A binary constraint is* arc-consistent *iff it is* $(1, 1)$*-consistent.*

A general notion of arc-consistency, which is not restricted to binary constraints, is generalised arc-consistency (GAC).

Definition 5 ([MM88]) *A constraint is* generalised arc-consistent *iff its variables have non-empty domains, and for any value in the domain of a variable, there exist consistent values in the domains of the other variables.*

A weaker form of local consistency which is defined only for totally ordered domains is bounds consistency (BC).

Definition 6 ([vHSD98]) *A constraint whose variables have totally ordered domains is* bounds consistent *iff its variables have non-empty domains, and for the minimum and the maximum values in the domain of a variable, there exist consistent values in the domains of the other variables.*

Different local consistency properties are compared in [DB97]. Let A and B be two local consistency properties. We say that:

- A is *as strong as* B (written $A \leadsto B$) iff in any problem in which A holds we have that B holds;

- A is *stronger than* B iff $A \leadsto B$ but not $B \leadsto A$;

- A is *incomparable* with B iff neither $A \leadsto B$ nor $B \leadsto A$;

- A is *equivalent* to B iff both $A \leadsto B$ and $B \leadsto A$.

Local consistency properties are generally not sufficient conditions for CSPs to be satisfiable. For instance, a value in a domain may not be a part of a solution even though all the constraints are arc-consistent. This is because a CSP typically has several constraints that need to be satisfied simultaneously, whereas consistency is local to each constraint and its variables.

In a constraint program, every variable participates in zero or more constraints. Searching for solutions is interleaved with local consistency which is first enforced to preprocess the problem to prune subsequent search and then maintained during search at each node of the search tree with respect to the current variable assignment. In this way, the domains of the uninstantiated variables shrink and the search tree gets smaller. This result of maintaining local consistency during search can be explained as follows. When the domain of a variable X_i is modified (due to either an instantiation or removal of inconsistent values), values in the domain of the other variables participating in the same constraints as X_i might lose their support and become inconsistent. If this is the case then such constraints has to be examined and local consistency needs to be established if necessary. The process of examining a constraint is called *propagation*, and a modification to a variable which leads to propagation is called a *propagation event.* Propagation may result in removal of inconsistent values, which is also known as *pruning* or *filtering.* This change may lead to further inconsistencies and prunings, hence the result of any modification is gradually propagated through the entire CSP. Finally, this process terminates. We then have three possible situations: (1) a domain becomes empty and thus a failure occurs; (2) a solution is found; (3) there exists one variable which is not ground (i.e. uninstantiated) and all the constraints are locally consistent. In the first case, the current branch is pruned and backtracking occurs as there exist no solutions under this branch. In the third case, the search tree which is below the current node is explored, as no solutions are yet found. If the domains have shrunk during propagation, the new search tree is smaller.

2.3 Global Constraints

The constraints arising in the real-life CSPs are often “complex” and non-binary. It is generally hard to decompose such constraints into simple binary constraints like $=$, $\neq$, $\leq$, $<$, etc. Even if such a decomposition is possible, the total pruning obtained by the propagation of each simpler constraint is likely to be weak, as the global view of the constraint is lost in the decomposition.

A *global constraint* is a predicate which involves more than 2 variables, and encapsulates a generic or specialised *filtering algorithm* which is used to propagate the constraint. Three notions of globality are introduced in [BvH03]: semantic globality, operational globality, and algorithmic globality. A constraint C is *semantically global* if there exist no constraint decomposition scheme for C. In this case, it is impossible to state C by a conjunction of simpler constraints and thus the global constraint facilitates problem formulation. A constraint C is *operationally* Φ*-global* if there exist no constraint decomposition scheme on the variables of C for which the local consistency notion Φ removes as many local inconsistencies as on C. In this case, a filtering algorithm which maintains Φ on C provides powerful pruning. When a constraint C is not operationally global wrt a local consistency notion Φ, then there is no benefit in using C in a problem formulation on which Φ is used. However, if a filtering algorithm which maintains Φ on C provides complexity and efficiency advantages, then C is *algorithmically* Φ*-global.*

Many useful global constraints have been proposed in the last ten years [Bel02]. An example is the *all-different* constraint:

$$\textit{all-different}(\langle X_1, \ldots, X_n \rangle)$$

which holds iff no pair of variables in $\langle X_1, \ldots, X_n \rangle$ are assigned the same value. An extension of the *all-different* constraint is the *global cardinality constraint* (*gcc*):

$$\textit{gcc}(\langle X_1, \ldots, X_n \rangle, \langle v_1, \ldots, v_m \rangle, \langle \langle l_1, u_1 \rangle, \ldots, \langle l_m, u_m \rangle \rangle)$$

which holds iff the number of variables in $\langle X_1, \ldots, X_n \rangle$ assigned to v_i is between l_i and u_i for all $i \in [1, m]$.

The filtering algorithm of a global constraint is either a generic algorithm, or a specialised algorithm which exploits the semantics of the constraint.

2.3.1 Generic Algorithms

Bessière and Régin have defined GAC-schema [BR97] which is a general framework for AC algorithms. GAC-schema is based on AC-7 arc-consistency algorithm for binary constraints [BFR99] and allows enforcing GAC on constraints of arbitrary arity. There are three instantiations of the schema to handle constraints defined by a set of allowed tuples, by a set of forbidden tuples, and by a predicate whose semantics is unknown.

Bessière and Régin have also defined a schema for enforcing GAC on an arbitrary conjunction of constraints [BR98]. This form of local consistency is called *conjunctive consistency*. The proposed filtering algorithm to achieve conjunctive consistency is based on the instantiation of GAC-schema for constraints given as a predicate. Since the semantics of the predicate is not known in such a general setting, finding a support for a value a of a variable V is done by searching for a solution for the problem derived when V is assigned a.

Bessière and Régin later proposed another instantiation of the GAC-schema, in which the related constraints are grouped together to form subproblems [BR99]. A subproblem is thus a global constraint which is a conjunction of constraints. The main idea is that maintaining GAC on subproblems independently may enhance the process of making the whole problem GAC. The advantage over the previous instantiations of the GAC schema is that the knowledge about the constraints in a subproblem can be taken into account in a generic way, and that allowed tuples/solutions are computed "on the fly" by means of a search algorithm.

The algorithms provided by the schemas are generic and can be used to achieve GAC on any constraint. This is clearly a big advantage, as it eliminates the need of a new filtering algorithm provided that an algorithm checking the consistency of a value is given. However, the generic algorithms are not efficient as generality comes with a high cost. For instance, the worst-case time complexity of the GAC-schema is $O(ed^k)$ where d is the size of the largest domain, k is the maximal arity of the constraints, and e is the total number of constraints. The high complexity of the algorithms restrict their use in practice.

2.3.2 Specialised Algorithms

The generic GAC algorithms of the GAC-schema and the schema for a conjunction of constraints are too costly to be useful in practice, as they do not take into account any specific knowledge. Several specialised algorithms for global constraints have been proposed in the past years, which exploit the semantics and the structure of the constraints in concern. Thus, many global constraints in the various CP toolkits have their own specific filtering algorithm, which typically achieve GAC at a lower cost compared to that of a generic GAC algorithm. As an example, Régin in [Rég94] gives a filtering algorithm for the *all-different* constraint which maintains GAC in time $O(n^{2.5})$ where n is the number of variables. The algorithm exploits the nice correspondence between the semantics of the constraint and the maximum matching problem in graph theory. The success of this global constraint has lead to several alternative algorithms which maintain weaker forms of local consistency but are very fast in practice (e.g. [Pug98][LQTvB03]). A survey on the various algorithms for the *all-different* constraint can be found in [vH01]. As another example, Régin in [Rég96] makes use of the flow theory and proposes an $O(n^2 d)$ GAC algorithm for the *gcc* constraint, where n is the number of variables and d is the size of the largest domain. This complexity is improved in an alternative GAC algorithm [QvBL+03]. Weaker but faster algorithms have been developed in [QvBL+03], [KT03a], and [KT03b]. The algorithms mentioned so far have all shown to be successful in tackling hard problems.

The semantics of a constraint helps in devising an efficient filtering algorithm in many ways. First, supports for values can be found quickly. The support of a value in the domain of a variable can be the support also for some other values in the domain. Second, inconsistent values can be identified without having to explore every value in the domain of each variable. Third, the calls to the algorithm can be limited to the necessary propagation events. Fourth, failure or success can be detected earlier. A constraint C is *entailed* when any assignment of values to its variables satisfy C, in which case C is $true$; whereas C is *disentailed* when any assignment of values to its variables fail to satisfy the constraint, in which case C is $false$. If a constraint is entailed then there is no need to propagate the constraint, and if it is disentailed then failure can immediately be established without having to wait until a domain wipe-out occurs. Finally, repeated computation can be avoided if identification of inconsistent values can be done incrementally.

2.4 Symmetry

A *symmetry* is a transformation of an entity which preserves the properties of the entity. The transformed entity is thus identical to and indistinguishable from the original entity. For instance, rotating a chess board 90° gives us a board which is indistinguishable from the original board. Regarding CSPs, it is possible that a CSP has symmetries in the variables or domains or both which preserve satisfiability. In the presence of symmetry, any (partial) assignment can be transformed into a set of symmetrically equivalent assignments without affecting whether or not the original assignment satisfies the constraints.

Definition 7 ([MT01]) *A symmetry σ of a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a sequence of $n+1$ bijective functions $\sigma_0, \sigma_1, \ldots, \sigma_n$ where:*

$$\begin{array}{l} \sigma_0 : \mathcal{X} \rightarrow \mathcal{X} \\ \sigma_i : \mathcal{D}(X_i) \rightarrow \mathcal{D}(\sigma_0(X_i)) \end{array}$$

such that for any (partial) assignment p to $\mathcal{X}$, p satisfies the constraints in $\mathcal{C}$ iff σ applied to the variables and their assignments in p does.

Symmetries divide the set of possible assignments into equivalence classes. A *symmetry class* is an equivalence class of assignments. Two assignments are equivalent if there is a symmetry mapping one assignment into the other. Symmetry may be inherent in the problem. Alternatively, it may be introduced as a result of modelling by distinguishing the objects which are in fact indistinguishable.

Symmetry in a CSP creates symmetric but essentially equivalent states in its search space. Visiting symmetric states can significantly slow down the search process. If a state leads to a solution then all its symmetric states also do, and these symmetric solutions are indistinguishable from each other. This could be a big problem when, for instance, proving optimality because it is worthless to traverse repeatedly the states which do not lead to any improvement in the objective function. Even if we are interested in one solution, we may explore many failed and symmetrically equivalent states before finding a solution. Therefore, problems often become very difficult to solve in the presence of symmetry.

Symmetry in a CSP can be exploited to improve the search efficiency. This can be done by pruning the symmetric parts of the search space. This process is often referred to as *symmetry breaking*. As can be witnessed by the proceedings of the recent CP conferences [Wal01][vH02][Ros03] and symmetry workshops [FP01][FP02b][GHS03], there has been an increasing interest in breaking symmetry in CSPs. The approaches taken fall into four categories: remodelling the problem, posting symmetry breaking constraints, modification of the search algorithm, and using symmetries to guide the search.

2.4.1 Remodelling

One way of symmetry breaking is to remodel the problem to obtain a model with less symmetries. There are no general guidelines on how to do this, as it all depends on the problem and its model. However, it has been recognised that modelling a group of indistinguishable objects using a set variable [Ger97] does not distinguish between the objects and thus avoids introducing unnecessary symmetry. An example is Stefano Novello's model (available at `www.icparc.ic.ac.uk/eclipse/examples/`) of the social golfers problem which is about scheduling $g * s$ golfers to play in g groups, each of size s, in each of w weeks. The model uses a set variable to represent each group of golfers in each week. Since the order of the players within a group are not important, this model has less symmetries compared to the one where each group is represented by a vector of integer variables. Smith in [Smi01] discusses other models of this problem with even less symmetries.

Modelling a problem in such a way that no symmetries are introduced may not be easy due to the lack of proper constructs in the modelling language. For instance, in the social golfers problem, each of the weeks, groups, and golfers are indistinguishable. As the golfers are partitioned into g groups in every week, we can model the problem as a set of w partitions (i.e., a set of sets of sets), which does not distinguish between the weeks, between the groups, and between the players. Unfortunately, such constructs are not yet available in any of the modelling languages. Recent progress in this direction is, however, promising [BFM03].

2.4.2 Symmetry Breaking Constraints

Another approach to symmetry breaking is to add extra constraints to the model of the problem [Pug93]. These constraints impose an ordering on the symmetric objects.

As a consequence, among the set of symmetric assignments, only those that satisfy the ordering constraints are chosen for consideration during the process of search. Since such ordering constraints are not a requirement of the problem but rather serve for symmetry breaking, they are called *symmetry breaking constraints*. Care needs to be taken when posting symmetry breaking constraints. The goal is to define reductions in the space of solutions in a such a way that the reduced space contains at least one solution from each equivalence class of solutions. If symmetry breaking constraints remove all the elements of an equivalence class of assignments then solutions can be lost.

Formulating appropriate symmetry breaking constraints is often done by hand [Pug93]. Crawford *et al.* compute symmetry breaking constraints automatically via a problem independent transformation [CGLR96]. The generated symmetry breaking constraints are satisfied by exactly one element, which is the smallest element, in each equivalence class of assignments. Even though such an approach guarantees that the search algorithm never visits two states in the search space that are symmetrically equivalent, the complexity results suggest that generating the constraints will be intractable in the general case. Puget in [Pug03b] states that it is not necessary to impose all the symmetry breaking constraints at the start of search for solutions. His approach is to add some of the symmetry breaking constraints during search. At each node of the search tree, only the constraints for symmetries that are not yet broken are imposed.

2.4.3 Modification of Search Algorithm

There has been a great deal of interest in modifying the backtracking search algorithms to break symmetry. The aim is to avoid exploring any part of the search tree that is symmetric to a part which is already considered. To the best of our knowledge, the initial work in this direction appears in [BFP88][BFP96], where each state of the search tree is checked whether it is an appropriate representative of all its symmetric states. The algorithms proposed are variations of a colour automorphism algorithm and make use of computational group theory. Even though the context is not CSPs, this approach can be seen as the ancestor of the related methods to break symmetry in CSPs.

Earlier work on breaking symmetry in CSPs by modifying the search algorithm focused on specific forms of symmetry. Freuder in [Fre91] introduces the notion of interchangeable values. Benhamou in [Ben94] discusses permutation of domain values in binary constraints. Both types of symmetries partition domain values into equivalence classes, and thus the approaches taken to break such symmetries are similar: whenever a value v is proved to be inconsistent for a variable V, all values belonging to the same equivalence class as v are pruned from the domain of V, as they will also be proved inconsistent for V in the future. In the context of permutable variables, Roy and Pachet propose a related pruning algorithm [RP98]: whenever a value v is proved to be inconsistent for a variable V, v is pruned from the domain of the variables permutable with V. All these approaches can be seen as particular cases of a more general strategy presented in [MT99].

Recent work on breaking symmetry in CSPs by modifying the search algorithm can be applied to arbitrary symmetries. The fundamental principle is to use every completely traversed subtree as a nogood[2] to prevent the exploration of any of its symmetrical variants. This principle is applied in two different ways. In SES [BW99][BW02] and SBDS [GS00], whenever a subtree is completely explored, a constraint for each symmetry of the problem is imposed to exclude all symmetric subtrees from consideration later in the

[2] A nogood is an assignment which cannot appear in any unenumerated solution.

search. Such constraints are enforced according to the current partial assignment. On the other hand, in SBDD [FSS01][FM01], a check is performed at each node of the search tree to see whether it is dominated by a symmetrically equivalent subtree already explored. If this is the case then the branch is pruned. Dominance checks are problem dependent.

2.4.4 Heuristic Methods

Meseguer and Torras take a heuristic approach to symmetry breaking [MT99][MT01]. The idea is to define a search heuristic which directs the search towards the subspaces with a high density of non-symmetric states. The authors propose a variable ordering heuristic which selects first the variables involved in the largest number of symmetries local to the current state. This greedy heuristic tries to maximise the total number of broken symmetries at each level of the search. It is argued that the more symmetries are broken the less unproductive effort a lookahead[3] performs. Moreover, a variable involved in many symmetries can have "good" values in its domain.

The proposed heuristic can also be combined with the popular *smallest-domain first* principle. The result of such a combination is a new heuristic for variable selection, which is shown to be more effective than each of the heuristics involved in the combination.

2.5 Ordering Relations

As symmetries are often broken by ordering symmetric objects, we now introduce some notions about orderings.

Given two sets $\mathcal{A}$ and $\mathcal{B}$, a subset $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ is a *binary relation* from $\mathcal{A}$ to $\mathcal{B}$. For each pair $(a, b) \in \mathcal{R}$, "a is related to b" via $\mathcal{R}$. If $\mathcal{A} = \mathcal{B} = \mathcal{S}$ then a relation $\mathcal{R}$ from $\mathcal{S}$ to $\mathcal{S}$ is a *binary relation on* $\mathcal{S}$. If $\mathcal{R}$ is a binary relation on $\mathcal{S}$ then $(a, b) \in \mathcal{R}$ is also denoted by $a\ \mathcal{R}\ b$, likewise $(a, b) \notin \mathcal{R}$ is denoted by $a\ \not{\mathcal{R}}\ b$. There are various properties of a binary relation.

Definition 8 *A binary relation $\mathcal{R}$ on $\mathcal{S}$ is:*

- reflexive *iff $(s, s) \in \mathcal{R}$ for all $s \in \mathcal{S}$;*
- irreflexive *iff $(s, s) \notin \mathcal{R}$ for all $s \in \mathcal{S}$;*
- antisymmetric *iff $(s, t) \in \mathcal{R}$ and $(t, s) \in \mathcal{R}$ implies $s = t$ for all $s, t \in \mathcal{S}$;*
- transitive *iff $(r, s) \in \mathcal{R}$ and $(s, t) \in \mathcal{R}$ implies $(r, t) \in \mathcal{R}$ for all $r, s, t \in \mathcal{S}$.*

A binary relation $\preceq$ on $\mathcal{S}$ is a *partial ordering* iff it is reflexive, antisymmetric, and transitive. Hence, $s \preceq t$ denotes that a member s of $\mathcal{S}$ either precedes another member t of $\mathcal{S}$, or $s = t$. A set $\mathcal{S}$ with a partial ordering relation $\preceq$ is called a *partially ordered set*, and is denoted by $(\mathcal{S}, \preceq)$. Indeed, the word *ordering* implies that the objects in the set are ordered according to some properties or criteria (e.g., smaller, larger, inferior, superior, etc). Note that not all pairs of members of $\mathcal{S}$ may be related by $\preceq$.

Each partial ordering $\preceq$ on a set $\mathcal{S}$ determines a converse relation $\succeq$, where $s \preceq t$ iff $t \succeq s$. The converse relation $\succeq$ is also a partial ordering. Given a partial ordering $\preceq$ on a set $\mathcal{S}$, we can define another binary relation $\prec$ on $\mathcal{S}$ by $s \prec t \ \leftrightarrow \ s \preceq t \ \wedge \ s \neq t$.

[3] A lookahead is a search algorithm which enforces a local consistency property during search.

The relation $\prec$ is irreflexive, antisymmetric, and transitive, and is called a *strict partial ordering.* A set $\mathcal{S}$ with a strict partial ordering $\prec$ is called a *strict partially ordered set* and is denoted by $(\mathcal{S}, \prec)$. Each partial ordering $\preceq$ on a set $\mathcal{S}$ yields a strict partial ordering $\prec$, and conversely given $\prec$ on a set $\mathcal{S}$, we can define $\preceq$ on $\mathcal{S}$ by $\; s \preceq t \;\; \leftrightarrow \;\; s \prec t \; \vee \; s = t.$

Two elements s and t of a partially ordered set $(\mathcal{S}, \preceq)$ are comparable if either $s \prec t$, or $t \prec s$, or $s = t$. Otherwise they are incomparable. If every two elements of a partially ordered set $(\mathcal{S}, \preceq)$ are comparable then $\preceq$ is a *total ordering* on $\mathcal{S}$, and $(\mathcal{S}, \preceq)$ is called a *totally ordered set.* Likewise, if every two elements of $(\mathcal{S}, \prec)$ are comparable then $\prec$ is a *strict total ordering* on $\mathcal{S}$, and $(\mathcal{S}, \prec)$ is called a *strict totally ordered set.* For two elements s and t of a totally ordered set $(\mathcal{S}, \preceq)$, $s \not\preceq t$ iff $s \succ t$, as s and t are comparable, and either $s \prec t$ or $t \prec s$, or $s = t$ holds.

A binary relation $\preceq$ on $\mathcal{S}$ is a *preordering* iff it is reflexive and transitive. Each preordering $\preceq$ on a set $\mathcal{S}$ determines a converse relation $\succeq$, where $s \preceq t$ iff $t \succeq s$. The converse relation $\succeq$ is also a preordering. Given a preordering $\preceq$ on a set $\mathcal{S}$, we can define another binary relation $\prec$ on $\mathcal{S}$ by $\; s \prec t \;\; \leftrightarrow \;\; s \preceq t \; \wedge \; s \neq t.$ The relation $\prec$ is irreflexive and transitive, and is called a *strict preordering.* Each preordering $\preceq$ on a set $\mathcal{S}$ yields a strict preordering $\prec$, and conversely given $\prec$ on a set $\mathcal{S}$, we can define $\preceq$ on $\mathcal{S}$ by $\; s \preceq t \;\; \leftrightarrow \;\; s \prec t \; \vee \; s = t.$

2.6 Notation

Throughout the dissertation, we assume finite integer domains, which are totally ordered. A variable designates a constrained variable that is bound to a finite non-empty integer domain. The domain of a variable V is denoted by $\mathcal{D}(V)$, and the minimum and the maximum elements in this domain by $min(V)$ and $max(V)$.

If a variable V is assigned a value a, then we denote this by $V \leftarrow a$. If two variables V and W are assigned the same values, that is the variables are now ground and equal, then we write $V \doteq W$, otherwise we write $\neg(V \doteq W)$. If V is assigned a value less than that assigned to W, then we write $V \lessdot W$. Similarly, we write $V \gtrdot W$ if V is assigned a value greater than that assigned to W.

A vector is an ordered list of elements. We denote a vector of n variables as $\vec{X} = \langle X_0, \ldots, X_{n-1} \rangle$, while we denote a vector of n integers as $\vec{x} = \langle x_0, \ldots, x_{n-1} \rangle$, where $\vec{X}$ or $\vec{x}$ can be any letter in the alphabet with an arrow on top. Unless otherwise stated, the indexing of vectors is from left to right, with 0 being the most significant index, and the variables of a vector $\vec{X}$ are assumed to be disjoint and not repeated.

The sub-vector of some $\vec{x}$ with start index a and last index b inclusive is denoted by $\vec{x}_{a \to b}$. Similarly, the sub-vector of some $\vec{X}$ with start index a and last index b inclusive is denoted by $\vec{X}_{a \to b}$. The vector $\vec{X}_{X_i \leftarrow d}$ is the vector $\vec{X}$ with some X_i being assigned to d. The functions $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{X})$ assign all the variables of $\vec{X}$ their minimum and maximum values, respectively.

A vector $\vec{x}$ in the domain of $\vec{X}$ is designated by $\vec{x} \in \vec{X}$. We write $\{\vec{x} \mid \; C \; \wedge \; \vec{x} \in \vec{X}\}$ to denote the set of vectors in the domain of $\vec{X}$ which satisfy condition C. Consequently, (1) $min\{\vec{x} \mid \; \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ and $max\{\vec{x} \mid \; \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ are the minimum and maximum vectors in the domain of $\vec{X}$ such that $\sum_i x_i = Sx$; (2) $min\{\vec{x} \mid \; x_i = k \wedge \vec{x} \in \vec{X}\}$ and $max\{\vec{x} \mid \; x_i = k \; \wedge \; \vec{x} \in \vec{X}\}$ are the minimum and maximum vectors in the domain of $\vec{X}$ such that some x_i is k; (3) $min\{\vec{x} \mid \; \sum_i x_i = Sx \; \wedge \; x_j = k \; \wedge \; \vec{x} \in \vec{X}\}$ and $max\{\vec{x} \mid \; \sum_i x_i = Sx \; \wedge \; x_j = k \; \wedge \; \vec{x} \in \vec{X}\}$ are the minimum and maximum vectors in the domain of $\vec{X}$ such that $\sum_i x_i = Sx$ and some x_j is k.

In the case of multiple vectors, n vectors of m finite integer domain variables are denoted by:

$$\begin{array}{ccllll} \vec{X}_0 & = & \langle X_{0,0}, & X_{0,1}, & \ldots & , X_{0,m-1}\rangle \\ \vec{X}_1 & = & \langle X_{1,0}, & X_{1,1}, & \ldots & , X_{1,m-1}\rangle \\ \vdots & & & & & \\ \vec{X}_{n-1} & = & \langle X_{n-1,0}, & X_{n-1,1}, & \ldots & , X_{n-1,m-1}\rangle \end{array}$$

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

An occurrence vector $occ(\vec{x})$ associated with $\vec{x}$ is indexed in decreasing order of significance from the maximum $max\{\!\{\vec{x}\}\!\}$ to the minimum $min\{\!\{\vec{x}\}\!\}$ value from the values in $\{\!\{\vec{x}\}\!\}$. The ith element of $occ(\vec{x})$ is the number of occurrences of $max\{\!\{\vec{x}\}\!\} - i$ in $\{\!\{\vec{x}\}\!\}$. When comparing two occurrence vectors, we assume they start and end with the occurrence of the same value, adding leading/trailing zeroes as necessary.

Finally, $sort(\vec{x})$ is the vector obtained by sorting the values in $\vec{x}$ in non-increasing order.

Chapter 3

Matrix Models

3.1 Introduction

An important and active research area of CP is modelling. To solve a problem using CP methods, we need first to model it as a CSP by specifying the decision variables, their domains, and the constraints on the variables. There are often many alternatives for each modelling decision, and a small modification to a model can have a huge impact on the efficiency of the solution method [Bor98][BT01][Hni03]. Hence, formulating an effective model for a given problem requires considerable skills in modelling and remains a challenging task even for modelling experts [Fre98].

We have observed that there are many recurring patterns in constraint programs [Wal03]. Identification of these patterns has two important benefits. First, patterns can be used to pass on modelling expertise. Second, special-purpose methods to support the patterns can be devised for use by a (non-expert) modeller. This helps tackle the difficulty of effective modelling and makes CP reachable to a wider audience.

One common pattern in constraint programs is a matrix model. Any formulation of a problem as a CSP which employs one or more matrices of decision variables is a matrix model. A wide range of problems can be effectively represented and efficiently solved using a matrix model. There are patterns that arise commonly within matrix models and this chapter identifies two. The first common pattern is row and column symmetry, which arises in 2-dimensional matrices where rows and columns represent indistinguishable objects and are therefore symmetric. The second pattern is value symmetry, which arises in matrices where the values taken by the variables are indistinguishable and are therefore symmetric.

This chapter is organised as follows. In Section 3.2, we show that a matrix model is a common pattern in constraint programs by going through some problems taken from a wide range of real-life application domains, namely combinatorics, design, configuration, scheduling, timetabling, bioinformatics, and code generation. We present an effective matrix model for each problem, give the intuition behind it, and point to the associated research work if the model has previously been proposed. Then, in Section 3.3, we identify two common patterns in matrix models: row and column symmetry, and value symmetry. We discuss why row and column symmetries are important and why techniques to deal with row and column symmetries are useful. Finally, we summarise in Section 3.4.

3.2 Matrix Models

A *matrix model* is the formulation of a problem as a CSP with one or more matrices of decision variables. Matrix models are a natural way to represent functions and relations [Hni03]. For example, assume we wish to assign papers for review to the programme committee (PC) members of a scientific conference. As there are typically more papers to be reviewed than the number of PC members, we want to assign multiple papers to a PC member who has a certain capacity. Also, we want a paper to be reviewed by more than one PC member to judge fairly the scientific quality of the paper. Hence, we are looking for a relation between the set of PC members and the set of papers. This can be modelled by a 2-dimensional 0/1 matrix X, where a variable $X_{i,j}$ is assigned 1 iff paper j is assigned to PC member i.

In this example, one matrix is sufficient to model the problem. Sometimes, a matrix model contains multiple matrices of variables. Channelling constraints can be used to link the different matrices together [CCLW99]. For instance, consider the problem of assigning papers to PC members, and assume every PC member has a cost described by the amount we need to pay her for reviewing papers. To minimise the total cost, we need to choose a subset of the PC members. In this case, we introduce a 1-dimensional 0/1 matrix Y, where a variable Y_i is assigned 1 iff PC member i is assigned at least one paper. The cost function is then $\sum_i Y_i * A_i$, where A is the 1-dimensional matrix of the cost of each PC member. We link the variables of X and Y by imposing:

$$\forall i\,.\ \forall j\,.\ X_{i,j} = 1 \rightarrow Y_i = 1$$

Matrix models have been long advocated in 0/1 ILP [Sch86] and are commonly used in CP. Of the 38 problems of CSPLib [CSP] on September 22, 2003, at least 33 of the 38 have matrix models, most of them already published and proved successful [FFH$^+$01b]. In this section, we explore a number of problems taken from 7 different real-life application domains, and show how each can be effectively represented by a matrix model.

The matrices of a matrix model can have an arbitrary number of dimensions. In what follows, we index each dimension of a matrix by the elements taken from a set of values. Hence, an n-dimensional (n-d) matrix of $\mathcal{S}_0 \times \mathcal{S}_1 \times \ldots \times \mathcal{S}_{n-1}$ is an $|\mathcal{S}_0| \times |\mathcal{S}_1| \times \ldots \times |\mathcal{S}_{n-1}|$ matrix. As we demonstrate later, 2-d matrices are often very useful to represent the problem constraints. We assume a 2-d $n \times m$ matrix has n columns and m rows.

3.2.1 Combinatorics

The **balanced incomplete block design (BIBD) problem** is a standard combinatorial problem from design theory [CD96] with applications in experimental design and cryptography (prob028 in CSPLib). BIBD generation is to find a set of $b > 0$ subsets of a set $\mathcal{V}$ of $v \geq 2$ elements such that:

- each subset consists of exactly k elements ($v > k > 0$);
- each element appears in exactly r subsets ($r > 0$);
- each pair of elements appear simultaneously in exactly λ subsets ($\lambda > 0$).

A BIBD instance is thus explained by its parameters $\langle v, b, r, k, \lambda \rangle$.

Matrix:

$$\begin{array}{ccccccccccc}
 & \multicolumn{10}{c}{\longrightarrow \mathcal{B} \longrightarrow} \\
X_{i,j} & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
\downarrow & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\mathcal{V} & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\downarrow & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
\qquad \langle v, b, r, k, \lambda \rangle \;=\; \langle 6, 10, 5, 3, 2 \rangle$$

Constraints:

$$\begin{array}{ll}
(1) & \forall j \in \mathcal{V}\,.\ \sum_{i \in \mathcal{B}} X_{i,j} = r \\
(2) & \forall i \in \mathcal{B}\,.\ \sum_{j \in \mathcal{V}} X_{i,j} = k \\
(3) & \forall j_1, j_2, j_1 < j_2 \in \mathcal{V}\,.\ \sum_{i \in \mathcal{B}} X_{i,j_1} * X_{i,j_2} = \lambda
\end{array}$$

Figure 3.1: The matrix model of the BIBD problem in [MT99].

The first CP approach to BIBD generation is due to Meseguer and Torras [MT99], where a BIBD instance is specified by a 2-d 0/1 matrix X of $\mathcal{B} \times \mathcal{V}$, where $\mathcal{B} = \{0, \ldots, b-1\}$. A variable $X_{i,j}$ in this matrix takes the value 1 iff the subset i contains the element j. The constraints therefore enforce exactly r 1s per row, k 1s per column, and a scalar product of λ between any pair of distinct rows (see Figure 3.1).

The **ternary Steiner problem (tSp)** is another combinatorial problem and originates from the computation of hypergraphs in combinatorial mathematics [LR80]. The tSp of order n is to find $b = n(n-1)/6$ subsets of $\mathcal{N} = \{1, \ldots, n\}$ such that:

- each subset is of cardinality 3;
- any two subsets have at most one element in common.

Gervet proposes in [Ger97] a model of this problem, where each subset is represented by a set variable. Whilst a cardinality constraint on each set variable restricts the size of each subset to 3, intersection constraints between every pair of set variables constrain the subsets to share at most one element. We adapt this model by replacing each set variable by a 1-d 0/1 matrix of $\mathcal{N}$, representing the characteristic function of $\mathcal{N}$. A variable in such a matrix is assigned 1 iff the corresponding element is in the subset. We thus have a 2-d 0/1 matrix X of $\mathcal{B} \times \mathcal{N}$, where $\mathcal{B} = \{0, \ldots, b-1\}$, with the constraints enforcing exactly 3 1s per column, and a scalar product of at most 1 between any pair of distinct columns (see Figure 3.2).

Even though this model is similar to that of BIBD generation, there are two main differences. First, the matrix of BIBD has sum constraints on its rows, but the matrix of tSp does not. Second, the scalar product constraint is posted between every pair of rows of the matrix of BIBD, but between every pair of columns of the matrix of tSp. Moreover, the scalar product is to be at most 1 in the tSp but exactly 1 in the BIBD model.

Matrix modelling provides an effective way of representing the BIBD problem and the tSp, as the problem constraints can be expressed in terms of sum and scalar product constraints. Many constraint toolkits provide these constraints together with a filtering algorithm which maintains BC during search.

Matrix:

$$\begin{array}{cccccccc}
 & & & \longrightarrow & \mathcal{B} & \longrightarrow & & \\
X_{i,j} & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\downarrow & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\mathcal{N} & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\downarrow & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{array} \qquad n \;=\; 7$$

Constraints:

$$\begin{array}{ll}
(1) & \forall i \in \mathcal{B}\,.\;\; \sum_{j \in \mathcal{N}} X_{i,j} = 3 \\
(2) & \forall i_1, i_2, i_1 < i_2 \in \mathcal{B}\,.\;\; \sum_{j \in \mathcal{N}} X_{i_1,j} * X_{i_2,j} \leq 1
\end{array}$$

Figure 3.2: A matrix model of the ternary Steiner problem.

Many other combinatorial problems can be effectively modelled using matrix models; e.g., quasigroup existence (prob003 in CSPLib), magic squares (prob019 in CSPLib), and projective planes (prob025 in CSPLib).

3.2.2 Design

The **steel mill slab design problem** is a difficult problem that reduces to variable-sized bin-packing with colour side-constraints (prob038 in CSPLib). Steel is produced by casting molten iron into slabs, of which a steel mill is capable of making a finite number of weights. A set $\mathcal{Z}$ of slab sizes is available. Given a set $\mathcal{O}rders$ of orders, the flexibility lies in the number and size of slabs chosen to fulfill the orders. Each order i in $\mathcal{O}rders$ is described by a tuple $\langle w_i, c_i \rangle$, where w_i is its weight and c_i is the colour corresponding to the route required through the steel mill. The problem is to decide how many slabs of which size to chose in order to pack the orders, such that:

- each order is assigned to exactly one slab;
- the total weight of orders assigned to a slab does not exceed the slab size;
- each slab contains at most p colours (p is usually 2);
- the total slab size is minimised.

A matrix model of this problem is given in Figure 3.3. We use potentially redundant variables to cope with the fact that the number of slabs in an optimal solution is unknown. If we assume that the greatest order weight does not exceed the maximum slab size, the worst case assigns each order to an individual slab. Hence, we assume we are given a set $\mathcal{S}labs$ of available slabs, where $|\mathcal{S}labs| = |\mathcal{O}rders|$.

We use a 1-d matrix S, indexed by $\mathcal{S}labs$, taking values from $\mathcal{Z}$, so as to decide what size each slab is. As some slabs may remain unused, 0 is added to $\mathcal{Z}$ to represent when

Matrices:

$$\begin{array}{rccccccccc} & \multicolumn{9}{c}{\longrightarrow \mathcal{S}labs \longrightarrow} \\ S_i & 4 & 3 & 3 & 3 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{rccccccccc} & \multicolumn{9}{c}{\longrightarrow \mathcal{O}rders \longrightarrow} \\ O_{i,j} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \downarrow & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathcal{S}labs & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \downarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & & & \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{rcl} \mathcal{Z} & = & \{0, 1, 3, 4\} \\ \mathcal{O}rders & = & \{\langle 2, 1\rangle, \langle 3, 2\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \\ & & \langle 1, 4\rangle, \langle 1, 4\rangle, \langle 2, 5\rangle, \langle 1, 5\rangle\} \end{array}$$

$$\begin{array}{rccccc} & \multicolumn{5}{c}{\rightarrow \mathcal{C}olours \rightarrow} \\ C_{i,j} & 0 & 0 & 0 & 1 & 1 \\ \downarrow & 0 & 0 & 1 & 1 & 0 \\ \mathcal{S}labs & 0 & 1 & 0 & 0 & 0 \\ \downarrow & 1 & 1 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 \\ & \vdots & & & & \\ & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{rcl} \mathcal{C}olours & = & \{1, 2, 3, 4, 5\} \\ p & = & 2 \end{array}$$

Constraints:

(1) $\forall j \in \mathcal{S}labs\, .\ \sum_{i \in \mathcal{O}rders} w_i * O_{i,j} \leq S_j$
(2) $\forall i \in \mathcal{O}rders\, .\ \sum_{j \in \mathcal{S}labs} O_{i,j} = 1$
(3) $\forall j \in \mathcal{S}labs\, .\ \sum_{i \in \mathcal{C}olours} C_{i,j} \leq p$
(4) $\forall i \in \mathcal{O}rders\, .\ \forall j \in \mathcal{S}labs\, .\ O_{i,j} = 1 \rightarrow C_{c_i,j} = 1$

Objective:

$$minimize \sum_{i \in \mathcal{S}labs} S_i$$

Figure 3.3: A matrix model of the steel mill slab design problem.

a slab is not used. In order to represent which orders are assigned to which slabs, a 2-d 0/1 matrix O of $\mathcal{O}rders \times \mathcal{S}labs$ is introduced. We have $O_{i,j} = 1$ iff order i is assigned to slab j. The first two constraints enforce that the capacity of each slab is not exceeded, and that each order is assigned to a slab and is not split between slabs. A second 2-d 0/1 matrix C of $\mathcal{C}olours \times \mathcal{S}labs$ is used to model the colour constraints, where $\mathcal{C}olours$ is the set of the colours of the orders. A variable $C_{i,j}$ in this matrix is assigned 1 iff slab j is assigned an order whose colour is i. The third constraint ensures that the number of colours in a slab does not exceed p. Finally, channelling constraints are used to connect this to the order matrix. The objective is then to minimise the total slab size.

Matrices:

$$\begin{array}{cccc} & & \longrightarrow \mathcal{T}emplates \longrightarrow & \\ Run_i & 51,000 & 107,000 & 250,000 \end{array}$$

$$\begin{array}{rccc} & & \longrightarrow \mathcal{T}emplates \longrightarrow & \\ T_{i,j} & 0 & 0 & 1 \\ & 5 & 0 & 0 \\ \downarrow & 3 & 1 & 0 \\ \mathcal{V}ariations & 0 & 0 & 2 \\ \downarrow & 0 & 0 & 2 \\ & 1 & 7 & 0 \\ & 0 & 1 & 4 \end{array}$$

$$\begin{array}{rcl} |\mathcal{T}emplates| & = & 3 \\ s & = & 9 \\ \mathcal{V}ariations & = & \{\langle 250,000\rangle, \langle 255,000\rangle, \\ & & \langle 260,000\rangle, \langle 500,000\rangle, \\ & & \langle 500,000\rangle, \langle 1,000,000\rangle, \\ & & \langle 1,100,000\rangle\} \end{array}$$

Constraints:

$$\begin{array}{ll} (1) & \forall i \in \mathcal{T}emplates\,.\ \sum_{j \in \mathcal{V}ariations} T_{i,j} = s \\ (2) & \forall j \in \mathcal{V}ariations\,.\ \sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} \geq d_j \end{array}$$

Objective:

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

Figure 3.4: The matrix model of the template design problem in [PS98].

Matrix modelling provides an effective way of representing the steel mill slab design problem. For instance, without C, we need to post large-arity constraints on O that can only be efficiently implemented by means of a complex daemon [FMW01]. In Figure 3.3, however, the problem constraints are expressed in terms of sum and scalar product constraints, which can be propagated effectively and efficiently in many constraint toolkits.

The **template design problem** is another design problem (prob002 in CSPLib). It arises from printing products of the same brand with several variations which have different display on them, but are identical in shape and size so they can be printed on the same sheet of board. For instance, two variations of a cat food cartoon may differ only in that one is printed "Chicken Flavour" whereas the other has "Rabbit Flavour" printed. Each sheet is printed from a template that has s slots. Each variation i in $\mathcal{V}ariations$ is described by a demand d_i, giving the minimum number of pressings required. The problem is to decide how many pressings of each template are needed, and how many copies of which variation to include on each template such that:

- the minimum number of pressings for each variation is met;
- every slot in each template is occupied by a variation;
- the total number of templates being produced is minimised.

One way of tackling this problem is to fix the number of templates and then to minimise the total number of pressings [PS98]. This can be modelled using a 1-d matrix *Run* giving

Matrices:

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

$$\begin{array}{rcccc} & \multicolumn{4}{c}{\rightarrow \mathcal{C}types \rightarrow} \\ C_{i,j} & 0 & 0 & 0 & 0 \\ \downarrow & 0 & 0 & 0 & 0 \\ \mathcal{R}acks & 0 & 1 & 2 & 0 \\ \downarrow & 0 & 3 & 0 & 1 \\ & 10 & 0 & 0 & 0 \end{array} \qquad \begin{array}{rcl} |\mathcal{R}acks| & = & 5 \\ \mathcal{R}ack\mathcal{M}odels & = & \{\langle 150, 8, 150\rangle, \langle 200, 16, 200\rangle\} \\ \mathcal{C}types & = & \{\langle 20, 10\rangle, \langle 40, 4\rangle, \langle 50, 2\rangle, \langle 75, 1\rangle\} \end{array}$$

Constraints:

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

Objective:

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

Figure 3.5: The matrix model of the rack configuration problem in [ILO02].

the number of pressings of each template, and a 2-d matrix T specifying how many copies of which variation are included on which template (see Figure 3.4). The matrix *Run* is indexed by the set $\mathcal{T}emplates$ of templates, taking values from $\{1, \ldots, maxQuantity\}$, where *maxQuantity* is the maximum among the number of pressings demanded for each variation. The matrix T is indexed by $\mathcal{T}emplates$ and $\mathcal{V}ariations$, taking values from $\{0, \ldots, s\}$ so that we might include 0 or more but maximum s of each variation on a template. The constraints enforce that every template has all its s slots occupied, and the total production of each variation is at least its demand. The objective is then to minimise the total number of pressings. This matrix model of the problem was previously proposed and showed effective in [PS98].

3.2.3 Configuration

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

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

Matrix:

$$\begin{array}{r|cccccccccccccccc}
 & \multicolumn{16}{c}{\longrightarrow \mathcal{C}ards \longrightarrow} \\
C_{i,j} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\downarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{R}acks & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
\downarrow & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Constraints:

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

Figure 3.6: The modifications to the model in Figure 3.5 to obtain a new matrix model of the rack configuration problem.

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

A matrix model of this problem is given in [ILO02] and shown in Figure 3.5. The main idea of this model is to assign a rack model to every available rack. Since some of the racks might not be needed in an optimal solution, a "dummy" rack model is introduced (i.e., a rack is assigned the dummy rack model when the rack is not needed). Furthermore, for every available rack, the number of cards — of a particular card type — plugged into the rack has to be determined. The assignment of rack models to racks is represented by a 1-d matrix R, indexed by $\mathcal{R}acks$, taking values from $\mathcal{R}ackModels$ which includes the dummy rack model. In order to represent the number of cards – of a particular card type – plugged into a particular rack, a 2-d matrix C of $\mathcal{C}types \times \mathcal{R}acks$ is introduced. A variable in this matrix takes values from $\{0, \ldots, maxConn\}$ where $maxConn$ is the maximum number of cards that can be plugged into any rack.

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

Another way of modelling this problem, as also mentioned in [ILO02], is to manipulate the cards rather than the number of cards of a given type plugged into a rack. In this case, we only modify the 2-d matrix C which is now indexed by the set $\mathcal{C}ards$ of cards and $\mathcal{R}acks$, taking values from $\{0, 1\}$. We have $C_{i,j} = 1$ iff card i is plugged into rack j. Figure 3.6 shows how we modify the initial model.

Note the similarity between the rack configuration and the steel mill slab design problems. In the former, we need to find a model for each available rack, and decide which

Matrix:

$$\begin{array}{rll} T_{i,j} & \multicolumn{2}{l}{\rightarrow \mathcal{G}roups \rightarrow} \\ \downarrow & \{1,2\} & \{3,4\} \\ \mathcal{W}eeks & \{1,3\} & \{2,4\} \end{array} \qquad \langle w,g,s\rangle \;=\; \langle 2,2,2\rangle$$

Constraints:

(1) $\forall j \in \mathcal{W}eeks\,.\ \forall i_1, i_2, i_1 < i_2 \in \mathcal{G}roups\,.\ T_{i_1,j} \cap T_{i_2,j} = \{\}$

(2) $\forall i \in \mathcal{G}roups\,.\ \forall j \in \mathcal{W}eeks\,.\ |T_{i,j}| = s$

(3) $\forall \langle i_1, j_1\rangle, \langle i_2, j_2\rangle, \langle i_1, j_1\rangle <_{lex} \langle i_2, j_2\rangle \in \mathcal{G}roups \times \mathcal{W}eeks\,.\ |T_{i_1,j_1} \cap T_{i_2,j_2}| \leq 1$

Figure 3.7: The matrix model of the social golfers problem by Novello.

cards are assigned to which racks; whereas in the latter, we need to find a size for each available slab, and decide which orders are assigned to which slabs. We introduce a dummy value in both problems as some racks or slabs might not be needed in an optimal solution. Hence, the matrix models of the rack configuration problem are very similar to that of the steel mill slab design problem. All problem constraints are expressed in terms of sum and scalar product constraints, which can be propagated effectively and efficiently in many constraint toolkits.

3.2.4 Scheduling

The **social golfers problem**, which is a variation of the Kirkman's schoolgirls problem [Bal38], has come from a question submitted on the newsgroups `comp.constraints` and `comp.op-research` (prob010 in CSPLib). The problem is to determine if it is possible for $g * s$ golfers to play in g groups, each of size s, in each of w weeks in such a way that:

- each golfer plays once a week;
- any two golfers play in the same group at most once.

An instance of this problem is thus explained by its parameters $\langle w, g, s\rangle$. The original question is to find the largest w such that 32 golfers can play in 8 groups of size 4.

A matrix model of this problem has been employed by Stefano Novello (available at `www.icparc.ic.ac.uk/eclipse/examples/`). The model uses a 2-d matrix T of $\mathcal{G}roups \times \mathcal{W}eeks$, where $\mathcal{G}roups$ is the set of g groups and $\mathcal{W}eeks$ is the set of w weeks. Each variable $T_{i,j}$ of the matrix is a set variable, which is a subset of the set $\mathcal{G}olfers$ of golfers, giving the set of golfers that play together as group i in week j (see Figure 3.7). The constraints enforce that the groups within a week are disjoint, every group is of size s, and that any two golfers appear together in the same group at most once.

This model has proven very effective as it helped find the best known solution to the original question, which is a 9 week schedule. Our model is a modification of this model in which each set variable is replaced by a 1-d 0/1 matrix of $\mathcal{G}olfers$, representing the characteristic function of $\mathcal{G}olfers$. A variable in such a matrix is assigned 1 iff the corresponding golfer is in the set. Hence, we have a 3-d 0/1 matrix T of $\mathcal{G}roups \times \mathcal{W}eeks \times \mathcal{G}olfers$, where $T_{i,j,k} = 1$ iff the kth player plays in group i of week j (see Figure 3.8).

Matrix:

$$\begin{array}{rccccc}
 & \multicolumn{2}{l}{\mathcal{G}olfers} & & & \\
 & & & & 0 & 1 \\
 & & & 0 & 1 & \\
 & \nearrow & 1 & 0 & & \\
T_{i,j,k} & 1 & 0 & & & \\
\downarrow & & & & 0 & 1 \\
\mathcal{W}eeks & & & 1 & 0 & \\
\downarrow & & 0 & 1 & & \\
 & 1 & 0 & & & \\
 & \multicolumn{5}{l}{\quad \rightarrow \mathcal{G}roups \rightarrow}
\end{array}
\qquad \langle w, g, s \rangle \;=\; \langle 2, 2, 2 \rangle$$

Constraints:

$$\begin{array}{ll}
(1) & \forall k \in \mathcal{G}olfers\,.\ \forall j \in \mathcal{W}eeks\,.\ \sum_{i \in \mathcal{G}roups} T_{i,j,k} = 1 \\
(2) & \forall j \in \mathcal{W}eeks\,.\ \forall i \in \mathcal{G}roups\,.\ \sum_{k \in \mathcal{G}olfers} T_{i,j,k} = s \\
(3) & \forall k_1, k_2, k_1 < k_2 \in \mathcal{G}olfers\,.\ \sum_{j \in \mathcal{W}eeks, i \in \mathcal{G}roups} T_{i,j,k_1} * T_{i,j,k_2} \leq 1
\end{array}$$

Figure 3.8: A matrix model of the social golfers problem using a 3-d matrix.

The constraints ensure that every golfer plays once in every week, every group of every week contains s players, and that every pair of players meet at most once. Replacing each set variable by its characteristic function means that the problem constraints can now be expressed in terms of sum and scalar product constraints, which can be propagated effectively and efficiently in many constraint toolkits.

Another interesting scheduling problem is the **sport scheduling problem** which is about scheduling games between n teams over $n-1$ weeks (prob026 in CSPLib). Each week is divided into $n/2$ periods, and each period is divided into two slots. The team in the first slot plays at home, while the team in the second slot plays away. The goal is to find a schedule such that:

- every team plays exactly once a week;
- every team plays against every other team;
- every team plays at most twice in the same period over the tournament.

Van Hentenryck *et al.* propose a model for this problem, where they introduce a "dummy" final week to make the problem more uniform [vHMPR99]. The model consists of two matrices: a 3-d matrix T of $\mathcal{E}weeks \times \mathcal{P}eriods \times \mathcal{S}lots$ and a 2-d matrix G of $\mathcal{W}eeks \times \mathcal{P}eriods$, where $\mathcal{E}weeks$ is the set of n extended weeks, $\mathcal{W}eeks$ is the set of $n-1$ weeks, $\mathcal{P}eriods$ is the set of $n/2$ periods, and $\mathcal{S}lots$ is the set of 2 slots. In T, weeks are extended to include the dummy week, and each variable takes a value from $\{1, \ldots, n\}$ expressing that a team plays in a particular week in a particular period, in the home or away slot. For the sake of simplicity, we will treat this matrix as 2-d where the rows represent the periods and the columns represent the extended weeks, and each entry of the matrix is a pair of variables. The variables of G takes values from $\{1, \ldots, n^2\}$, and

Matrices:

$$\begin{array}{r|ccccccc}
 & \mathcal{S}lots & & & & & & \\
 & \nearrow & 3 & 2 & 6 & 6 & 5 & 5 \\
T_{i,j,k} & 1 & 1 & 2 & 3 & 4 & 4 & \\
\downarrow & & 4 & 5 & 4 & 5 & 6 & 6 \\
\mathcal{P}eriods & 2 & 3 & 3 & 2 & 1 & 1 & \\
\downarrow & & 6 & 6 & 5 & 4 & 3 & 3 \\
 & 5 & 4 & 1 & 1 & 2 & 2 & \\
 & \multicolumn{7}{l}{\rightarrow \mathcal{E}weeks \rightarrow}
\end{array}
\qquad n = 6$$

$$\begin{array}{r|ccccc}
 & \multicolumn{5}{l}{\rightarrow \mathcal{W}eeks \rightarrow} \\
G_{i,j} & 3 & 2 & 12 & 18 & 23 \\
\downarrow & 10 & 17 & 16 & 11 & 6 \\
\mathcal{P}eriods & 30 & 24 & 5 & 4 & 9
\end{array}$$

Constraints:

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

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

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

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

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

Figure 3.9: The matrix model of the sport scheduling problem in [vHMPR99].

each denotes a particular unique combination of home and away teams. More precisely, a game played between a home team h and an away team a is uniquely identified by $(h-1)*n+a$ (see Figure 3.9).

Since there are n teams playing in each extended week, an *all-different* constraint on the teams playing in every extended week enforces that every team plays exactly once a week. As the 2-d matrix G represents the games, an *all-different* constraint on G ensures that every team plays against every other team. With the extended weeks, the global cardinality constraints (*gcc*) ensure that every team plays exactly twice in every period of the tournament. The channelling constraints link the variables of T and G, and are enforced by restricting every game between two teams h and a to be $(h-1)*n+a$, where $h < a$. Finally, an ordering constraint on the slot variables of each period and week ensures correct channelling between the matrices.

This model employs two powerful global constraints which are propagated effectively and efficiently: *all-different*, which is available in many constraint toolkits, and *gcc*, which is available in, for instance, ILOG Solver 5.3 [ILO02], SICStus Prolog constraint solver 3.10.1 [SIC03], and FaCiLe constraint solver 1.0 [FaC01]. Even though the chanelling constraints are imposed via the set of allowed tuples, some constraint toolkits, such as ILOG Solver 5.3 [ILO02], can maintain GAC on constraints expressed in terms of valid combination of values. Van Hentenryck *et al.* demonstrate in [vHMPR99] the effectiveness of this matrix model of the sport scheduling problem.

3.2.5 Timetabling

The **progressive party problem** arises in the context of organising the social programme for a yachting rally (prob013 in CSPLib). In the original formulation of the progressive party problem, there are 42 boats in total. Certain boats are to be designated as hosts, and the crews of the remaining boats are in turn to visit the host boats for 6 half-hour periods. The crew of a host boat remains on board to act as hosts while the crew of a guest boat together visits several hosts. Every boat can only host a limited number of guests at a time (its capacity). A guest boat cannot not revisit a host and guest crews cannot meet more than once. The problem is to minimise the number of host boats. The data of the problem can be found in CSPLib.

Smith *et al.* in [SBHW96] first showed that ILP formulations of this problem resulted in very large models. Even if several strategies were tried, an optimal solution could not be found. Second, they recognised that the total capacity of the first 13 boats (arranged in descending order of spare capacity) is sufficient to accommodate all the crews, but the capacity of the first 12 is not. Also they found a feasible solution –using an ILP approach– with the first 14 hosts by relaxing some constraints, and manually modifying the infeasible assignments so as to satisfy the violated constraints. To determine whether 13 or 14 is the optimal solution, they reformulated the progressive party problem with the 13 specified host boats as a CSP: there is a set $\mathcal{H}osts$ of host boats (2-12, 14, 16: the first 13 boats arranged in descending order of spare capacity), and a set $\mathcal{G}uests$ of guest boats (the 29 remaining boats). Each host boat i is characterised by a tuple $\langle hc_i, c_i \rangle$, where and hc_i is its crew size and c_i is its capacity; and each guest boat is described by gc_i giving its crew size. The problem is to assign hosts to guests over 6 time periods, such that:

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

A matrix model of this problem, as proposed in [SBHW96], is as follows. A 2-d matrix H is used to represent the assignment of hosts to guests in time periods (see Figure 3.10). The matrix H is indexed by the set $\mathcal{P}eriods$ of time periods and $\mathcal{G}uests$, taking values from $\mathcal{H}osts$. The first constraint enforces that two guests can meet at most once by introducing a new set of 0/1 variables:

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

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

In this matrix model of the progressive party problem, the problem constraints are stated using sum, scalar product, and *all-different* constraints, which are propagated effectively and efficiently in many constraint toolkits. The effectiveness of this model is discussed in [SBHW96].

Matrices:

$$\begin{array}{rccc}
 & \multicolumn{3}{c}{\rightarrow \mathcal{P}eriods \rightarrow} \\
H_{i,j} & 0 & 1 & 2 \\
\downarrow & 2 & 1 & 0 \\
\mathcal{G}uests & 0 & 2 & 1 \\
\downarrow & 1 & 0 & 2
\end{array}$$

$$\begin{array}{rccccc}
 & \multicolumn{2}{c}{\mathcal{H}osts} & 0 & 0 & 1 \\
C_{i,j,k} & \nearrow & 0 & 1 & 0 & 0 \\
 & 1 & 0 & 0 & 0 & 0 \\
\downarrow & 0 & 0 & 1 & 1 & 1 \\
\mathcal{G}uests & 1 & 0 & 0 & 0 & \\
\downarrow & 0 & 1 & 0 & & \\
 & \multicolumn{3}{c}{\rightarrow \mathcal{P}eriods \rightarrow} & &
\end{array}$$

Constraints:

$$\begin{array}{ll}
(1) & \forall j_1, j_2, j_1 < j_2 \in \mathcal{G}uests\,.\ \sum_{i \in \mathcal{P}eriods}(H_{i,j_1} = H_{i,j_2}) \leq 1 \\
(2) & \forall j \in \mathcal{G}uests\,.\ \textit{all-different}(\langle H_{0,j}, H_{1,j}, \ldots, H_{p-1,j}\rangle) \\
(3) & \forall i \in \mathcal{P}eriods\,.\ \forall k \in \mathcal{H}osts\,.\ \sum_{j \in \mathcal{G}uests} gc_j * C_{i,j,k} \leq c_k - hc_k \\
(4) & \forall i \in \mathcal{P}eriods\,.\ \forall j \in \mathcal{G}uests\,.\ \sum_{k \in \mathcal{H}osts} C_{i,j,k} = 1 \\
(5) & \forall i \in \mathcal{P}eriods\,.\ \forall j \in \mathcal{G}uests\,.\ \forall k \in \mathcal{H}osts\,.\ H_{i,j} = k \leftrightarrow C_{i,j,k} = 1
\end{array}$$

Figure 3.10: The matrix model of the progressive party problem in [SBHW96].

3.2.6 Bioinformatics

The **word design problem** has its roots in bioinformatics and coding theory (prob033 in CSPLib). The problem is to find as large a set $\mathcal{W}$ of words (i.e. strings) of length 8 over the alphabet $\{A, C, T, G\}$ such that:

- each pair of distinct words in $\mathcal{W}$ differ in at least 4 positions;
- each pair of words x and y in $\mathcal{W}$ (where x and y may be identical) are such that x^R and y^C differ in at least 4 positions, where x^R is the reverse of x, and y^C is the Watson-Crick complement of y, i.e. the word where each A is replaced by a T and vice versa, and each C is replaced by a G and vice versa;
- the symbols A and T occur together 4 times in a word in $\mathcal{W}$;
- the symbols C and G occur together 4 times in a word in $\mathcal{W}$.

A model of this problem, proposed in [Hni02], is to maintain two 2-d matrices M and CM of $\mathcal{L} \times \mathcal{W}ords$, where $\mathcal{L} = \{0, \ldots, 7\}$ and $\mathcal{W}ords = \{0, \ldots, size - 1\}$ where $size = |\mathcal{W}|$. The symbols in $\{A, C, T, G\}$ are represented by the integers $\{0, 1, 2, 3\}$. In M, the rows correspond to the words in the set $\mathcal{W}$, and each variable $M_{i,j}$ takes a value from $\{0, 1, 2, 3\}$, giving the symbol of the word j at position i. In CM, the rows correspond to the complements of the words in $\mathcal{W}$, and each variable $CM_{i,j}$ takes a value from $\{0, 1, 2, 3\}$, giving the symbol of the word j^C at position i (see Figure 3.11).

Matrices:

$$\begin{array}{r|cccccccc}
 & \multicolumn{8}{c}{\rightarrow \mathcal{L} \rightarrow} \\
M_{i,j} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\downarrow & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\mathcal{W}ords & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \\
\downarrow & 3 & 3 & 1 & 0 & 1 & 0 & 0 & 0 \\
 & 3 & 3 & 1 & 1 & 0 & 0 & 2 & 2
\end{array}$$

$$size \;=\; 5$$

$$\begin{array}{r|cccccccc}
 & \multicolumn{8}{c}{\rightarrow \mathcal{L} \rightarrow} \\
CM_{i,j} & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\
\downarrow & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 \\
\mathcal{W}ords & 3 & 3 & 3 & 3 & 0 & 0 & 0 & 0 \\
\downarrow & 1 & 1 & 3 & 2 & 3 & 2 & 2 & 2 \\
 & 1 & 1 & 3 & 3 & 2 & 2 & 0 & 0
\end{array}$$

Constraints:

$$\begin{array}{ll}
(1) & \forall j_1, j_2, j_1 < j_2 \in \mathcal{W}ords \,.\; \sum_{i \in \mathcal{L}} (M_{i,j_1} = M_{i,j_2}) \leq 4 \\
(2) & \forall j_1, j_2, j_1 < j_2 \in \mathcal{W}ords \,.\; \sum_{i \in \mathcal{L}} (M_{size-1-i,j_1} = CM_{i,j_2}) \leq 4 \\
(3) & \forall j \in \mathcal{W}ords \,.\; \langle M_{0,j}, M_{1,j}, \ldots, M_{size-1,j} \rangle \in \mathcal{A} \\
(4) & \forall i \in \mathcal{L} \,.\; \forall j \in \mathcal{W}ords \,.\; \langle M_{i,j}, CM_{i,j} \rangle \in \{\langle 0,2 \rangle, \langle 1,3 \rangle, \langle 2,0 \rangle, \langle 3,1 \rangle\}
\end{array}$$

Figure 3.11: The matrix model of the word design problem by Hnich.

Whilst the first constraint enforces each pair of distinct words to differ in at least 4 positions, the second constraint ensures that the reverse of a word and the complement of any other word differ in at least 4 positions. These constraints are enforced by introducing two new sets of 0/1 variables:

$$\begin{array}{l}
\forall i \in \mathcal{L} \,.\; \forall j_1, j_2, j_1 < j_2 \in \mathcal{W}ords \,.\; X_{i,j_1,j_2} = 1 \leftrightarrow M_{i,j_1} = M_{i,j_2} \\
\forall i \in \mathcal{L} \,.\; \forall j_1, j_2, j_1 < j_2 \in \mathcal{W}ords \,.\; Y_{i,j_1,j_2} = 1 \leftrightarrow M_{size-1-i,j_1} = CM_{i,j_2}
\end{array}$$

A sum constraint on each set of new variables ensures that at most 4 variables are set to 1. The rest of the constraints are posted via the third constraint which restricts the words to be a member of the set $\mathcal{A}$. This set is a set of $size$-tuples, such that each tuple a satisfies the following properties:

- a^R and a^C differ in at least 4 positions;
- 0 and 2 occur together 4 times in a;
- 1 and 3 occur together 4 times in a.

The channelling constraints are then used to link the variables of M and CM.

In this model of the word design problem, some of the problem constraints are enforced using sum constraints which are effectively and efficiently propagated in many constraint

Matrix:

$$\begin{array}{r ccccc} & & \rightarrow & \mathcal{B} & \rightarrow & \\ X_{i,j} & 0 & 0 & 0 & 0 & 0 \\ \downarrow & 0 & 0 & 0 & 1 & 1 \\ \mathcal{C}odes & 0 & 0 & 1 & 0 & 1 \\ \downarrow & 0 & 0 & 1 & 1 & 0 \\ & 0 & 1 & 0 & 0 & 1 \end{array} \qquad \langle n, b, d\rangle \quad = \quad \langle 5, 5, 2\rangle$$

Constraints:

(1) $\forall j_1, j_2, j_1 < j_2 \in \mathcal{C}odes\,.\ \ \sum_{i\in\mathcal{B}}(X_{i,j_1} = X_{i,j_2}) \leq d$

Figure 3.12: The matrix model of generating Hamming codes.

toolkits. Even though the rest of the problem constraints and the chanelling constraints are imposed via the set of allowed tuples, some constraint toolkits, such as ILOG Solver 5.3 [ILO02], can maintain GAC on constraints expressed in terms of valid combination of values. This model of the word design problem is promising for finding large sets of words [Hni02].

3.2.7 Code Generation

The problem of **generating Hamming codes** is to find b-bit words to code n symbols where the Hamming distance between every two symbol codes is at least d [Oz03]. The Hamming distance between two codes is the number of positions where they differ. An instance of this problem is thus described by its parameters $\langle n, b, d\rangle$.

A model of this problem, as given in [Oz03], is to represent each code by a set variable. For a code w, we have $s_w \subseteq \mathcal{B}$ where $\mathcal{B} = \{1, \ldots, b\}$ and $e \in s_w$ iff the bit position e is set. The Hamming distance between two codes a and b represented as sets s_a and s_b is computed as $b - |s_a \cap s_b| - |\mathcal{B}\backslash(s_a \cup s_b)|$. We adapt this model by replacing each set variable by a 1-d 0/1 matrix of $\mathcal{B}$, representing the characteristic function of $\mathcal{B}$. A variable in such a matrix is assigned 1 iff the corresponding element is in the subset. We thus have a 2-d 0/1 matrix X of $\mathcal{B} \times \mathcal{C}odes$, where $\mathcal{C}odes = \{0, \ldots, n-1\}$. The Hamming distance between every pair of codes is ensured by introducing new 0/1 variables:

$$\forall i \in \mathcal{B}\,.\ \ \forall j_1, j_2, j_1 < j_2 \in \mathcal{C}odes\,.\ \ Y_{i,j_1,j_2} = 1 \leftrightarrow X_{i,j_1} = X_{i,j_2}$$

A sum constraint on these variables ensures that at most d variables are set to 1. Replacing each set variable by its characteristic function means that the problem constraints can now be expressed in terms of sum constraints, which can be propagated effectively and efficiently in many constraint toolkits.

3.3 Symmetry

As shown in Section 3.2, many problems from a wide range of application domains can be effectively formulated using matrix models. A common pattern in matrix models is row and column symmetry, as well as value symmetry. In this section, we identify these patterns by examining the matrix models presented in Section 3.2.

3.3.1 Row and Column Symmetry

2-d matrices are often very useful to effectively represent some or all the problem constraints. We can witness this in the matrix models of the problems described in the previous section: all models employ one or more 2-d matrices. A *matrix symmetry* of a 2-d matrix of variables is a permutation of its variables which preserves satisfiability.

Definition 9 *Given an $n \times m$ matrix X of variables each with a domain $\mathcal{D}$, and a set $\mathcal{C}$ of constraints defined on the variables of X, a matrix symmetry is a bijective function:*

$$f : \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\} \rightarrow \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\}$$

such that for any (partial) assignment h to the variables of X:

$$h : \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\} \rightarrow \mathcal{D}$$

h satisfies the constraints in $\mathcal{C}$ iff $h \circ f$ does.

A *row symmetry* of a 2-d matrix of variables is a matrix symmetry between the variables of two of its rows.

Definition 10 *Given an $n \times m$ matrix X of variables, a row symmetry is a matrix symmetry:*

$$f : \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\} \rightarrow \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\}$$

such that for some $0 \leq j_1, j_2 < m$, $j_1 \neq j_2$, and for all $0 \leq i < n$:

$$f(X_{i,j_1}) = X_{i,j_2} \ \wedge \ f(X_{i,j_2}) = X_{i,j_1}$$

Hence, two rows are *indistinguishable* if the roles of their variables can pairwise be interchanged due to a row symmetry. A 2-d matrix of variables *has row symmetry* if all its rows are indistinguishable. If, however, only a strict subset(s) of the rows are indistinguishable, then the matrix *has partial row symmetry.*

Similar definitions hold for the columns. A *column symmetry* of a 2-d matrix of variables is a matrix symmetry between the variables of two of its columns.

Definition 11 *Given an $n \times m$ matrix X of variables, a column symmetry is a matrix symmetry:*

$$f : \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\} \rightarrow \{X_{i,j} \mid \ 0 \leq i < n \ \wedge \ 0 \leq j < m\}$$

such that for some $0 \leq i_1, i_2 < n$, $i_1 \neq i_2$, and for all $0 \leq j < m$:

$$f(X_{i_1,j}) = X_{i_2,j} \ \wedge \ f(X_{i_2,j}) = X_{i_1,j}$$

Hence, two columns are *indistinguishable* if the roles of their variables can pairwise be interchanged due to a column symmetry. A 2-d matrix of variables *has column symmetry* if all its columns are indistinguishable. If, however, only a strict subset(s) of the columns are indistinguishable, then the matrix *has partial column symmetry.*

A 2-d matrix of variables *has row and column symmetry* iff the matrix has row symmetry, as well as column symmetry. A matrix has row symmetry and/or column symmetry when, for example, its rows and/or columns represent indistinguishable objects. In such a case, in any (partial) assignment to the variables, the rows and/or columns can be swapped without affecting whether or not the (partial) assignment satisfies the constraints. In the following, we give examples of (partial) row and/or (partial) column symmetry by examining the matrix models presented in Section 3.2.

Matrix model of the BIBD problem In Figure 3.1, the rows of the 2-d matrix X represent the elements, and the columns represent the subsets. Since the elements and the subsets are indistinguishable, the matrix X has both row and column symmetry.

Matrix model the tSp problem The rows of the 2-d matrix X in Figure 3.2 represent the elements of the set $\mathcal{N}$, and the columns represent the subsets. Similar to the BIBD problem, the elements and the subsets are indistinguishable. The matrix X has therefore both row and column symmetry.

Matrix model of the steel mill slab design problem In Figure 3.3, the rows of the 2-d matrix O represent the slabs, whereas the columns represent the orders. The matrix O has partial row symmetry because the slabs of the same size are indistinguishable. The matrix O has also partial column symmetry because the orders of the same size and colour are indistinguishable.

Matrix model of the template design problem The rows of the 2-d matrix T in Figure 3.4 correspond to the variations, and the columns correspond to the templates. The matrix T has partial row symmetry because the variations with equal demands are indistinguishable. The matrix T has also partial column symmetry because the templates with equal number of pressings are indistinguishable.

Matrix model of the rack configuration problem We have presented two matrix models for this problem. In Figures 3.5 and 3.6, the rows of the 2-d matrix C correspond to the racks. Since the racks of the same model are indistinguishable, C has partial row symmetry in both models. Whilst the columns of C correspond to the different card types in Figure 3.5, they give the individual cards in Figure 3.6. Different card types are distinguishable. Therefore, the matrix C in Figure 3.5 does not have column symmetry. On the other hand, the cards of the same type are indistinguishable. The matrix C in 3.6 therefore has also partial column symmetry.

Matrix model of the social golfers problem In this problem, the weeks are indistinguishable, and so are the groups and the golfers. In Figure 3.7, the 2-d matrix T has row and column symmetry, as the rows represent the weeks and the columns represent the groups. Similarly, the matrix T in Figure 3.8 has symmetry along each of the three dimensions.

Matrix model of the sport scheduling problem We treat the 3-d matrix T as 2-d in Figure 3.9, where the rows represent the periods and columns represent the extended weeks, and each entry of the matrix is a pair of variables. The extended weeks over which the tournament is held, as well the periods are indistinguishable. The matrix T thus has row and column symmetry.

Matrix model of the progressive party problem In Figure 3.10, the rows of the 2-d matrix H correspond to the guest boats, whereas the columns correspond to the time periods. The matrix H has partial row symmetry, as the guest boats with equal crew size are indistinguishable. The matrix H has also column symmetry, as the time periods are indistinguishable.

Matrix model of the word design problem Whilst the rows of the 2-d matrix M in Figure 3.11 represent the words in the set W, the columns represent the position of the symbols. This matrix has row symmetry since a set of words is modelled as a list of words, and the order of the words in the set are not important. However, M does not have column symmetry, because the positions are distinguishable due to the constraint which enforces two words x and y to be such that x^R and y^C differ in at least 4 positions.

Matrix model of the problem of generating Hamming codes In this problem, the codes are indistinguishable, so are the positions of the bits. In Figure 3.12, the 2-d matrix X has row and column symmetry, as the rows describe the codes and the columns give the positions.

Given an $n \times m$ matrix with row and column symmetry, there are $n!m!$ permutations of the rows and columns. Hence, the number of symmetries grows super-exponentially as the matrix size gets larger. This dramatically increases the size of the search space, as during search there will be many states (e.g., partial assignments, solutions, failures) that are symmetric but essentially equivalent to the already visited states. Consequently, this class of symmetries is very important. In many cases, large problems are intractable unless these symmetries are significantly reduced. We therefore need to develop special techniques to deal with a super-exponential number of symmetries effectively and efficiently.

3.3.2 Value Symmetry

Value symmetry arises when the values taken by some variables can be permuted without affecting satisfiability.

Definition 12 *Given a set $\mathcal{X}$ of variables each associated with a domain $\mathcal{D}$, and a set $\mathcal{C}$ of constraints defined on the variables in $\mathcal{X}$, value symmetry is a bijective function:*

$$f : \mathcal{D} \rightarrow \mathcal{D}$$

such that for any (partial) assignment h to the variables in $\mathcal{X}$:

$$h : \mathcal{X} \rightarrow \mathcal{D}$$

h satisfies the constraints in $\mathcal{C}$ iff $f \circ h$ does.

Hence, two values in a domain are *indistinguishable* if their roles can be interchanged due to a value symmetry. A matrix of variables *has value symmetry* if all the values in the domain of the variables are indistinguishable. If, however, only a strict subset(s) of the values are indistinguishable, then the matrix *has partial value symmetry.* In such cases, in any (partial) assignment to the variables, some or all values can be swapped without affecting whether or not the (partial) assignment satisfies the constraints. In the following, we give examples of (partial) value symmetry by examining 4 of the matrix models presented in Section 3.2.

Matrix model of the social golfers problem In this problem, the golfers are indistinguishable. In Figure 3.7, the 2-d matrix T has value symmetry, as the set variables of the matrix are constrained to be the subset of $\mathcal{G}olfers$ which is a set of indistinguishable elements.

Matrix model of the sport scheduling problem In this problem, the teams are indistinguishable. The 2-d matrix T in Figure 3.9 has value symmetry, as each variable of T takes a value from $\{1, \ldots, n\}$ which is the set of indistinguishable teams.

Matrix model of the progressive party problem A variable of the 2-d matrix H in Figure 3.10 takes a value from $\mathcal{H}osts$. As the host boats of the same capacity are indistinguishable, the matrix H has partial value symmetry.

Matrix model of the word design problem In this problem, the symbols 0 and 2, and the symbols 1 and 3 are indistinguishable. The 2-d matrix M in Figure 3.11 has partial value symmetry, as each variable of M takes a value from $\{0, \ldots, 3\}$ which is the set of partially indistinguishable symbols.

Unlike row and column symmetry, value symmetry is not confined to matrix models. Any CSP may have value symmetry, whether it is formulated using a matrix model or not. On the other hand, as we will show later, value symmetry in a matrix can be transformed to, for instance, row symmetry. This is another advantage of developing effective techniques for dealing with row and column symmetries. Such techniques can deal both with problems that naturally have row and column symmetry, as well as with problems that have value symmetry. We will elaborate on this topic in Chapter 4.

3.4 Summary

In this chapter, we have recognised the central role played in many constraints programs by matrix models, and identified two patterns that commonly arise in matrix models: row and column symmetry, and value symmetry.

Symmetry in constraint programs causes wasted search effort, as it generates symmetrically equivalent states in the search space. Some symmetry breaking methods have been devised in the past years, such as SES [BW99][BW02], SBDS [GS00], and SBDD [FM01][FSS01], all of which can directly be used to break all row and column symmetries. SES and SBDS treat each symmetry individually, which is impractical when the number of symmetries is large. Even though it is correct to use a subset of the symmetries to break some but not necessarily all symmetries using these methods, it requires writing problem-specific search methods. The dominance checks of SBDD can be very expensive in the presence of many symmetries. Moreover, the dominance checks are problem dependent. Row and column symmetry is therefore an important class of symmetries, as it is very common and often generates too many symmetries to be handled effectively, efficiently, and easily by methods like SES, SBDS, and SBDD. In many cases, large problems are intractable unless row and column symmetries are significantly reduced. We therefore need to develop special techniques to deal with this pattern effectively. As we will show later, value symmetry in a matrix can easily be transformed to, for instance, row symmetry. This demonstrates another benefit of tackling row and column symmetry.

By extending constraint toolkits to support these patterns, we believe that we help many constraint programmers towards formulating effective models. This not only strengthens the power of CP, but also promotes the reach of CP to a wider user base.

Chapter 4

Breaking Row and Column Symmetry

4.1 Introduction

In Chapter 3, we identified an important class of symmetries in CP, which arises in matrices of decision variables where rows and columns represent indistinguishable objects and are therefore symmetric. We can permute any two rows as well as two columns of a 2-d matrix with row and column symmetry without affecting the satisfiability of any (partial) assignments. An $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries, which increase super-exponentially. Consequently, it can be very costly to visit all the symmetric branches in a tree search. In this chapter, we help tackle this problem.

Ideally, we would like to cut off all the symmetric parts of the search tree. Symmetry breaking methods such as SES [BW99][BW02], SBDS [GS00], and SBDD [FM01][FSS01] achieve this goal by not exploring the parts of the search tree which are symmetric to the those that are already considered. Even though these methods are applicable to any class of symmetries, they have difficulty in dealing with the super-exponential number of symmetries in a problem with row and column symmetry. SES and SBDS treat each symmetry individually, and the dominance checks of SBDD are costly. Moreover, using SBDD and a subset of the symmetries in the presence of many symmetries in SES and SBDS require problem-specific adjustments. In order to break row and column symmetries in a simpler and efficient way, we investigate ordering constraints that can be posted on the rows and columns of matrices. Ordering constraints can be used to break the row and column symmetries of *any* problem modelled using a matrix. Indeed, they can be used for matrices of arbitrary dimension.

This chapter is organised as follows. In Section 4.2, we propose some ordering constraints for breaking row and column symmetries of a 2-d matrix. In particular, we show that we can enforce lexicographic ordering or multiset ordering constraints on the symmetric rows and columns with the guarantee of finding at least one solution from each equivalence class of solutions. These constraints can also be combined to obtain new symmetry breaking constraints. The effectiveness of the ordering constraints in breaking symmetry, from a theoretical point of view, is discussed in Section 4.3. We extend our results in Section 4.4 to deal with matrices of arbitrary dimension, as well as with matrices that contain partial symmetry or value symmetry. In Section 4.5, we identify special cases where all row and column symmetries can easily be broken. Finally, we compare with related work in Section 4.6, and summarise our contributions in Section 4.7.

4.2 Ordering Constraints

One of the easiest and most efficient ways of symmetry breaking is adding extra constraints to the model of our problem [Pug93]. These constraints impose an ordering on the symmetric objects. As a consequence, among the set of symmetric assignments, only those that satisfy the ordering constraints are chosen for consideration during the process of search.

As the symmetry breaking constraints remove some assignments from a symmetry class, it is important that at least one element remains in the class. Otherwise, we may lose solutions by symmetry breaking. We say that an ordering constraint is a *consistent* symmetry breaking constraint iff for every assignment to the variables that does not satisfy the ordering constraint, there is a symmetric assignment that does.

In this section, we investigate what consistent ordering constraints we can post on the rows and columns of a 2-d matrix of decision variables to break row and column symmetries.

4.2.1 Lexicographic Ordering Constraint

One popular approach to sorting vectors of values is to order them lexicographically [CGLR96]. 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 13 *Strict lexicographic ordering* $\vec{x} <_{lex} \vec{y}$ *between two vectors of integers* $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ *and* $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1} \rangle$ *holds iff* $\exists k \; 0 \leq k < n$ *such that* $x_i = y_i$ *for all* $0 \leq i < k$ *and* $x_k < y_k$.

That is, there exists an index k above which the subvectors are equal, and x_k is less than y_k. We can weaken the ordering to include equality.

Definition 14 *Two vectors of integers* $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ *and* $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1} \rangle$ *are lexicographically ordered* $\vec{x} \leq_{lex} \vec{y}$ *iff* $\vec{x} <_{lex} \vec{y}$ *or* $\vec{x} = \vec{y}$.

As the rows of a matrix are vectors, lexicographic ordering is a very natural ordering of the rows. We say that the rows of a matrix of values are lexicographically ordered if each row is lexicographically less than or equal to the next row (if any), where the first row is the top-most row of the matrix. More formally:

Definition 15 *The rows of an* $n \times m$ *matrix* x *of values are lexicographically ordered iff for all* $0 \leq i < m-1$ *we have* $\langle x_{0,i}, x_{1,i}, \ldots, x_{n-1,i} \rangle \leq_{lex} \langle x_{0,i+1}, x_{1,i+1}, \ldots, x_{n-1,i+1} \rangle$.

Similarly, as the columns of a matrix are vectors, we say that the columns of a matrix of values are lexicographically ordered if each column is lexicographically less than or equal to the next column (if any), where the first column is the left-most column of the matrix.

Definition 16 *The columns of an* $n \times m$ *matrix* x *of values are lexicographically ordered iff for all* $0 \leq i < n-1$ *we have* $\langle x_{i,0}, x_{i,1}, \ldots, x_{i,m-1} \rangle \leq_{lex} \langle x_{i+1,0}, x_{i+1,1}, \ldots, x_{i+1,m-1} \rangle$.

Given two vectors of variables $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a lexicographic ordering constraint as $\vec{X} \leq_{lex} \vec{Y}$ and a strict lexicographic ordering constraint as $\vec{X} <_{lex} \vec{Y}$. These constraints ensure that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ are ordered according to Definitions 14 and 13, respectively. We can utilise such constraints to break row or column symmetry. Lexicographic ordering constraints in one of the symmetric dimensions of a matrix are consistent symmetry breaking constraints.

Theorem 1 *Given a 2-d matrix with row symmetry, each symmetry class has exactly one element where the rows are lexicographically ordered. Similarly, given a 2-d matrix with column symmetry, each symmetry class has exactly one element where the columns are lexicographically ordered.*

Proof: We consider only the row symmetry, as the proof for the column symmetry is entirely analogous. Suppose there is an assignment a of values to the variables of the matrix. Since lexicographic ordering is total, the rows of a can be ordered lexicographically, and swapping any two rows of a breaks the lexicographic ordering unless the rows are identical. Swapping two identical rows does not give us a new assignment. Hence, there is exactly one element in the symmetry class of a where the rows are lexicographically ordered. QED.

That is, we can break the row symmetry of an $n \times m$ matrix by enforcing that the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{m-1}$ are lexicographically ordered:

$$\vec{R}_0 \leq_{lex} \vec{R}_1 \ldots \leq_{lex} \vec{R}_{m-1}$$

If no pair of rows can be equal to each other, we can then enforce strict lexicographic ordering constraints:

$$\vec{R}_0 <_{lex} \vec{R}_1 \ldots <_{lex} \vec{R}_{m-1}$$

Similarly, we can break the column symmetry of an $n \times m$ matrix by insisting that the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{n-1}$ are lexicographically ordered:

$$\vec{C}_0 \leq_{lex} \vec{C}_1 \ldots \leq_{lex} \vec{C}_{n-1}$$

and strengthen the model by imposing instead strict lexicographic ordering constraints if no pair of columns can be equal to each other:

$$\vec{C}_0 <_{lex} \vec{C}_1 \ldots <_{lex} \vec{C}_{n-1}$$

Whilst it is easy to break symmetry in one dimension of the matrix, it is not obvious how to break symmetry in both dimensions, as the rows and columns intersect. After constraining the rows to be lexicographically ordered, we distinguish the columns and thus the columns are not symmetric anymore. Nevertheless, given a matrix with row and column symmetry, each symmetry class has at least one element where both the rows and columns are lexicographically ordered. This result is due to a theorem by Lubiw, in which a "doubly lexical ordering" of a matrix is defined as "an ordering of the rows and the columns so that the rows – as vectors – are lexicographically increasing and the columns – as vectors – are lexicographically increasing".

Theorem 2 ([Lub85][Lub87]) *Every real-valued matrix has a doubly lexical ordering.*

Consequently, we can break the row and column symmetries by insisting that the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{m-1}$ and the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{n-1}$ are both lexicographically ordered:

$$\vec{R}_0 \leq_{lex} \vec{R}_1 \ldots \leq_{lex} \vec{R}_{m-1} \ \wedge \ \vec{C}_0 \leq_{lex} \vec{C}_1 \ldots \leq_{lex} \vec{C}_{n-1}$$

We refer to such symmetry breaking constraints as double-lex constraints. Depending on whether any pair of rows and/or columns can be equal or not, we can strengthen the symmetry breaking constraints by enforcing instead strict lexicographic ordering constraints on the rows and/or columns.

Note that Lubiw's theorem applies to any matrix where the values are drawn from a totally ordered set and thus is sufficient to prove that the double-lex constraints are consistent. We can reach this conclusion via another approach. One way to break variable symmetry is to force an assignment – within its equivalence class of assignments – to be lexicographically the smallest [CGLR96]. To do this, we first select a configuration of the variables where we place the variables in a vector in the order we prefer. Second, we insist that an assignment of variables in the selected configuration is lexicographically less than or equal to any assignment in the configuration obtained by permuting the symmetric variables of the selected configuration. For instance, assume we have three symmetric variables X_0, X_1, and X_2 subject to permutation. To break the symmetry, we can enforce the following constraints:

$$\begin{array}{lcl}
\langle X_0, X_1, X_2\rangle & \leq_{lex} & \langle X_0, X_2, X_1\rangle \\
\langle X_0, X_1, X_2\rangle & \leq_{lex} & \langle X_1, X_0, X_2\rangle \\
\langle X_0, X_1, X_2\rangle & \leq_{lex} & \langle X_1, X_2, X_0\rangle \\
\langle X_0, X_1, X_2\rangle & \leq_{lex} & \langle X_2, X_0, X_1\rangle \\
\langle X_0, X_1, X_2\rangle & \leq_{lex} & \langle X_2, X_1, X_0\rangle
\end{array}$$

where $\langle X_0, X_1, X_2\rangle$ is the chosen configuration. Indeed, these constraints simplify to $X_0 \leq X_1 \leq X_2$.

This technique can easily be applied to break row and column symmetries. Assume we have the following matrix of variables where the rows and columns are symmetric:

$$\begin{pmatrix}
X_1, & X_2, & X_3, & \ldots, & X_n \\
Y_1, & Y_2, & Y_3, & \ldots, & Y_n \\
\vdots & \vdots & \vdots & \ldots & \vdots \\
Z_1, & Z_2, & Z_3, & \ldots, & Z_n
\end{pmatrix} \tag{4.1}$$

We first need to choose a configuration of the variables. By, for instance, appending the rows of the matrix, we get:

$$\langle X_1,\ X_2,\ X_3,\ \ldots,\ X_n,\ Y_1,\ Y_2,\ Y_3,\ \ldots,\ Y_n,\ \ldots,\ Z_1,\ Z_2,\ Z_3,\ \ldots,\ Z_n\rangle \tag{4.2}$$

Second, we need to enforce that 4.2 is lexicographically less than or equal to any configuration obtained by swapping the blocks of variables in 4.2 as a result of permuting the rows and/or columns of 4.1. Consider the first column of 4.1. Permuting it with the second column swaps the following variables in 4.2:

$$\langle \overset{\curvearrowright}{X_1},\ \overset{\curvearrowleft}{X_2},\ X_3,\ \ldots,\ X_n,\ \overset{\curvearrowright}{Y_1},\ \overset{\curvearrowleft}{Y_2},\ Y_3,\ \ldots,\ Y_n,\ \ldots,\ \overset{\curvearrowright}{Z_1},\ \overset{\curvearrowleft}{Z_2},\ Z_3,\ \ldots,\ Z_n\rangle$$

This gives us the configuration:

$$\langle X_2,\ X_1,\ X_3,\ \ldots,\ X_n,\ Y_2,\ Y_1,\ Y_3,\ \ldots,\ Y_n,\ \ldots,\ Z_2,\ Z_1,\ Z_3,\ \ldots,\ Z_n\rangle \tag{4.3}$$

We then insist that 4.2 is lexicographically less than or equal to 4.3. Ignoring the variables whose position are the same in 4.2 and 4.3, we have the following constraint:

$$\begin{gathered}\langle X_1,\ X_2,\ \cancel{X_3},\ \cancel{\ldots},\ \cancel{X_n},\ Y_1,\ Y_2,\ \cancel{Y_3},\ \cancel{\ldots},\ \cancel{Y_n},\ \cancel{\ldots},\ Z_1,\ Z_2,\ \cancel{Z_3},\ \cancel{\ldots},\ \cancel{Z_n\rangle}\ \leq_{lex}\\ \langle X_2,\ X_1,\ \cancel{X_3},\ \cancel{\ldots},\ \cancel{X_n},\ Y_2,\ Y_1,\ \cancel{Y_3},\ \cancel{\ldots},\ \cancel{Y_n},\ \cancel{\ldots},\ Z_2,\ Z_1,\ \cancel{Z_3},\ \cancel{\ldots},\ \cancel{Z_n\rangle}\end{gathered}$$

which simplifies to:

$$\langle X_1, Y_1, \ldots, Z_1\rangle \leq_{lex} \langle X_2, Y_2, \ldots, Z_2\rangle$$

By permuting the first and the third column of 4.1, we get:

$$\begin{gathered}\langle X_1,\ \cancel{X_2},\ X_3,\ \cancel{\ldots},\ \cancel{X_n},\ Y_1,\ \cancel{Y_2},\ Y_3,\ \cancel{\ldots},\ \cancel{Y_n},\ \cancel{\ldots},\ Z_1,\ \cancel{Z_2},\ Z_3,\ \cancel{\ldots},\ \cancel{Z_n\rangle}\ \leq_{lex}\\ \langle X_3,\ \cancel{X_2},\ X_1,\ \cancel{\ldots},\ \cancel{X_n},\ Y_3,\ \cancel{Y_2},\ Y_1,\ \cancel{\ldots},\ \cancel{Y_n},\ \cancel{\ldots},\ Z_3,\ \cancel{Z_2},\ Z_1,\ \cancel{\ldots},\ \cancel{Z_n\rangle}\end{gathered}$$

which simplifies to:

$$\langle X_1, Y_1, \ldots, Z_1\rangle \leq_{lex} \langle X_3, Y_3, \ldots, Z_3\rangle$$

By permuting the first column with every other column, we obtain the following constraints:

$$\begin{gathered}\langle X_1, Y_1, \ldots, Z_1\rangle \leq_{lex} \langle X_2, Y_2, \ldots, Z_2\rangle\\ \langle X_1, Y_1, \ldots, Z_1\rangle \leq_{lex} \langle X_3, Y_3, \ldots, Z_3\rangle\\ \vdots\\ \langle X_1, Y_1, \ldots, Z_1\rangle \leq_{lex} \langle X_n, Y_n, \ldots, Z_n\rangle\end{gathered}$$

Now consider the second column of 4.1. Permuting it with the third column swaps the following variables in 4.2:

$$\langle X_1,\ \overset{\curvearrowright}{X_2},\ \overset{\curvearrowleft}{X_3},\ \ldots,\ X_n,\ Y_1,\ \overset{\curvearrowright}{Y_2},\ \overset{\curvearrowleft}{Y_3},\ \ldots,\ Y_n,\ \ldots,\ Z_1,\ \overset{\curvearrowright}{Z_2},\ \overset{\curvearrowleft}{Z_3},\ \ldots,\ Z_n\rangle$$

giving the configuration:

$$\langle X_1,\ X_3,\ X_2,\ \ldots,\ X_n,\ Y_1,\ Y_3,\ Y_2,\ \ldots,\ Y_n,\ \ldots,\ Z_1,\ Z_3,\ Z_2,\ \ldots,\ Z_n\rangle \tag{4.4}$$

Now we enforce that 4.2 is lexicographically less than or equal to 4.4. We obtain the following constraint, after eliminating those variables whose position are the same in 4.2 and 4.4:

$$\begin{gathered}\cancel{\langle X_1},\ X_2,\ X_3,\ \cancel{\ldots},\ \cancel{X_n},\ \cancel{Y_1},\ Y_2,\ Y_3,\ \cancel{\ldots},\ \cancel{Y_n},\ \cancel{\ldots},\ \cancel{Z_1},\ Z_2,\ Z_3,\ \cancel{\ldots},\ \cancel{Z_n\rangle}\\ \cancel{\langle X_1},\ X_3,\ X_2,\ \cancel{\ldots},\ \cancel{X_n},\ \cancel{Y_1},\ Y_3,\ Y_2,\ \cancel{\ldots},\ \cancel{Y_n},\ \cancel{\ldots},\ \cancel{Z_1},\ Z_3,\ Z_2,\ \cancel{\ldots},\ \cancel{Z_n\rangle}\end{gathered}$$

This constraint simplifies to:

$$\langle X_2, Y_2, \ldots, Z_2\rangle \leq_{lex} \langle X_3, Y_3, \ldots, Z_3\rangle$$

By permuting the second column with every other column (except the first one), we obtain the following constraints:

$$\begin{gathered}\langle X_2, Y_2, \ldots, Z_2\rangle \leq_{lex} \langle X_3, Y_3, \ldots, Z_3\rangle\\ \langle X_2, Y_2, \ldots, Z_2\rangle \leq_{lex} \langle X_4, Y_4, \ldots, Z_4\rangle\\ \vdots\\ \langle X_2, Y_2, \ldots, Z_2\rangle \leq_{lex} \langle X_n, Y_n, \ldots, Z_n\rangle\end{gathered}$$

By considering every $\binom{n}{2}$ permutation, we obtain lexicographic ordering constraints between every pair of columns:

$$\begin{array}{rcl}
\langle X_1, Y_1, \ldots, Z_1\rangle & \leq_{lex} & \langle X_2, Y_2, \ldots, Z_2\rangle \\
\langle X_1, Y_1, \ldots, Z_1\rangle & \leq_{lex} & \langle X_3, Y_3, \ldots, Z_3\rangle \\
 & \vdots & \\
\langle X_1, Y_1, \ldots, Z_1\rangle & \leq_{lex} & \langle X_n, Y_n, \ldots, Z_n\rangle \\
\langle X_2, Y_2, \ldots, Z_2\rangle & \leq_{lex} & \langle X_3, Y_3, \ldots, Z_3\rangle \\
\langle X_2, Y_2, \ldots, Z_2\rangle & \leq_{lex} & \langle X_4, Y_4, \ldots, Z_4\rangle \\
 & \vdots & \\
\langle X_2, Y_2, \ldots, Z_2\rangle & \leq_{lex} & \langle X_n, Y_n, \ldots, Z_n\rangle \\
 & \vdots & \\
\langle X_{n-2}, Y_{n-2}, \ldots, Z_{n-2}\rangle & \leq_{lex} & \langle X_{n-1}, Y_{n-1}, \ldots, Z_{n-1}\rangle \\
\langle X_{n-2}, Y_{n-2}, \ldots, Z_{n-2}\rangle & \leq_{lex} & \langle X_n, Y_n, \ldots, Z_n\rangle \\
\langle X_{n-1}, Y_{n-1}, \ldots, Z_{n-1}\rangle & \leq_{lex} & \langle X_n, Y_n, \ldots, Z_n\rangle
\end{array}$$

Repeating the same procedure on the rows, we have lexicographic ordering constraints also between every pair of rows:

$$\begin{array}{rcl}
\langle X_1, X_2, X_3 \ldots, X_n\rangle & \leq_{lex} & \langle Y_1, Y_2, Y_3, \ldots, Y_n\rangle \\
 & \vdots & \\
\langle X_1, X_2, X_3, \ldots, X_n\rangle & \leq_{lex} & \langle Z_1, Z_2, Z_3, \ldots, Z_n\rangle \\
 & \vdots & \\
\langle Y_1, Y_2, Y_3, \ldots, Y_n\rangle & \leq_{lex} & \langle Z_1, Z_2, Z_3, \ldots, Z_n\rangle \\
 & \vdots &
\end{array}$$

Hence, the double-lex constraints are generated by applying the technique proposed in [CGLR96] to break the row and column symmetries of a matrix.

So far we have defined (strict) lexicographic ordering in non-decreasing order, and identified consistent symmetry breaking ordering constraints based on this definition. We can obtain totally dual ordering constraints by defining new lexicographic orderings. If two vectors are ordered lexicographically in non-increasing order, then we call the (strict) lexicographic ordering a (strict) anti-lexicographic ordering.

Definition 17 *Strict anti-lexicographic ordering $\vec{x} >_{lex} \vec{y}$ between two vectors of integers $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1}\rangle$ and $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1}\rangle$ holds iff $\exists k$ $0 \leq k < n$ such that $x_i = y_i$ for all $0 \leq i < k$ and $x_k > y_k$.*

Definition 18 *Two vectors of integers $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1}\rangle$ and $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1}\rangle$ are anti-lexicographically ordered $\vec{x} \geq_{lex} \vec{y}$ iff $\vec{x} >_{lex} \vec{y}$ or $\vec{x} = \vec{y}$.*

Given two vectors of variables $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write an anti-lexicographic ordering constraint as $\vec{X} \geq_{lex} \vec{Y}$, and a strict anti-lexicographic ordering constraint as $\vec{X} >_{lex} \vec{Y}$. These constraints ensure that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ are ordered according to Definitions 18 and 17, respectively. Given a matrix with row (resp. column) symmetry, the rows (resp. columns) of an assignment to the variables of the matrix can as well be made anti-lexicographically ordered. Hence, we can as well post anti-lexicographic ordering constrains on the rows (resp. columns) to break the symmetry, and strengthen the constraints by enforcing instead strict anti-lexicographic ordering constraints if no pair of rows (resp. columns) can be equal.

Given a matrix with row and column symmetry, it is not hard to prove that anti-lexicographic ordering constraints both on the rows and columns, called double anti-lex constraints, are consistent symmetry breaking constraints. By forcing an assignment – within its symmetry class – to be lexicographically the largest, as opposed to be the smallest, double anti-lex constraints are generated by the method given in [CGLR96] to break the row and column symmetries of a matrix. Therefore, we can as well post double anti-lex constraints to break row and column symmetries. Depending on whether any pair of rows and/or columns can be equal or not, we can strengthen the symmetry breaking constraints by enforcing instead strict anti-lexicographic ordering constraints on the rows and/or columns.

Regrettably, lexicographic ordering constraints in one dimension and anti-lexicographic ordering constraints in the other dimension of a matrix with row and column symmetry are not consistent symmetry breaking constraints. This is because a symmetry class may not have an element where the rows are lexicographically ordered but the columns are anti-lexicographically ordered, or vice-versa. As an example, consider a 2 × 2 matrix X with row and column symmetry, and suppose $x_0 = \begin{pmatrix} 0 & 3 \\ 2 & 1 \end{pmatrix}$ is an assignment to the variables of X. By permuting the rows and columns of x_0, we get $x_1 = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$, $x_2 = \begin{pmatrix} 3 & 0 \\ 1 & 2 \end{pmatrix}$, and $x_3 = \begin{pmatrix} 1 & 2 \\ 3 & 0 \end{pmatrix}$ as symmetric assignments. Whilst x_0 and x_3 have both the rows and columns lexicographically ordered, x_1 and x_2 have both the rows and columns anti-lexicographically ordered. There is, however, no symmetric assignment in which the rows are lexicographically ordered and the columns are anti-lexicographically ordered, or vice versa.

4.2.2 Multiset Ordering Constraint

Lexicographic ordering is very focused on positions and ignores values beneath the position where the vectors differ. Multiset ordering, on the other hand, ignores positions but focuses on values. Multiset ordering is a way of sorting unordered lists of values, and has been successfully used for proving program termination [DM79] and termination of term rewriting systems [KB70]. Multiset ordering is a total ordering on multisets and is defined as follows.

Definition 19 *Strict multiset ordering* $\mathbf{x} <_m \mathbf{y}$ *between two multisets of integers* $\mathbf{x}$ *and* $\mathbf{y}$ *holds iff:*

$$\begin{array}{r} \mathbf{x} = \{\!\{\}\!\} \ \wedge \ \mathbf{y} \neq \{\!\{\}\!\} \quad \vee \\ max(\mathbf{x}) < max(\mathbf{y}) \quad \vee \\ max(\mathbf{x}) = max(\mathbf{y}) \ \wedge \ \mathbf{x} - \{\!\{max(\mathbf{x})\}\!\} <_m \mathbf{y} - \{\!\{max(\mathbf{y})\}\!\} \end{array}$$

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

Definition 20 *Two multisets of integers* $\mathbf{x}$ *and* $\mathbf{y}$ *are multiset ordered* $\mathbf{x} \leq_m \mathbf{y}$ *iff* $\mathbf{x} <_m \mathbf{y}$ *or* $\mathbf{x} = \mathbf{y}$.

Even though the rows and columns of a matrix are vectors, it may be useful to ignore the positions but rather concentrate on the values taken by the variables when ordering the rows and/or columns. We can do this by treating each row and/or column of a matrix as a multiset. We say that the rows of a matrix of values are multiset ordered if each row, as a multiset, is no greater than the rows below it. More formally:

Definition 21 *The rows of an $n \times m$ matrix x of values are multiset ordered iff for all $0 \leq i < m-1$ we have $\{\!\{\langle x_{0,i}, x_{1,i}, \ldots, x_{n-1,i}\rangle\}\!\} \leq_m \{\!\{\langle x_{0,i+1}, x_{1,i+1}, \ldots, x_{n-1,i+1}\rangle\}\!\}$.*

Similarly, we say that the columns of a matrix of values are multiset ordered if each column, as a multiset, is no greater than the columns to the right of it.

Definition 22 *The columns of an $n \times m$ matrix x of values are multiset ordered iff for all $0 \leq i < n-1$ we have $\{\!\{\langle x_{i,0}, x_{i,1}, \ldots, x_{i,m-1}\rangle\}\!\} \leq_m \{\!\{\langle x_{i+1,0}, x_{i+1,1}, \ldots, x_{i+1,m-1}\rangle\}\!\}$.*

Whilst multiset ordering is a total ordering on multisets, it is not a total ordering on vectors. In fact, it is a preordering as it is not antisymmetric. Consider $\vec{x} = \langle 1,2,3\rangle$ and $\vec{y} = \langle 3,2,1\rangle$. We have $\{\!\{\vec{x}\}\!\} \leq_m \{\!\{\vec{y}\}\!\}$ and $\{\!\{\vec{y}\}\!\} \leq_m \{\!\{\vec{x}\}\!\}$, but not $\vec{x} = \vec{y}$. Hence, multiset ordering on the vectors is a different ordering than multiset ordering on multisets.

Given two vectors of variables $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a multiset ordering constraint as $\vec{X} \leq_m \vec{Y}$ and a strict multiset ordering constraint as $\vec{X} <_m \vec{Y}$. These constraints ensure that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$, when viewed as multisets, are multiset ordered according to Definitions 20 and 19, respectively. We can break row or column symmetry by imposing multiset ordering constraints on the rows or columns. Such constraints are consistent symmetry breaking constraints.

Theorem 3 *Given a 2-d matrix with row symmetry, each symmetry class has at least one element where the rows are multiset ordered. Similarly, given a 2-d matrix with column symmetry, each symmetry class has at least one element where the columns are multiset ordered.*

Proof: We consider only the row symmetry, as the proof for the column symmetry is entirely analogous. Suppose there is an assignment a of values to the variables of the matrix. Any two rows can be multiset ordered. Since multiset ordering is transitive, the rows of a can be multiset ordered. Hence, there is at least one element in the symmetry class of a where the rows are multiset ordered. QED.

As multiset ordering is not a total ordering on vectors, each symmetry class may have more than one element where the rows (resp. columns) are multiset ordered.

Due to Theorem 3, we can break the row symmetry of an $n \times m$ matrix by enforcing that the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{m-1}$ are multiset ordered:

$$\vec{R}_0 \leq_m \vec{R}_1 \ldots \leq_m \vec{R}_{m-1}$$

If no pair of rows, when viewed as multisets, can be equal to each other, we can then enforce strict multiset ordering constraints:

$$\vec{R}_0 <_m \vec{R}_1 \ldots <_m \vec{R}_{m-1}$$

Similarly, we can break the column symmetry of an $n \times m$ matrix by insisting that the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{n-1}$ are multiset ordered:

$$\vec{C}_0 \leq_m \vec{C}_1 \ldots \leq_m \vec{C}_{n-1}$$

and strengthen the model by imposing instead strict multiset ordering constraints if no pair of columns, when viewed as multisets, can be equal to each other:

$$\vec{C}_0 <_m \vec{C}_1 \ldots <_m \vec{C}_{n-1}$$

By defining multiset orderings in non-increasing order, we can obtain the dual ordering constraints. If two multisets are ordered in non-increasing order then we call the (strict) multiset ordering a (strict) anti-multiset ordering.

Definition 23 *Strict anti-multiset ordering* $\mathbf{x} >_m \mathbf{y}$ *between two multisets of integers* $\mathbf{x} = \{\!\{x_0, x_1, \ldots, x_{n-1}\}\!\}$ *and* $\mathbf{y} = \{\!\{y_0, y_1, \ldots, y_{n-1}\}\!\}$ *holds iff:*

$$\begin{aligned} \mathbf{x} \neq \{\!\{\}\!\} \ \wedge \ \mathbf{y} = \{\!\{\}\!\} \quad & \vee \\ max(\mathbf{x}) > max(\mathbf{y}) \quad & \vee \\ max(\mathbf{x}) = max(\mathbf{y}) \ \wedge \ \mathbf{x} - \{\!\{max(\mathbf{x})\}\!\} >_m \mathbf{y} - \{\!\{max(\mathbf{y})\}\!\} & \end{aligned}$$

Definition 24 *Two multisets of integers* $\mathbf{x} = \{\!\{x_0, x_1, \ldots, x_{n-1}\}\!\}$ *and* $\mathbf{y} = \{\!\{y_0, y_1, \ldots, y_{n-1}\}\!\}$ *are anti-multiset ordered* $\mathbf{x} \geq_m \mathbf{y}$ *iff* $\mathbf{x} >_m \mathbf{y}$ *or* $\mathbf{x} = \mathbf{y}$.

Given two vectors of variables $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write an anti-multiset ordering constraint as $\vec{X} \geq_m \vec{Y}$, and a strict anti-multiset ordering constraint as $\vec{X} >_m \vec{Y}$. These constraints ensure that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$, when viewed as multisets, are ordered according to Definitions 24 and 23, respectively. Given a matrix with row (resp. column) symmetry, the rows (resp. columns) of an assignment to the variables of the matrix can as well be made anti-multiset ordered. Hence, we can as well post anti-multiset ordering constrains on the rows (resp. columns) to break the symmetry, and strengthen the constraints by enforcing instead strict anti-multiset ordering constrains if no pair of rows (resp. columns), as multisets, can be equal.

How can we utilise multiset ordering constraints for breaking row and column symmetries? One of the nice features of using multiset ordering for breaking symmetry is that by constraining the rows of a matrix to be (anti-)multiset ordered, we do not distinguish the columns. We can still freely permute the columns, as (anti-)multiset ordering the rows ignores positions and is invariant to column permutation. It follows that we can consistently post multiset or anti-multiset ordering constraints on the columns together with (anti-)multiset ordering constraints on the rows. Given the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{m-1}$ and the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{n-1}$, there are thus four ways to break row and column symmetries using multiset orderings. We insist that:

1. the rows and columns are both multiset ordered:
$$\vec{R}_0 \leq_m \vec{R}_1 \ldots \leq_m \vec{R}_{m-1} \ \wedge \ \vec{C}_0 \leq_m \vec{C}_1 \ldots \leq_m \vec{C}_{n-1}$$
2. the rows and columns are both anti-multiset ordered:
$$\vec{R}_0 \geq_m \vec{R}_1 \ldots \geq_m \vec{R}_{m-1} \ \wedge \ \vec{C}_0 \geq_m \vec{C}_1 \ldots \geq_m \vec{C}_{n-1}$$
3. the rows are multiset ordered but the columns are anti-multiset ordered:
$$\vec{R}_0 \leq_m \vec{R}_1 \ldots \leq_m \vec{R}_{m-1} \ \wedge \ \vec{C}_0 \geq_m \vec{C}_1 \ldots \geq_m \vec{C}_{n-1}$$
4. the rows are anti-multiset ordered but the columns are multiset-ordered:
$$\vec{R}_0 \geq_m \vec{R}_1 \ldots \geq_m \vec{R}_{m-1} \ \wedge \ \vec{C}_0 \leq_m \vec{C}_1 \ldots \leq_m \vec{C}_{n-1}$$

If no pair of rows and/or columns, when viewed as multisets, can be equal, we can then instead impose strict (anti-)multiset ordering constraints on the rows and/or columns.

4.2.3 Lexicographic or Multiset Ordering Constraints?

We have shown that we can make use of lexicographic ordering or multiset ordering constraints for breaking row and/or column symmetries. This naturally raises the following questions. Does any of the orderings dominate the other? When we have row (resp. column) symmetry, is it better to post lexicographic ordering or multiset ordering constraints on the rows (resp. columns)? Similarly, when we have row and column symmetry, is it better to post lexicographic ordering or multiset ordering constraints on the rows and columns? Is posting multiset ordering constraints in both dimensions preferable to posting multiset ordering constraints in one dimension but anti-multiset ordering constraints in the other? Before answering these questions, we note that, for ease of presentation, we write $\preceq R$ for posting the ordering constraint $\preceq$ on the rows. Similarly, posting the ordering constraint $\preceq$ on the columns is specified as $\preceq C$, and on the rows and columns as $\preceq RC$.

Assume we have a 2×2 matrix X with row symmetry, and suppose $x_0 = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}$ is an assignment to the variables of X. The matrix $x_1 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$ is the only symmetric assignment to x_0, which is obtained by permuting the rows of x_0. The assignments x_0 and x_1 together form a symmetry class S. Even though the rows of x_0 are lexicographically ordered, they are not multiset ordered. The situation is the other way around for x_1. The rows of x_1 are multiset ordered but not lexicographically ordered. This shows that lexicographic ordering and multiset ordering are incomparable. As a result, $\leq_m R$ eliminates x_0 from S but $\leq_{lex} R$ does not, and $\leq_{lex} R$ removes x_1 from S while $\leq_m$ does not. This shows that the elements removed by $\leq_{lex} R$ from a symmetry class are not necessarily the same elements eliminated by $\leq_m R$, and therefore $\leq_{lex} R$ and $\leq_m R$ are incomparable. A similar argument holds between $\geq_{lex} R$ and $\geq_m R$, as well as between $\leq_{lex} C$ and $\leq_m C$, and between $\geq_{lex} C$ and $\geq_m C$, when we want to break column symmetry.

Since $\leq_{lex} R$ and $\leq_m R$ are incomparable, as well as $\geq_{lex} R$ and $\geq_m R$, $\leq_{lex} C$ and $\leq_m C$, and $\geq_{lex} C$ and $\geq_m C$, it is not hard to show that $\leq_{lex} RC$, $\leq_m RC$, $\leq_m R \geq_m C$, $\geq_m R \leq_m C$, and $\geq_m RC$ are all incomparable. Assume we have a 3×3 matrix X with row and column symmetry, and suppose $x_0 = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 0 & 3 \\ 1 & 1 & 1 \end{pmatrix}$ is an assignment to the variables of X. In the symmetry class S of x_0, only x_0, $x_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 3 \end{pmatrix}$, $x_2 = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{pmatrix}$, $x_3 = \begin{pmatrix} 0 & 0 & 3 \\ 0 & 0 & 2 \\ 1 & 1 & 1 \end{pmatrix}$, and $x_4 = \begin{pmatrix} 3 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ have both the rows and columns ordered. In particular, only x_0 has both the rows and columns lexicographically ordered, only x_1 has both the rows and columns multiset ordered, only x_2 has its rows multiset ordered but the columns anti-multiset ordered, only x_3 has its rows anti-multiset ordered but the columns multiset ordered, and only x_4 has both the rows and columns anti-multiset ordered. Whilst $\leq_{lex} RC$ leaves only x_0 in S, $\leq_m RC$ leaves only x_1, $\leq_m R \geq_m C$ leaves only x_2, $\geq_m R \leq_m C$ leaves only x_3, and $\geq_m RC$ only x_4. Since $\leq_{lex} RC$, $\leq_m RC$, $\leq_m R \geq_m C$, $\geq_m R \leq_m C$, and $\geq_m RC$ do not necessarily remove the same elements from a symmetry class, they are incomparable. A similar argument holds between $\geq_{lex} RC$, $\geq_m RC$, $\geq_m R \leq_m C$, $\leq_m R \geq_m C$, and $\leq_m RC$.

4.2.4 Combining Lexicographic and Multiset Ordering Constraints

Lexicographic ordering and multiset ordering are two different ways of ordering vectors. Whilst the former looks at positions, the latter ignores positions but looks at values in vectors. It is therefore natural that lexicographic ordering constraints and multiset ordering constraints on the rows (resp. columns) are incomparable symmetry breaking constraints. An alternative way to deal with row and column symmetries is to combine lexicographic ordering and multiset ordering constraints, and post this combination of constraints on the rows and columns. In this way, we benefit from each of the orderings, and we conjecture that such a combination of constraints can in practice be superior to both constraints.

As lexicographic ordering and multiset ordering are incomparable, we need to be careful when we combine the two different ways of symmetry breaking. For instance, given a matrix with row symmetry, we cannot simultaneously enforce lexicographic ordering and multiset ordering constraints on the rows. This is because a symmetry class may not have an element where the rows are both lexicographically and multiset ordered. As an example, suppose we have a 2 × 2 matrix X with row symmetry, and $x_0 = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}$ is an assignment to the variables of X. The matrix $x_1 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$ is the only symmetric assignment to x_0, which is obtained by permuting the rows of x_0. Whilst x_0 has its rows lexicographically ordered but not multiset ordered, x_1 has its rows multiset ordered but not lexicographically ordered. There is, however, no symmetric assignment to x_0 in which the rows are both lexicographically and multiset ordered.

Even though we cannot enforce lexicographic and multiset ordering constraints in the same dimension of a matrix, it is consistent to impose one ordering in one dimension and the other ordering in the other dimension. As explained in Section 4.2.2, by constraining the rows of a matrix to be (anti-)multiset ordered, we do not distinguish the columns. We can still freely permute the columns, as (anti-)multiset ordering the rows is invariant to column permutation. This shows that we can consistently post lexicographic or anti-lexicographic ordering constraints on the columns together with (anti-)multiset ordering constraints on the rows. By a similar argument, we can consistently post lexicographic or anti-lexicographic ordering constraints on the rows together with (anti-)multiset ordering constraints on the columns. Given the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{m-1}$ and the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{n-1}$, there are thus eight ways to break row and column symmetries by combining lexicographic ordering constraints in one dimension together with multiset ordering constraints in the other.

If no pair of rows, when viewed as vectors or multisets, can be equal, we can then replace (anti-)lexicographic ordering or (anti-)multiset ordering constraints on the rows by strict (anti-)lexicographic ordering constraints or strict (anti-)multiset ordering constraints, respectively. Similarly, if no pair of columns, when viewed as vectors or multisets, can be equal, we can then instead impose strict (anti-)lexicographic ordering or strict (anti-)multiset ordering constraints on the columns, respectively.

Since lexicographic ordering and multiset ordering are incomparable, imposing one ordering in one dimension and the other ordering in the other dimension of a matrix is incomparable to imposing the same ordering on both dimensions of the matrix. Assume we have a 3 × 3 matrix X with row and column symmetry, and suppose $x_0 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 2 & 3 \\ 1 & 0 & 1 \end{pmatrix}$ is an assignment to the variables of X. By swapping the last two columns, the last two rows, and the last two columns and rows of x_0, we get:

$$x_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 3 & 2 \\ 1 & 1 & 0 \end{pmatrix} \; x_2 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 3 \end{pmatrix} \; x_3 = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 3 & 2 \end{pmatrix}$$

respectively. The assignment x_0 has its rows lexicographically ordered and the columns multiset ordered, but neither the rows are multiset ordered nor the columns are lexicographically ordered. The assignment x_1 has both the rows and columns lexicographically ordered, but neither the rows nor the columns are multiset ordered. The assignment x_2 has both the rows and columns multiset ordered, but neither the rows nor the columns are lexicographically ordered. Finally, x_3 has its rows multiset ordered and the columns lexicographically ordered, but neither the rows are lexicographically ordered nor the columns are multiset ordered.

4.3 Symmetry Breaking with Ordering Constraints: A Theoretical Perspective

Imposing ordering constraints to a model changes the structure of the search tree. Among the set of symmetric assignments, only those that satisfy the ordering constraints are chosen for consideration during search. Hence, enforcing an ordering on the symmetric objects is a very easy way to break symmetry. However, enforcing ordering constraints does not necessarily break all symmetries, as this all depends on whether the ordering imposed is total or not. If the ordering is total then there is exactly one element in an equivalence class of assignments satisfying the ordering constraints. In this way, only one element is chosen from each symmetry class, and all its symmetric assignments are eliminated.

Let us consider only the row (resp. column) symmetry of a matrix. As lexicographic ordering is total, there is exactly one element in each equivalence class of assignments where the rows (resp. columns) are lexicographically ordered (Theorem 1). As a result, enforcing the rows (resp. columns) to be lexicographically ordered breaks all symmetries. Indeed, imposing any total ordering on the rows (resp. columns) breaks all symmetries. On the other hand, multiset ordering is not a total ordering on vectors. Two non-identical rows (resp. columns) may be identical when viewed as multisets. If swapping two non-identical rows (resp. columns) does not change the satisfiability of an assignment when the rows (resp. columns) are treated as multisets, then there may be more than one element in its symmetry class where the rows (resp. columns) are multiset ordered. Hence, enforcing the rows (resp. columns) to be multiset ordered may not break all symmetries. As an example, consider a 2 × 2 matrix X with row symmetry, and suppose $x_0 = \begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}$ is an assignment to the variables of X. By permuting the rows of x_0, we obtain only $x_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$. Both x_0 and x_1 have their rows multiset ordered. Despite imposing an ordering which is not total, the multiset ordering constraint is still an interesting ordering constraint, as lexicographic ordering constraints and multiset ordering constraints on the rows (resp. columns) are two incomparable symmetry breaking constraints, as discussed in Section 4.2.3.

Let us now consider the row and column symmetry of a matrix. Since imposing multiset ordering on the rows (resp. columns) may not break all row (resp. column) symmetries, we expect that enforcing both the rows and columns to be multiset ordered may not break all row and column symmetries either. Indeed, the example given above supports this argument. Assume X has row and column symmetry. By permuting the rows and columns of x_0, we again obtain only $x_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$. Both x_0 and x_1 have their rows and columns multiset ordered. Even though lexicographic ordering is total and imposing this ordering in one dimension of the matrix breaks all the symmetries of that dimension, enforcing both the rows and columns to be lexicographically ordered (double-lex constraints) may leave some symmetry. This is because double-lex constraints are a strict subset of the constraints generated by applying the technique of [CGLR96] which break all symmetries. The situation is not different when we insist multiset ordering in one dimension and lexicographic ordering in the other dimension of the matrix. For instance, consider a 3×3 matrix X with row and column symmetry, and suppose $x_0 = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \\ 1 & 2 & 3 \end{pmatrix}$ is an assignment to X. By permuting the first two rows and the first two columns of x_0, we get $x_1 = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \\ 2 & 1 & 3 \end{pmatrix}$. Both x_0 and x_1 have their columns lexicographically ordered, and rows both lexicographically and multiset ordered.

By imposing ordering constraints some symmetries may remain, but then how much symmetry is broken? Due to the repeating values in an assignment to the variables, the size of a symmetry class is not necessarily equal to the number of symmetries. Also, not every symmetry class has the same number of elements satisfying the ordering constraints. Suppose X is a 3×3 matrix with row and column symmetry, and $x_0 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ and $x_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$ are two different assignments to X. There are $3!3! = 36$ symmetries. Since all the rows, and also two columns of x_0 are the same, the size of the symmetry class of x_0 is $36/2!3! = 3$. On the other hand, the size of the symmetry class of x_1 is $36/2!2! = 9$ as only two rows, and also two columns are the same. Whilst there is only one element in the symmetry class of x_0 in which the rows and columns are lexicographically ordered, there are two such elements in the symmetry class of x_1. It is therefore not obvious how to judge the effectiveness of the ordering constraints in breaking symmetry from a theoretical point of view.

Posting ordering constraints does not necessarily reduce the size of the search tree, even if the ordering imposed is total. Since we search for a solution by a sequence of partial assignments and constraint propagation, a solution satisfying the ordering constraints might be obtained without having to enforce the ordering constraints, thanks to the decisions we make at every node of the search tree. For instance, by enforcing the rows of a matrix with row symmetry to be lexicographically ordered, we break all symmetries. Now, consider a 2×2 matrix of 0/1 variables, $X_{i,j}$, with the constraints $\sum_{i \in \{0,1\}} X_{i,j} = 1$ for all $0 \leq j < 2$. That is, every row of the matrix is constrained to have a single 1. Suppose that we look for a solution by labelling the variables in the order $X_{0,0}, X_{0,1}, X_{1,0}, X_{1,1}$ and exploring the domains in ascending order, and that we maintain generalised arc-consistency after every labelling. After the first branching (i.e., $X_{0,0} \leftarrow 0$), we get $\begin{pmatrix} 0 & 1 \\ 0..1 & 0..1 \end{pmatrix}$, as the sum constraint in the first row is propagated. We obtain a solution $\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$ after the second branching (i.e., $X_{0,1} \leftarrow 0$) which triggers the sum constraint in the second row. Since the rows of the matrix are symmetric, we can post $\langle X_{0,0}, X_{1,0} \rangle \leq_{lex} \langle X_{0,1}, X_{1,1} \rangle$. Every value in the partial assignment, which is obtained at the first node, is a consistent value for the lexicographic ordering constraint. Therefore, no extra value is pruned, and the solution, in which the rows are lexicographically ordered, is found at the second node. Since the labelling heuristic has been pushing the search towards a solution satisfying the ordering constraint, the search tree has remained unchanged.

In the previous example, we have seen that a solution satisfying the ordering constraints may easily be found even if the ordering constraints are not posted. Hence, imposing the ordering constraints may not reduce the size of the search tree when searching for one solution, or when searching for the first solution if all solutions need to be found. A similar phenomenon occurs if the problem constraints imply the ordering constraints. The ordering constraints then may serve as redundant constraints [Tsa93], which give no extra propagation, rather than as symmetry breaking constraints. As an example, consider the sport scheduling problem which was introduced in Chapter 3.2.4. In Figure 3.9, one way of modelling the problem is presented. The (extended) weeks over which the tournament is held, as well the periods are indistinguishable. The rows and the columns of T and G are therefore symmetric. Note that we treat T as a 2-d matrix where the rows represent the periods and columns represent the extended weeks, and each entry of the matrix is a pair of variables.

Consider the rows of T. The global cardinality constraints posted on the rows ensure that each of $1 \dots n$ occur exactly twice in every row. This means that in any solution to the problem, the rows are equal, when viewed as multisets. Now consider the columns of T. The *all-different* constraints posted on the columns state that each column is a permutation of $1 \dots n$. This suggests that in any solution to the problem, also the columns are equal when viewed as multisets. The multiset ordering constraint on a pair of rows/columns of T is a logical consequence of the problem constraints which may not give any additional constraint propagation. Therefore, enforcing multiset ordering constraints does not necessarily reduce the size of the search tree. Similarly, imposing multiset ordering constraints on the rows and columns of the matrix model of the quasigroup existence problem (see prob003 in CSPLib) and the BIBD problem (see Chapter 3.2.1) may not be effective.

In summary, it is difficult to assess theoretically how much symmetry we can break and whether we can significantly reduce the size of the search tree by imposing ordering constraints. Also, as discussed in Sections 4.2.3 and 4.2.4, theory cannot distinguish the ordering constraints which impose incomparable orderings. It is therefore desirable to study the ordering constraints explored in Section 4.2 experimentally to judge their effectiveness in breaking row and column symmetries.

4.4 Extensions

In this section, we argue that we can benefit from lexicographic ordering and multiset ordering constraints for breaking symmetry in matrices of arbitrary number of dimensions (Section 4.4.1), for breaking partial symmetry (Section 4.4.2), and finally for breaking value symmetry (Section 4.4.3).

4.4.1 Higher Dimensions

Many problems can be effectively modelled and efficiently solved using a matrix of more than two dimensions. For instance, consider the sport scheduling problem which was introduced in Chapter 3.2.4. In Figure 3.9, one way of modelling the problem is presented. One of the matrices in the model is a 3-d matrix T whose dimensions correspond to n extended weeks, $n/2$ periods, and 2 slots. A variable $T_{i,j,k}$ in this matrix takes a value between 1 and n expressing that a team plays in a particular week in a particular period, in the home or away slot. The matrix T has symmetry along each of the three dimensions: the extended weeks are indistinguishable, and so are the periods and the slots. Another example is the social golfers problem which was also introduced in Chapter 3.2.4. A model of this problem, as shown in Figure 3.8, is a 3-d 0/1 matrix T whose dimensions correspond to w weeks, g groups, and $g * s$ players. A variable $T_{i,j,k}$ in this matrix is 1 iff the kth player plays in group i of week j. As the groups are indistinguishable, and so are the weeks and the golfers, this matrix has symmetry along each of the three dimensions. As in the case of 2-d matrices with row and column symmetry, we can break some of these symmetries by imposing ordering constraints in each symmetric dimension. We now generalise the lexicographic ordering and multiset ordering constraints to any number of dimensions.

Consider a 2-d matrix. If we look along a particular dimension, we see 1-d matrices at right angles to this axis. To break the symmetries, we enforce lexicographic ordering constraints on these matrices, when we treat them as vectors, or we enforce multiset ordering constraints on the matrices, when we treat them as multisets. Now consider a 3-d matrix. If we look along a particular dimension, we see 2-d slices of the matrix that are orthogonal to this axis. To break the symmetries, we need to impose ordering constraints on these slices. One way is first to flatten the slices onto 1-d matrices and then insisting that they are either lexicographically or multiset ordered. In n dimensions, we see slices that are $n-1$ dimensional hypercubes, which can be compared by flattening onto 1-d matrices and insisting that they are either lexicographically or multiset ordered.

Definition 25 *An n-dimensional matrix x of values, with $n \geq 1$, is multi-dimensionally lexicographically ordered iff the following conditions hold:*

$$\begin{array}{lll} \forall i & \text{flatten}(x[i][\,]\ldots[\,]) & \leq_{lex} \text{flatten}(x[i+1][\,]\ldots[\,]) \\ \forall j & \text{flatten}(x[\,][j]\ldots[\,]) & \leq_{lex} \text{flatten}(x[\,][j+1]\ldots[\,]) \\ & & \vdots \\ \forall k & \text{flatten}(x[\,][\,]\ldots[k]) & \leq_{lex} \text{flatten}(x[\,][\,]\ldots[k+1]) \end{array}$$

where $x[\,]\ldots[\,][i][\,]\ldots[\,]$ denotes the $n-1$ dimensional hypercube obtained by taking the slice of x at position i in the dimension where $[i]$ appears in $[\,]\ldots[\,][i][\,]\ldots[\,]$, and where flatten *is used to flatten a slice of a matrix into a 1-d vector and is defined by:*

$$\begin{array}{ll} \text{flatten}(x[1..m]) = x[1..m] & \\ \text{flatten}(x[1..m][\,]\ldots[\,]) = \text{append}(& \text{flatten}(x[1][\,]\ldots[\,]), \\ & \vdots, \\ & \text{flatten}(x[m][\,]\ldots[\,])) \end{array}$$

with $\text{append}(v_1, \ldots, v_n)$ *denoting the left-to-right concatenation of the* 1*-d vectors* $v_1, \ldots, v_n$.

Definition 26 *An n-dimensional matrix x of values, with $n \geq 1$, is multi-dimensionally multiset ordered iff the following conditions hold:*

$$\begin{array}{lll} \forall i & \{\!\{\text{flatten}(x[i][\,]\ldots[\,])\}\!\} & \leq_m \{\!\{\text{flatten}(x[i+1][\,]\ldots[\,])\}\!\} \\ \forall j & \{\!\{\text{flatten}(x[\,][j]\ldots[\,])\}\!\} & \leq_m \{\!\{\text{flatten}(x[\,][j+1]\ldots[\,])\}\!\} \\ & & \vdots \\ \forall k & \{\!\{\text{flatten}(x[\,][\,]\ldots[k])\}\!\} & \leq_m \{\!\{\text{flatten}(x[\,][\,]\ldots[k+1])\}\!\} \end{array}$$

where $x[\,]\ldots[\,][i][\,]\ldots[\,]$ *denotes the* $n-1$ *dimensional hypercube obtained by taking the slice of* x *at position* i *in the dimension where* $[i]$ *appears in* $[\,]\ldots[\,][i][\,]\ldots[\,]$*, and where* flatten *is used to flatten a slice of a matrix into a* 1*-d vector and is defined by:*

$$\begin{aligned}
\text{flatten}(x[1..m]) &= x[1..m] \\
\text{flatten}(x[1..m][\,]\ldots[\,]) &= \text{append}(\;\; \text{flatten}(x[1][\,]\ldots[\,]), \\
&\qquad\qquad\qquad \vdots, \\
&\qquad\qquad\qquad \text{flatten}(x[m][\,]\ldots[\,]))
\end{aligned}$$

with $\text{append}(v_1,\ldots,v_n)$ *denoting the left-to-right concatenation of the* 1*-d vectors* $v_1,\ldots,v_n$.

As in the 2-d case, we can show that multi-dimensional lexicographic ordering or multiset ordering constraints are consistent symmetry breaking constraints. Unfortunately, such constraints may not break all the symmetries as the 2-d counter examples generalise to other numbers of dimensions.

Theorem 4 *Given a matrix with symmetry along each of its* n *dimension, where* $n \geq 1$*, each symmetry class has at least one element where the matrix is multi-dimensionally lexicographically ordered.*

Proof: A proof for the 3-d case is in [FFH$^+$01a]; it generalises to any number of dimensions. QED.

Theorem 5 *Given a matrix with symmetry along each of its* n *dimension, where* $n \geq 1$*, each symmetry class has at least one element where the matrix is multi-dimensionally multiset ordered.*

Proof: Suppose a is an assignment of values to the variables of the matrix. In one dimension of a, we have slices which are $n-1$ dimensional hypercubes. Since multiset ordering is transitive, we can always multiset order the slices by flattening each slice into a one dimensional matrix and treating each matrix as a multiset. By multiset ordering one dimension of a matrix, we do not distinguish the other dimensions. We can still freely permute the slices of the other dimensions, as multiset ordering one dimension is invariant to the permutation of the slices in the other dimensions. This shows that we can always make a multi-dimensionally multiset ordered by permuting the slices in each dimension. QED.

4.4.2 Partial Symmetry

We may only have partial row and/or column symmetry in a matrix, namely when only strict subset(s) of the rows and/or columns are indistinguishable. As an example, consider the steel mill slab design problem which was introduced in Chapter 3.2.2. A model of this problem, as shown in Figure 3.3, uses a 2-d 0/1 matrix O to represent which orders are assigned to which slabs. This matrix has partial row symmetry since only the slabs of the same size are indistinguishable. This matrix has also partial column symmetry because only the orders of the same size and colour are indistinguishable. As another example, consider the progressive party problem which was introduced in Chapter 3.2.5. A matrix model for this problem is shown in Figure 3.10. The matrix H is indexed by the set of time periods and the set of guests. A variable $H_{i,j}$ in this matrix gives the host assigned to guest j at time period i. Even though the columns of H are all symmetric, because the time periods are all indistinguishable, only the rows which correspond to the guests of the same crew size are indistinguishable. Hence, H has partial row symmetry as well as column symmetry. Further examples of partial symmetry are in the matrix models of the template design problem (see Chapter 3.2.2) and the rack configuration problem (see Chapter 3.2.3).

To tackle partial row and/or column symmetry, we impose lexicographic or multiset ordering constraints only on the rows and/or columns that are indistinguishable, assuming that the indistinguishable rows and/or columns are in neighbouring positions. To show that this is a correct way to break partial symmetry, suppose we have partial column symmetry and the symmetric columns are in neighbouring positions. We add an extra row to the top of the matrix, in which we label identically those columns that are subject to permutation, and increase the label as we switch from one subset of the symmetric columns to another subset. Now the columns are all indistinguishable. Enforcing lexicographic or multiset ordering constraints on all the columns are consistent symmetry breaking constraints. As the recursive definition of lexicographic or multiset ordering separates the columns into disjoint sets of permutable columns that have to be ordered, imposing the ordering constraints only on the symmetric rows and/or columns is correct.

4.4.3 Value Symmetry

As argued in the last chapter, we often have (partial) value symmetry in a matrix model. We can deal with symmetric values using the techniques we have developed above for dealing with symmetric variables. A variable V of an n dimensional matrix that takes a value from a domain of indistinguishable values $\{v_1, \ldots, v_m\}$ can be replaced by a vector $\langle V_1, \ldots, V_m \rangle$ of 0/1 variables, with the semantics $V_i = 1 \leftrightarrow V = v_i$. A set variable V taking a set of values from a similar domain of indistinguishable values can also be replaced by a vector of 0/1 variables with the semantics $(V_i = 1 \leftrightarrow v_i \in V)$. Hence, we have introduced $n \times m$ 0/1 variables and constraints. In other words, the (set) variable is replaced by a characteristic function, whose variables take values that are not indistinguishable. This converts indistinguishable values into indistinguishable variables, which become a new dimension in the now $n+1$ dimensional matrix.

As an example, consider the social golfers problem which was introduced in Chapter 3.2.4. A model of this problem, as shown in Figure 3.7, employs a 2-d matrix T of set variables, where each row represents a week and each column represents a group. Each element $T_{i,j}$ of the matrix is a set of golfers that play together as group i in week j. This matrix has row and column symmetry as the weeks are indistinguishable, and so are the groups. This matrix has also value symmetry because the golfers are indistinguishable. Another model of the problem, as shown in Figure 3.8, replaces each set variable of the initial model by a 0/1 vector of integer variables of length $g{*}s$ representing the characteristic function of the set. For example, given 4 golfers, the set of golfers $\{1, 2\}$ is represented by the vector $\langle 0, 0, 1, 1 \rangle$. Hence, we have a 3-d matrix T of variables where $T_{i,j,k} = 1$ iff the kth player plays in group i of week j. As the groups are indistinguishable, and so are the weeks and the golfers, this matrix has symmetry along each of the three dimensions. The advantage of this approach is that we can use the multi-dimensional symmetry breaking constraints to deal simultaneously with symmetric variables and symmetric values.

We can also use the techniques outlined in Section 4.4.2 to deal with values that are only partially symmetric. Consider the progressive party problem which was introduced in Chapter 3.2.5. A model of this problem, as shown in Figure 3.10, uses a 2-d matrix H. A variable $H_{i,j}$ in this matrix takes as a value the host boat visited by guest j in period i. The matrix H has column symmetry, because the time periods are all indistinguishable. The matrix H has partial row symmetry because only the rows which correspond to the guests of the same crew size are indistinguishable. As the host boats of the same capacity are indistinguishable, the matrix H has also partial value symmetry. We can turn this partial value symmetry into a partial variable symmetry by channelling H into a new 3-d 0/1 matrix C. A variable $C_{i,j,k}$ in this new matrix is 1 iff the host boat k is visited by guest j in period i. The new matrix C has partial symmetry along its third dimension.

4.5 Breaking All Symmetries

It is always possible to break all row and column symmetries of a matrix of decision variables by using the method proposed in [CGLR96] and described in Section 4.2.1. This method requires posting a lexicographic ordering constraint for each symmetry. As the number of row and column symmetries grows super-exponentially, it is not practical to break all symmetries in this way. We therefore identify special and useful cases where all row and column symmetries can easily be broken.

Quite often, we have a matrix with row and column symmetry and the variables of the matrix are constrained to be all different. As an example, consider the sport scheduling problem which was introduced in Chapter 3.2.4. In Figure 3.9, one way of modelling the problem is presented. As the 2-d matrix G represents the games, an *all-different* on G ensures that every team plays against every other team. The weeks over which the tournament is held, as well the periods are indistinguishable. The rows and the columns of G are therefore symmetric. Another example is the problem of finding graceful graphs which is about deciding whether a graph has a graceful labelling [Gal02]. Petrie and Smith have studied a number of graphs whose gracefulness are unknown, and showed that a class of graphs can be modelled using a matrix of variables with an *all-different* constraint on the variables [PS03], where the rows and columns are symmetric.

Theorem 6 *If a 2-d matrix with row and column symmetry, as well as with a constraint requiring all the values in that matrix to be distinct, has a solution, then each symmetry class has:*

- *a unique member with the largest value placed in the bottom-right corner as well as the last row and the last column ordered;*
- *a unique member with the largest value placed in the top-right corner as well as the first row and the last column ordered;*
- *a unique member with the largest value placed in the top-left corner as well as the first row and the first column ordered;*
- *a unique member with the largest value placed in the bottom-left corner as well as the last row and the first column ordered.*

Proof: A proof for the first case is in [FFH$^+$02]. Dual arguments hold if the largest value is placed in one of the remaining three corners. QED.

As a result, enforcing the largest element to be in one corner, as well as enforcing the row and the column, where the largest value is, to be ordered breaks all row and column symmetries. In fact, we will break all symmetries even if the other rows and columns contain repeated values. By enforcing the row and the column, where the largest value is, to be ordered, we prevent any permutation of the rows and columns. We do not therefore need every value in the matrix to be distinct.

Another special case which is of interest is 0/1 matrices with row and column symmetry and the constraints enforcing a single one in each row or column. Such matrices frequently arise when modelling problems involving demand constraints. As an example, consider the steel mill slab design problem introduced in Chapter 3.2.2. Every order is to be packed onto exactly one slab. A model of this problem, as shown in Figure 3.3, uses a 2-d 0/1 matrix O to decide which orders are packed onto which slabs. Sum constraints on the columns of O ensure that every order is packed onto one slab:

$$\forall i \in \mathcal{O}rders\,.\ \sum_{j \in \mathcal{S}labs} O_{i,j} = 1$$

The matrix O has partial row symmetry because the slabs of the same size are indistinguishable. The matrix O has also partial column symmetry because the orders of the same size and colour are indistinguishable. A similar problem is the rack configuration problem introduced in Chapter 3.2.3. Every card is to be plugged into exactly one rack. In the second model of the problem shown in Figure 3.6, we manipulate the cards rather than the number of cards of a given type plugged into a rack. Hence, a 2-d 0/1 matrix C is used to determine which cards are plugged into which racks. Sum constraints on the columns of C ensure that every card is plugged into one rack:

$$\forall i \in \mathcal{C}ards\,.\ \sum_{j \in \mathcal{R}acks} C_{i,j} = 1$$

The matrix C has partial row symmetry because the racks of the same model are indistinguishable. The matrix C has also partial column symmetry because the cards of the same type are indistinguishable.

The following theorem shows that in such cases it is possible to break all row and column symmetries. A similar theorem appears also in [Shl01].

Theorem 7 *Given a 2-d 0/1 matrix with row and column symmetry, as well as with a constraint requiring every row to have a single 1, each symmetry class has a unique member with the rows ordered lexicographically, as well as the columns ordered lexicographically and by their sums. Similarly, given a 2-d 0/1 matrix with row and column symmetry, as well as with a constraint requiring every column to have a single 1, each symmetry class has a unique member with the columns ordered lexicographically, as well as the rows ordered lexicographically and by their sums.*

Proof: See [FFH$^+$02]. QED.

4.6 Related Work

Even though row and column symmetries have not been previously identified and studied as a class of symmetries, instances of such symmetries have been encountered and addressed in a simple way. For instance, the symmetry between the racks of the same rack model in the rack configuration problem is tackled in [ILO02] by insisting the first elements of the symmetric rows in the matrix modelling the problem to be equal or ordered.

This is a very weak form of symmetry breaking, as it can well be the case that the first elements are assigned the same values. In this case, symmetry between the remaining elements of the rows are not eliminated.

Crawford *et al.* in [CGLR96] propose a method of eliminating all symmetries which requires posting a lexicographic ordering constraint for each symmetry. An $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries, which increase super-exponentially. Whilst in theory we can eliminate all symmetries using the method of [CGLR96], in practice there are too many constraints to post and propagate. We call the set of constraints generated by the method of [CGLR96] to break all row and column symmetries as the full symmetry breaking constraints. Our approach to tackling row and column symmetries is instead posting ordering constraints between the symmetric rows and columns. Lexicographic ordering constraints on the rows and columns (called double-lex constraints) are a subset of the full symmetry breaking constraints. A similar conclusion is also due to Shlyakhter in [Shl01] in parallel to [FFH$^+$01a], but his results are restricted to 0/1 matrices where all the rows and/or columns are symmetric. Our results are, however, applicable to matrices of any domain, with (partial) row and (partial) column symmetry.

Subsequent to [FFH$^+$01a][FFH$^+$02], the full symmetry breaking constraints have been analysed by a number of researchers. Flener and Pearson have attempted simplifying these constraints to obtain a polynomial size subset which is small enough to break all symmetries [FP02a]. They conjecture that for 0/1 matrices there is a strict subset of the full symmetry breaking constraints which break all symmetries. Even if the conjecture is true, the size of the subset may be super-polynomial, and it is not clear how the study can be extended for matrices of any domain. Frisch *et al.*, on the other hand, have shown that a subset of the full symmetry breaking constraints impose that the first row is lexicographically less than or equal to all permutations of all other rows [FJM03]. It is also argued that it is consistent to enforce this subset of the constraints (called the *allperm* constraint) together with the double-lex constraints. The experimental results in [FJM03] show that enforcing both all-perm and double-lex constraints reduces the search effort more than the double-lex constraints alone, but still leaves some symmetry. Puget in [Pug03b] states that it is not necessary to impose the full symmetry breaking constraints at the start of search for solutions. His approach to breaking row and column symmetries is to impose some of the full symmetry breaking constraints during search. At each node of the search tree, only the constraints for symmetries that are not yet broken are imposed. The computational results in [Pug03b] show a clear benefit of using this technique in preference to the double-lex constraints to tackle row and column symmetries effectively and efficiently.

Many symmetry breaking methods have been devised in the past years, such as SES [BW99][BW02], SBDS [GS00], and SBDD [FM01][FSS01], all of which can directly be used to break all row and column symmetries. Unfortunately, these techniques work well when the number of symmetries is small, and are not practical to use when the number of symmetries is large, as in the case of row and columns symmetries. Following [FFH$^+$01a], several researchers have started investigating ways to improve these techniques to eliminate large number of symmetries much more efficiently.

As an $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries, breaking all symmetries using SBDS would require as many SBDS functions, so SBDS can only be used for small matrices (e.g. 3×3 and 4×4). Therefore, in [GS01], using a subset of the full SBDS functions is proposed, to reduce row and column symmetries in large matrices. Since row (resp. column) symmetry alone can entirely be eliminated by row (resp. column) transpositions, a promising subset is the row transpositions *and* the column transpositions, which require only $O(n^2)$ SBDS functions for an $n \times n$ matrix. Adding all combinations of a row transposition and a column transposition to the row and column transpositions would eliminate even more symmetry. This, however, increases the number of SBDS functions to $O(n^4)$. In [GS01], combining SBDS with symmetry breaking constraints is also explored. One can for instance use column transpositions in SBDS to remove the column symmetry. This would require only $O(n^2)$ SBDS functions. To reduce the row symmetry, one can add constraints that order the rows by their sums.

Just like the double-lex constraints, none of the methods described in [GS01] eliminate all row and column symmetries. Some experiments are carried out in [GS01] to see which of these methods and the double-lex constraints breaks more symmetry. According to the experimental results, the double-lex constraints eliminate less symmetric solutions than the combination of the column transpositions with the row-sum ordering constraints, as well as less than the row and column transpositions and their combinations. In the experiments, however, any constraints on the matrix to be constructed are ignored. Only the size of the matrix and the domain size of the elements in the matrix are specified, and then a set of matrices such that no two can be generated from each other by permuting the rows and/or columns are searched. The methods are then compared only with respect to the number of matrices found. For instance, imposing double-lex constraints on an 3×3 0/1 matrix returns 45 matrices. Moreover, the lexicographic ordering constraints are implemented using the arithmetic constraint approach, which is expensive to calculate as the matrix size gets larger (see Chapter 5.5.1 for a detailed discussion on this). It is therefore not clear which of the techniques gives the most significant improvement to the search effort.

As the symmetries of a CSP form a group, computational group theory methods have proven useful in dealing with large number of symmetries efficiently. In [GHK02], an implementation of SBDS combined with the GAP system [GAP00] (called GAP-SBDS) is presented, which allows the symmetries of a CSP to be represented by the generators of the group. GAP-SBDS can handle around 10^9 symmetries, many more than the initial implementation of SBDS [GS00]. This is a promising approach, but the experiments reported show that GAP-SBDS is much slower than the double-lex constraints to solve a BIBD instance represented by a 6×10 matrix with row and column symmetry. In [GHKL03], a generic implementation of SBDD is introduced (called GAP-SBDD), in which the GAP system forms the symmetry group. Also, GAP-SBDD eliminates the need of implementing a dominance checker, either directly [FSS01][FM01] or as a separate CSP [Pug02c], because dominance checks are performed by GAP via the search for the dominating group element. GAP-SBDD is able to handle around 10^{23} symmetries, many more than GAP-SBDS. Even though the authors experiment with row and column symmetries using the BIBD problem, they do not report whether GAP-SBDD is faster or slower than the double-lex constraints. Later, the authors state that they do not report the best run-times compared to the related work. It is hard to tell therefore if, unlike GAP-SBDS, GAP-SBDD is faster than the double-lex constraints to solve BIBD instances represented by matrices with row and column symmetry.

Puget in [Pug02c] improves SBDD [FSS01][FM01] by generalising the way the nogoods are recorded. This significantly reduces the number of dominance checks, as well the size and the number of nogoods, giving superior performance. Puget does not use his method in [Pug02c] to tackle row and column symmetries. However, as he indicates in the presentation of his work [Pug02a], this version of SBDD together with the double-lex constraints solves many BIBD instances modelled by matrices with row and column symmetry much faster than the double-lex constraints alone. We are also informed by Puget [Pug02b] that without the double-lex constraints, SBDD alone cannot cope with the large number of row and column symmetries.

Another implementation of SBDD [FSS01][FM01], which is combined with the **nauty** system [nau], is due to Pearson [Pea03]. During search, each nogood and partial solution is converted into a graph, and graph isomorphism is used for dominance check. Pearson experiments with some instances of the problem of generating comma free codes [Lam03], which can be modelled by a matrix with row symmetry. The results of the experiments show that as the instances become larger, this implementation of SBDD reduces the search effort more than enforcing lexicographic ordering constraints on the rows. It is, however, not clear whether the implementation is engineered for the comma free codes or works for any CSP.

4.7 Summary

In this chapter, we have tackled an important class of symmetries: row and column symmetries of a matrix of decision variables. An $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries, which makes it very difficult and costly to break all the symmetries. To deal with such symmetries in an easy and efficient way, we have investigated what ordering constraints we can post on the rows and columns.

We have shown that we can insist the rows and columns to be lexicographically ordered. Alternatively, we can treat each row and column of a matrix as a multiset and impose that the rows and columns are multiset ordered. These symmetry breaking ordering constraints guarantee that we can find at least one solution from each equivalence class of solutions. One of the nice features of using multiset ordering for breaking symmetry is that by constraining the rows (resp. columns) of a matrix to be multiset ordered, we do not distinguish the columns (resp. rows). We can still freely permute the columns (resp. rows), as multiset ordering the rows (resp. columns) is invariant to column (resp. row) permutation. We have therefore suggested combining multiset ordering constraints in one dimension of a matrix with lexicographic ordering constraints in the other dimension, so as to obtain new symmetry breaking ordering constraints. We have also argued that lexicographic ordering and multiset ordering are incomparable. This has lead us to discover that imposing lexicographic ordering constraints on the rows and columns, imposing multiset ordering constraints on the rows and columns, and imposing one ordering in one dimension and the other ordering in the other dimension of a matrix are all incomparable.

We have extended these results to cope with symmetries in any number of dimensions, with partial symmetries, and with symmetric values. We have also identified a couple of special and useful cases where all compositions of the row and column symmetries can be broken by means of adding only a linear number of constraints. Finally, we have compared with related work.

Symmetry breaking is very easy by enforcing an ordering on the symmetric objects via some ordering constraints. However, we need to be able to post and propagate such constraints effectively and efficiently. Moreover, we may not eliminate all the symmetries by enforcing the ordering constraints. Furthermore, the presence of the ordering constraints may not reduce the size of the search tree. Finally, theory cannot distinguish the ordering constraints which impose incomparable orderings. These lead us to the following questions:

- how can we post and propagate the lexicographic ordering constraint effectively and efficiently?
- how can we post and propagate the multiset ordering constraint effectively and efficiently?
- are the ordering constraints explored in Section 4.2 often not redundant and bring additional pruning?
- are the ordering constraints explored in Section 4.2 effective in breaking row and column symmetries in practice?
- how do the ordering constraints explored in Section 4.2 compare in practice?

Addressing these questions is important and will be the concern of the following chapters.

Chapter 5

Lexicographic Ordering Constraint

5.1 Introduction

Given two vectors $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a lexicographic ordering constraint as $\vec{X} \leq_{lex} \vec{Y}$ and a strict lexicographic ordering constraint as $\vec{X} <_{lex} \vec{Y}$. The lexicographic ordering constraint $\vec{X} \leq_{lex} \vec{Y}$ ensures that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ respectively are lexicographically ordered according to Definition 14. That is, either $\vec{x} = \vec{y}$, or $\exists k\ 0 \leq k < n$ such that $x_i = y_i$ for all $0 \leq i < k$ and $x_k < y_k$. The strict lexicographic ordering constraint disallows $\vec{x} = \vec{y}$.

We can utilise lexicographic ordering constraints for breaking row and column symmetries, but how can we post and propagate these constraints effectively and efficiently? In this chapter, we design global constraints for lexicographic orderings, each of which encapsulates its own filtering algorithm.

This chapter is organised as follows. In Section 5.2, we present a filtering algorithm for the lexicographic ordering constraint $\vec{X} \leq_{lex} \vec{Y}$. Then in Section 5.3, we discuss the complexity of the algorithm, and prove that the algorithm is correct and complete. In Section 5.4, we show how we can extend the algorithm to obtain filtering algorithms for $\vec{X} <_{lex} \vec{Y}$ and $\vec{X} \neq \vec{Y}$, to detect entailment, and to handle vectors of any length. Alternative approaches to propagating the lexicographic ordering constraint are discussed in Section 5.5. We demonstrate in Section 5.6 that decomposing a chain of lexicographic ordering constraints into lexicographic ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. In Section 5.7, we show how we can generalise the algorithm for vectors whose variables are repeated and shared. Related work is discussed in Section 5.8 and computational results are presented in Section 5.9. Finally, before summarising in Section 5.11, we give in Section 5.10 the details of the implementation.

5.2 A Filtering Algorithm for Lexicographic Ordering Constraint

In this section, we present a filtering algorithm for the lexicographic ordering constraint which either detects the disentailment of $\vec{X} \leq_{lex} \vec{Y}$ or prunes inconsistent values so as to achieve GAC on $\vec{X} \leq_{lex} \vec{Y}$. Before giving the details in Section 5.2.3, we first sketch the main features of the algorithm in Section 5.2.1 and then in Section 5.2.2 provide the theoretical background from which the algorithm is derived.

5.2.1 A Worked Example

The main idea behind the algorithm is to maintain two pointers α and β which save us from repeatedly traversing the vectors. The pointer α points to the most significant index of the vectors above which the variables are all ground and they are pairwise equal. The pointer β points to the most significant index such that $\vec{X}_{\beta \to n-1} >_{lex} \vec{Y}_{\beta \to n-1}$ is *true*. That is, starting from index β, the vectors are ordered the wrong way around no matter what values are assigned to the future variables. If this is not the case then β points to ∞.

Consider the lexicographic ordering constraint $\vec{X} \leq_{lex} \vec{Y}$ with:

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

We traverse the vectors once in linear time in order to initialise the pointers. 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 $\alpha = 2$ as Y_2 is not ground.

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

We give an initial temporary value -1 to β and then look for the position of β starting from α. At index 2 we have $min(X_2) = max(Y_2)$, which means that $X_2 \geq Y_2$ is *true*. Starting from this index, the vectors could be ordered the wrong way around for any combination of assignments to the future variables, so we set $\beta = 2$. We move to index 3, where $min(X_3) = max(Y_3)$. Since β is supposed to be the leftmost index and β has already been set a value before, we move to index 4. Here, it is possible to satisfy $X_4 \leq Y_4$, therefore β cannot point to 2. We reset $\beta = -1$ and move to index 5. We now set $\beta = 5$ because $min(X_5) = max(Y_5)$. At index 6, we have $min(X_6) > max(Y_6)$ which means that $X_6 > Y_6$ is *true*. However, β is pointing to an earlier index. To keep β as the most significant index we do not change its value.

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

The algorithm restricts domain prunings to the index α. As variables are assigned, α moves monotonically right, β moves monotonically left, and they are both bounded by n. The constraint is disentailed if the pointers meet. Down one branch of the search tree until all the $2n$ variables are assigned, α moves at most n positions. On the average, α moves one position for each assignment. Hence, the algorithm runs amortised[1] in time $O(b)$, where b is the cost of adjusting the bounds of a variable.

[1] In an amortised analysis, the time required to perform a sequence of operations is averaged over all the operations performed [Tar85].

Consider the vectors again. As the vectors above α are ground and equal, there is no support for any value in $\mathcal{D}(Y_\alpha)$ which is less than $min(X_\alpha)$. We therefore remove 0 and 1 from $\mathcal{D}(Y_\alpha)$ and increment α to 3:

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

Similar to the previous case, there is no support for any value in $\mathcal{D}(X_\alpha)$ which is greater than $max(Y_\alpha)$. We therefore remove 3 and 4 from $\mathcal{D}(X_\alpha)$ and increment α to 4:

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

Since α has now reached $\beta-1$, there is no support for any value in $\mathcal{D}(X_\alpha)$ which is greater than or equal to $max(Y_\alpha)$. Similarly, there is no support for any value in $\mathcal{D}(Y_\alpha)$ which is less than or equal to $min(X_\alpha)$. We must therefore enforce AC($X_\alpha < Y_\alpha$). That is, we remove 4 and 5 from $\mathcal{D}(X_\alpha)$, and also 0 and 1 from $\mathcal{D}(Y_\alpha)$:

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

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

5.2.2 Theoretical Background

The filtering algorithm of the lexicographic ordering constraint is based on two theoretical results which show in turn when $X \leq_{lex} Y$ is disentailed and what condition ensures GAC on $X \leq_{lex} Y$.

As seen in the running example of the algorithm, the pointers α and β play an important role. These pointers are in fact the main ingredients of the theoretical foundation of the algorithm. We start by defining the pointer α.

Definition 27 *Given $\vec{X}$ and $\vec{Y}$ where $\exists k\ 0 \leq k < n$ such that $\neg(X_k \doteq Y_k)$, the pointer α is set to the index in $[0,n)$ such that:*

$$\neg(X_\alpha \doteq Y_\alpha) \wedge$$

$$\forall i\ 0 \leq i < \alpha \,.\, X_i \doteq Y_i$$

Informally, α points to the most significant index in $[0,n)$ such that all the variables above it are ground and pairwise equal. That is, $\vec{X}_{0\to\alpha-1} = \vec{Y}_{0\to\alpha-1}$ is *true*. The importance of α is that it gives us the most significant index where the vectors can either take equal values or be ordered. Note that we define α on a pair of vectors that are not ground and equal.

A constraint is said to be *disentailed* when the constraint is $false$. In the case of lexicographic ordering constraint, it is important to know the most significant index after which the constraint is disentailed. This knowledge helps us to decide whether the vectors can be assigned equal values or not at index α. For this purpose, we introduce the pointer β.

Definition 28 *Given $\vec{X}$ and $\vec{Y}$, the pointer β is set either to the index in $[\alpha, n)$ such that:*

$$\exists k\ \beta \leq k < n\,.\,(min(X_k) > max(Y_k) \wedge \forall i\ \beta \leq i < k\,.\,min(X_i) = max(Y_i)\ )$$

where:

$$min(X_{\beta-1}) \neq max(Y_{\beta-1}) \wedge (\forall i\ \alpha \leq i < \beta\,.\,min(X_i) \leq max(Y_i)\ )$$

or (if this is not the case) to ∞.

Informally, the pointer β points to the most significant index in $[\alpha, n)$ such that $\vec{X}_{\beta \to n-1} >_{lex} \vec{Y}_{\beta \to n-1}$ is *true*. That is, starting from index β, the vectors are ordered the wrong way around no matter what values are assigned to the future variables. If this is not the case then β points to ∞. It is important to set β to ∞ when such an index does not exist. Consider two vectors $\vec{X} = \langle X_0 \rangle$ and $\vec{Y} = \langle Y_0 \rangle$ with $\mathcal{D}(X_0) = \mathcal{D}(Y_0) = \{0, 1\}$. The pointer α is set to 0 and if we set the pointer β to 1, then our rule would prune 1 from $\mathcal{D}(X_0)$ and 0 from $\mathcal{D}(X_1)$, which is wrong. To avoid such situations, the pointer β is set to a value greater than the length of the vectors.

We can make use of the pointers α and β to prune inconsistent values as well as detect disentailment. Moreover, the pointers save us from repeatedly traversing the vectors.

Theorem 8 *Given $\beta = \alpha$, $\vec{X} \leq_{lex} \vec{Y}$ is disentailed.*

Proof: By Definition 27, $\vec{X}_{0 \to \alpha-1} = \vec{Y}_{0 \to \alpha-1}$ is *true*, and by Definition 28 $\vec{X}_{\alpha \to n-1} >_{lex} \vec{Y}_{\alpha \to n-1}$ is *true*, whatever the remaining assignments are. Hence, $\vec{X} >_{lex} \vec{Y}$ is *true*. QED.

It is not hard to show that given $\beta > \alpha$, generalised arc-inconsistent values can exist only in the interval $[\alpha, min\{n, \beta\})$. As the following theorem shows, generalised arc-inconsistent values exist only at index α indeed.

Theorem 9 *Given $\beta > \alpha$, GAC($\vec{X} \leq_{lex} \vec{Y}$) iff:*

1. $\beta = \alpha + 1 \rightarrow AC(X_\alpha < Y_\alpha)$
2. $\beta > \alpha + 1 \rightarrow AC(X_\alpha \leq Y_\alpha)$

($\Rightarrow$) Assume $\vec{X} \leq_{lex} \vec{Y}$ is GAC but either $X_\alpha < Y_\alpha$ is not AC when $\beta = \alpha + 1$ or $X_\alpha \leq Y_\alpha$ 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 ground and pairwise equal above α, a or b lacks support from all the variables in the vectors. This contradicts that $\vec{X} \leq_{lex} \vec{Y}$ is GAC.
($\Leftarrow$) All variables above α are ground and pairwise equal. The assignment $X_\alpha \gtrdot Y_\alpha$ cannot be extended to a consistent assignment to satisfy $\vec{X} \leq_{lex} \vec{Y}$, because in this case $\vec{X}$ is already greater than $\vec{Y}$ at position α. The reverse holds for the assignment $X_\alpha \lessdot Y_\alpha$. In this case, any future assignment satisfies $\vec{X} \leq_{lex} \vec{Y}$ as $\vec{X}$ is already less than $\vec{Y}$ at position α. Whether assigning the same values to X_α and Y_α is consistent or not depends on the index β points to.

If $\beta = \alpha + 1$ then:

$$\exists k\ \alpha < k < n\,.\,(min(X_k) > max(Y_k) \wedge \forall i\ \alpha < i < k\,.\,min(X_i) = max(Y_i)\ )$$

Algorithm 1: `Initialise`

```
Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
Result : $\alpha$ and $\beta$ are initialised, GAC($\vec{X} \leq_{lex} \vec{Y}$)
$i := 0$;
while $i < n \ \wedge \ X_i \doteq Y_i$ do $i := i + 1$;
if $i = n$ then return;
else $\alpha := i$;
$\beta := -1$;
while $i \neq n \ \wedge \ min(X_i) \leq max(Y_i)$ do
    if $min(X_i) = max(Y_i)$ then
        if $\beta = -1$ then $\beta := i$;
    else
        $\beta := -1$;
    end
    $i := i + 1$;
end
if $i = n$ then $\beta := \infty$;
else if $\beta = -1$ then $\beta := i$;
if $\alpha = \beta$ then fail;
LexLeq($\alpha$);
```

That is, there is an index k between α and n, and $X_k > Y_k$ is *true*. Moreover, $X_i \geq Y_i$ for all $\alpha < i < k$ is *true*. The assignment $X_\alpha \doteq Y_\alpha$ cannot therefore be extended to a consistent assignment satisfying $\vec{X} \leq_{lex} \vec{Y}$. This proves that AC($X_\alpha < Y_\alpha$) implies GAC($\vec{X} \leq_{lex} \vec{Y}$) when $\beta = \alpha + 1$.

If $\beta > \alpha + 1$ then we consider two cases, namely when $\alpha + 1 < \beta < n$ and when $\beta = \infty$. In the first case:

$$min(X_{\beta-1}) \neq max(Y_{\beta-1}) \wedge (\forall i \ \alpha \leq i < \beta \, . \, min(X_i) \leq max(Y_i) \ )$$

That is, we can find values in $\mathcal{D}(X_i)$ and $\mathcal{D}(Y_i)$ for all $\alpha < i < \beta - 1$ to satisfy $X_i \leq Y_i$, and in $\mathcal{D}(X_{\beta-1})$ and $\mathcal{D}(Y_{\beta-1})$ to satisfy $X_{\beta-1} < Y_{\beta-1}$. In the second case we have:

$$\forall i \ \alpha \leq i < n \, . \, min(X_i) \leq max(Y_i)$$

That is, we can find values in $\mathcal{D}(X_i)$ and $\mathcal{D}(Y_i)$ now for all $\alpha < i < n$ to satisfy $X_i \leq Y_i$. Either case shows that the assignment $X_\alpha \doteq Y_\alpha$ can be extended to a consistent assignment satisfying $\vec{X} \leq_{lex} \vec{Y}$, and proves that AC($X_\alpha \leq Y_\alpha$) implies GAC($\vec{X} \leq_{lex} \vec{Y}$) when $\beta > \alpha + 1$. QED.

5.2.3 Algorithm Details

Based on Theorems 8 and 9, we have designed an efficient linear time filtering algorithm, `LexLeq`, which either detects the disentailment of $\vec{X} \leq_{lex} \vec{Y}$ or prunes inconsistent values so as to achieve GAC on $\vec{X} \leq_{lex} \vec{Y}$.

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

Algorithm 2: `LexLeq`(i)

```
      Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, Integer $i$
      Result : GAC($\vec{X} \leq_{lex} \vec{Y}$)
1     if $i = \alpha \ \wedge \ i+1 = \beta$ then
1.1       AC($X_i < Y_i$);
      end
2     if $i = \alpha \ \wedge \ i+1 < \beta$ then
2.1       AC($X_i \leq Y_i$);
2.2       if $X_i \doteq Y_i$ then UpdateAlpha($i+1$);
      end
3     if $\alpha < i < \beta$ then
3.1       if $(i = \beta - 1 \ \wedge \ min(X_i) = max(Y_i)\ ) \ \vee \ min(X_i) > max(Y_i)$ then
3.1.1         UpdateBeta($i-1$);
          end
      end
```

Line 2 of `Initialise` traverses $\vec{X}$ and $\vec{Y}$, starting at index 0, until either it reaches the end of the vectors (all pairs of variables are ground and equal), or it finds an index where the pair of variables are not ground and equal. In the first case, the algorithm returns (line 3) as $\vec{X} \leq_{lex} \vec{Y}$ is *true*. In the second case, α is set to the most significant index where the pair of variables are not ground and equal (line 4). The vectors are traversed in line 6, starting at index α, until either the end of the vectors are 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, β is set to ∞ (line 7). In the second case, β is guaranteed to be at most i (line 8). If, however, there exist a pair of sub-vectors $\vec{X}_{h\to i-1}$ and $\vec{Y}_{h\to i-1}$ such that $min(X_j) = max(Y_j)$ for all $h \leq j \leq i-1$, then β can be revised to h (lines 6.1-6.1.1).

If $\alpha = \beta$ then disentailment is detected and `Initialise` terminates with failure (line 9). Theorem 9 established that GAC($\vec{X} \leq_{lex} \vec{Y}$) iff: AC($X_\alpha < Y_\alpha$) when $\beta = \alpha + 1$, and AC($X_\alpha \leq Y_\alpha$) when $\beta > \alpha + 1$. We can therefore restrict pruning to the index α. In line 10, the filtering algorithm `LexLeq(`α`)` is called if $\beta > \alpha$.

When $\vec{X} \leq_{lex} \vec{Y}$ is GAC, every value a in $\mathcal{D}(X_\alpha)$ is supported by $max(Y_\alpha)$, and also by $\langle min(X_{\alpha+1}), \ldots, min(X_{min\{n,\beta\}-1})\rangle$ and $\langle max(Y_{\alpha+1}), \ldots, max(Y_{min\{n,\beta\}-1})\rangle$ if $a = max(Y_\alpha)$. Similar argument holds for the values in $\mathcal{D}(Y_\alpha)$. Every value a in $\mathcal{D}(Y_\alpha)$ is supported by $min(X_\alpha)$, and also by $\langle min(X_{\alpha+1}), \ldots, min(X_{min\{n,\beta\}-1})\rangle$ and $\langle max(Y_{\alpha+1}), \ldots, max(Y_{min\{n,\beta\}-1})\rangle$ if $a = min(X_\alpha)$. Therefore, `LexLeq(`i`)` is also called by the event handler whenever $min(X_i)$ or $max(Y_i)$ of some i in $[\alpha, min\{n, \beta\})$ changes.

In Algorithm 2, we show the steps of `LexLeq`. Lines 1-1.1, 2-2.2, and 3-3.1.1 are mutually exclusive, and will be referred to as blocks 1, 2, and 3, respectively.

Block 1: `AC(`$X_i < Y_i$`)` maintains arc-consistency on $X_i < Y_i$. This is implemented as follows. If $max(X_i) < max(Y_i)$ then all the other elements in $\mathcal{D}(X_i)$ are supported. Otherwise, $max(X_i)$ is tightened. Similarly, if $min(X_i) < min(Y_i)$ then all the other elements in $\mathcal{D}(Y_i)$ are supported. Otherwise, $min(Y_i)$ is tightened. If $i = \alpha \ \wedge \ i+1 = \beta$ then we need to ensure that AC($X_i < Y_i$). Now $\vec{X} \leq_{lex} \vec{Y}$ is GAC.

Block 2: If $i = \alpha \ \wedge \ i+1 < \beta$ then we need to ensure AC($X_i \leq Y_i$). Now $\vec{X} \leq_{lex} \vec{Y}$ is GAC. If X_i and Y_i are ground and equal after executing line 2.1 then α is updated by calling `UpdateAlpha(`$i+1$`)`.

```
Procedure UpdateAlpha($i$)
1     if $i = \beta$ then fail;
2     if $i = n$ then return;
3     if $\neg(X_i \doteq Y_i)$ then
3.1       $\alpha := i$;
3.2       LexLeq($i$);
      else
4.1       UpdateAlpha($i+1$);
      end
```

In lines 3 and 4.1 of UpdateAlpha(i), the vectors are traversed until the most significant index k where $\neg(X_k \doteq Y_k)$ is found. If such an index does not exist then the vectors are ground and equal, so the procedure returns (line 2). Otherwise, α is set to k (line 3.1). LexLeq is then called with this new value of α (line 3.2).

Block 3: If $\alpha < i < \beta$ then i now might be a more significant position for β, in which case we need to update β. The condition for updating β is derived from Definition 28: at i we either have $min(X_i) > max(Y_i)$, or i is $\beta - 1$ and $min(X_i) = max(Y_i)$. The pointer β is updated by calling UpdateBeta($i - 1$).

```
Procedure UpdateBeta($i$)
1     if $i + 1 = \alpha$ then fail;
2     if $min(X_i) < max(Y_i)$ then
2.1       $\beta := i + 1$;
2.2       LexLeq($i$);
      else
3.1       if $min(X_i) = max(Y_i)$ then UpdateBeta($i - 1$);
      end
```

In lines 2 and 3.1 of UpdateBeta(i), the vectors are traversed until the most significant index k where $min(X_k) < max(Y_k)$ is found. The pointer β is set to $k + 1$ (line 2.1). LexLeq(k) is then called in line 2.2 in case $k = \alpha$ and we need to ensure AC($X_k < Y_k$).

When updating α or β, if these pointers meet then disentailment is detected and thus failure is established (line 1 in UpdateAlpha and UpdateBeta). This situation can only arise if the event queue contains several domain prunings due to other constraints, for the following reasons. After initialising the pointers, the lexicographic ordering constraint is either disentailed or is GAC. In the second case, every value is supported. If after every single assignment we enforce GAC, this property persists. Therefore, we can only fail when the queue contains a series of updates which must be dealt with simultaneously.

When we prune a value, we do not need to check recursively for previous support. In the worst case α moves one index at a time until it reaches $min\{n, \beta\} - 1$ and on each occasion AC is enforced. This tightens $max(X_i)$ and $min(Y_i)$ without touching $min(X_i)$ and $max(Y_i)$ for all $\alpha \leq i < min\{n, \beta\}$, which provide support for the values in the vectors. The exception is when a domain wipe out occurs. In this case, the constraint is disentailed and the algorithm fails.

5.3 Theoretical Properties

The algorithm `Initialise` runs in linear time in the length of the vectors. The filtering algorithm `LexLeq` runs amortised in constant time, but in the worst in linear time in the length of the vectors.

Theorem 10 `Initialise` *runs in time $O(n)$.*

Proof: In the worst case both vectors are traversed once from the beginning till the end. QED.

Theorem 11 `LexLeq` *runs in time $O(nb)$ where b is the cost of adjusting the bounds of a variable, but runs amortised in time $O(b)$.*

Proof: Since arc-consistency on $X_i < Y_i$ or $X_i \leq Y_i$ is achieved by adjusting $max(X_i)$ and $min(Y_i)$, the worst-case complexity of `AC(`$X_i < Y_i$`)` and `AC(`$X_i \leq Y_i$`)` is $O(b)$. We now analyse `LexLeq` block-by-block.

Block 1: This block of the algorithm has an $O(b)$ complexity.

Block 2: In the worst case α moves one index at a time and on each occasion AC is enforced. Hence, this block of the algorithm gives an $O(nb)$ complexity.

Block 3: In the worst case the whole vectors are traversed to update β. Once β is set to its new value, `LexLeq(`i`)` is called recursively. In this recursive call, since $i = \beta - 1$ and $min(X_i) < max(Y_i)$, only **Block 1** can be executed to ensure AC($X_\alpha < Y_\alpha$) in case β is now $\alpha + 1$. Hence, this block of the algorithm gives an $O(n + b)$ complexity.

The worst-case complexity of `LexLeq` is bounded by $O(nb)$. The pointer α moves monotonically right, β moves monotonically left, and they are both bounded by n. The lexicographic ordering constraint is disentailed if the pointers meet. Down one branch of the search tree until all the $2n$ variables are assigned, α moves at most n positions. On the average, α moves one position for each assignment. Hence, the amortised complexity of `LexLeq` is bounded by $O(b)$. QED.

Both `Initialise` and `LexLeq` are correct and complete.

Theorem 12 `Initialise` *initialises the pointers α and β as per their definitions. Then it either establishes failure if $\vec{X} \leq_{lex} \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC($\vec{X} \leq_{lex} \vec{Y}$).*

Proof: Line 2 of `Initialise` traverses $\vec{X}$ and $\vec{Y}$, starting at index 0, until either it reaches the end of the vectors (all pairs of variables are ground and equal), or it finds an index where the pair of variables are not ground and equal. In the first case, we have GAC($\vec{X} \leq_{lex} \vec{Y}$) and the algorithm returns (line 3). In the second case, α is set to the most significant index where the pair of variables are not ground and equal (line 4) as per Definition 27. The vectors are traversed in line 6, starting at index α, until either the end of the vectors are 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, β is set to ∞ (line 7) as per Definition 28. In the second case, β is guaranteed to be at most i (line 8). If, however, there exist a pair of sub-vectors $\vec{X}_{h \to i-1}$ and $\vec{Y}_{h \to i-1}$ such that $min(X_j) = max(Y_j)$ for all $h \leq j \leq i - 1$, then β can be revised to h (lines 6.1-6.1.1) as per Definition 28.

If $\alpha = \beta$ then $\vec{X} \leq_{lex} \vec{Y}$ is disentailed by Theorem 8 and thus `Initialise` terminates with failure (line 9). Otherwise, in line 10, the filtering algorithm `LexLeq(`α`)` is called to ensure GAC($\vec{X} \leq_{lex} \vec{Y}$). QED.

Theorem 13 *Whenever $min(X_i)$ or $max(Y_i)$ of some i in $[\alpha, min\{n, \beta\})$ changes,* `LexLeq`(i) *either establishes failure if $\vec{X} \leq_{lex} \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC($\vec{X} \leq_{lex} \vec{Y}$).*

Proof: `LexLeq` may update the values of α and β. We first give the pre- and post-conditions of `UpdateAlpha` and `UpdateBeta`, and show that these two procedures establish their post-conditions given the necessary pre-conditions.

`UpdateAlpha(`i`)` must be invoked only when $X_\alpha \doteq Y_\alpha$ which violates the definition of α, and i is the most significant candidate index that can be assigned to α. If there is any index satisfying Definition 27 then α must be updated to that index. With the new value of α, `UpdateAlpha(`i`)` must either establish failure if $\vec{X} \leq_{lex} \vec{Y}$ is disentailed, or ensure GAC($\vec{X} \leq_{lex} \vec{Y}$). Assuming that the pre-condition is satisfied, in lines 3 and 4.1 of `UpdateAlpha(`i`)`, the vectors are traversed until the most significant index k where $\neg(X_k \doteq Y_k)$ is found. If such an index does not exist (the vectors are ground and equal) then we have GAC($\vec{X} \leq_{lex} \vec{Y}$) and the algorithm returns (line 2). Otherwise, α is set to k (line 3.1) as per Definition 27. `LexLeq(`α`)` is then called to ensure GAC($\vec{X} \leq_{lex} \vec{Y}$). During this traversal, if β is encountered then $\vec{X} \leq_{lex} \vec{Y}$ is disentailed by Theorem 8 and `UpdateAlpha(`i`)` terminates with failure (line 1). Hence, `UpdateAlpha(`i`)` establishes its post-condition.

`UpdateBeta(`i`)` must be invoked only when the definition of β is violated, and $i+1$ is the least significant candidate index that can be assigned to β. The pointer β must be set to most significant index satisfying Definition 28. With this new value of β, `UpdateBeta(`i`)` must either establish failure if $\vec{X} \leq_{lex} \vec{Y}$ is disentailed, or ensure GAC($\vec{X} \leq_{lex} \vec{Y}$). Assuming that the pre-condition is satisfied, in lines 2 and 3.1 of `UpdateBeta(`i`)`, the vectors are traversed until the most significant index k where $min(X_k) < max(Y_k)$ is found. The pointer β is set to $k+1$ (line 2.1) as per Definition 28. `LexLeq(`k`)` is then called to ensure GAC($\vec{X} \leq_{lex} \vec{Y}$). During this traversal, if α is encountered then $\vec{X} \leq_{lex} \vec{Y}$ is disentailed by Theorem 8 and thus `UpdateBeta(`i`)` terminates with failure (line 1). Hence, `UpdateBeta`(i) establishes its post-condition.

We now analyse `LexLeq` block-by-block.

Block 1: If $i = \alpha \ \wedge \ i+1 = \beta$ then `AC(`$X_i < Y_i$`)` is executed, which maintains arc-consistency on $X_i < Y_i$. This ensures GAC($\vec{X} \leq_{lex} \vec{Y}$) by Theorem 9.

Block 2: If $i = \alpha \ \wedge \ i+1 < \beta$ then `AC(`$X_i \leq Y_i$`)` is executed, which maintains arc-consistency on $X_i \leq Y_i$. This ensures GAC($\vec{X} \leq_{lex} \vec{Y}$) by Theorem 9. Afterwards, if $X_i \doteq Y_i$ then `UpdateAlpha(`$i+1$`)` is called satisfying its precondition. The post-condition of `UpdateAlpha(`$i+1$`)` is either failure or GAC($\vec{X} \leq_{lex} \vec{Y}$).

Block 3: If $\alpha < i < \beta$ and $(i = \beta - 1 \ \wedge \ min(X_i) = max(Y_i)\ ) \vee min(X_i) > max(Y_i)$ then `UpdateBeta(`$i-1$`)` is called satisfying its pre-condition. The post-condition of `UpdateBeta(`$i-1$`)` is either failure or GAC($\vec{X} \leq_{lex} \vec{Y}$).

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

5.4 Extensions

In this section, we consider a number of interesting extensions to the filtering algorithm of the lexicographic ordering constraint. Section 5.4.1 shows how we can obtain a filtering algorithm for the strict lexicographic ordering constraint. Section 5.4.2 presents how we can catch entailment. Section 5.4.3 discusses that propagating vector disequality and vector equality constraints using (strict) lexicographic ordering constraints hinders constraint propagation, shows how we can obtain a filtering algorithm for the vector disequality constraint, and discusses how we can propagate the vector equality constraint. Finally, Section 5.4.4 describes how we can deal with a pair of vectors of different length.

5.4.1 Strict Lexicographic Ordering Constraint

With very little effort, `LexLeq` can be adapted to obtain a filtering algorithm, `LexLess`, which either detects the disentailment of $\vec{X} <_{lex} \vec{Y}$ or prunes inconsistent values so as to achieve GAC on $\vec{X} <_{lex} \vec{Y}$. The reason of the similarity is that, as soon as β is assigned a value other than ∞, `LexLeq` enforces strict lexicographic ordering on the vectors.

Before showing how we modify `LexLeq`, we give the necessary theoretical background. We define the pointer α between two vectors $\vec{X}$ and $\vec{Y}$ as in Definition 27. Also, the pointer β again points to the index starting from which the vectors are ordered the wrong way around no matter what values are assigned to the future variables. However, we redefine β, because β should now point to the index such that $\vec{X}_{\beta \to n-1} <_{lex} \vec{Y}_{\beta \to n-1}$ is disentailed.

Definition 29 *Given $\vec{X}$ and $\vec{Y}$, the pointer β is set either to the index in $[\alpha, n)$ such that:*

$$\exists k \;\; \beta \leq k < n \,.\, (min(X_k) > max(Y_k) \land \forall i \;\; \beta \leq i < k \,.\, min(X_i) = max(Y_i) \;) \;\; \lor$$

$$\forall i \;\; \beta \leq i < n \,.\, min(X_i) = max(Y_i)$$

where:

$$min(X_{\beta-1}) \neq max(Y_{\beta-1}) \land (\forall i \;\; \alpha \leq i < \beta \,.\, min(X_i) \leq max(Y_i) \;)$$

or (if this is not the case) to n.

Informally, the pointer β points to the most significant index in $[\alpha, n)$ such that $\vec{X}_{\beta \to n-1} \geq_{lex} \vec{Y}_{\beta \to n-1}$ is *true*. If this is not the case then β points to n. It is important to set β to n when such an index does not exist because by this way equality between the vectors is not allowed. Consider two vectors $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1} \rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1} \rangle$ with $X_i \doteq Y_i$ for all $0 \leq i < n-1$ and $\mathcal{D}(X_{n-1}) = \mathcal{D}(Y_{n-1}) = \{0, 1\}$. The pointer α is set to $n-1$. Setting β to ∞ would allow equality between the vectors, which is wrong.

We can now make use of the pointers to detect disentailment as well as prune inconsistent values.

Theorem 14 *Given $\beta = \alpha$, $\vec{X} <_{lex} \vec{Y}$ is disentailed.*

Proof: By Definition 27, $\vec{X}_{0 \to \alpha-1} = \vec{Y}_{0 \to \alpha-1}$ is *true*, and by Definition 29 $\vec{X}_{\alpha \to n-1} \geq_{lex} \vec{Y}_{\alpha \to n-1}$ is *true*, whatever the remaining assignments are. Hence, $\vec{X} \geq_{lex} \vec{Y}$ is *true*. QED.

It is again not hard to show that given $\beta > \alpha$, generalised arc-inconsistent values can exist only in the interval $[\alpha, \beta)$. We can again show that generalised arc-inconsistent values exist only at index α indeed.

Theorem 15 *Given $\beta > \alpha$, GAC($\vec{X} <_{lex} \vec{Y}$) iff:*

1. $\beta = \alpha + 1 \rightarrow AC(X_\alpha < Y_\alpha)$
2. $\beta > \alpha + 1 \rightarrow AC(X_\alpha \leq Y_\alpha)$

($\Rightarrow$) Similar to the proof of Theorem 9.
($\Leftarrow$) All variables above α are ground and pairwise equal. The assignment $X_\alpha \gtrdot Y_\alpha$ cannot be extended to a consistent assignment to satisfy $\vec{X} <_{lex} \vec{Y}$, because in this case $\vec{X}$ is already greater than $\vec{Y}$ at position α. The reverse holds for the assignment $X_\alpha \lessdot Y_\alpha$. In this case, any future assignment satisfies $\vec{X} <_{lex} \vec{Y}$ as $\vec{X}$ is already less than $\vec{Y}$ at position α. Whether assigning the same values to X_α and Y_α is consistent or not depends on the index β points to.

If $\beta = \alpha + 1$ then we consider two cases, namely when $\beta < n$ and when $\beta = n$. In the first case:

$$\exists k \ \alpha < k < n \,.\, (min(X_k) > max(Y_k) \wedge \forall i \ \alpha < i < k \,.\, min(X_i) = max(Y_i) \ ) \ \vee$$
$$\forall i \ \alpha < i < n \,.\, min(X_i) = max(Y_i)$$

This means that either there is an index k between α and n such that $X_k > Y_k$ is *true* and additionally for all $\alpha < i < k$ we have $X_i \geq Y_i$ is *true*, or for all $\alpha < i < n$ we have $X_i \geq Y_i$ is *true*. The assignment $X_\alpha \doteq Y_\alpha$ cannot therefore be extended to a consistent assignment satisfying $\vec{X} <_{lex} \vec{Y}$. In the second case, we have $\alpha = n - 1$ and $\beta = n$. The only position to strictly order the vectors is index $n - 1$. Equality at this position only makes the vectors equal. Either case proves that AC($X_\alpha < Y_\alpha$) implies GAC($\vec{X} \leq_{lex} \vec{Y}$) when $\beta = \alpha + 1$.

If $\beta > \alpha + 1$ then:

$$min(X_{\beta-1}) \neq max(Y_{\beta-1}) \wedge (\forall i \ \alpha \leq i < \beta \,.\, min(X_i) \leq max(Y_i) \ )$$

That is, we can find values in $\mathcal{D}(X_i)$ and $\mathcal{D}(Y_i)$ for all $\alpha < i < \beta - 1$ to satisfy $X_i \leq Y_i$, and in $\mathcal{D}(X_{\beta-1})$ and $\mathcal{D}(Y_{\beta-1})$ to satisfy $X_{\beta-1} < Y_{\beta-1}$. This shows that the assignment $X_\alpha \doteq Y_\alpha$ can be extended to a consistent assignment satisfying $\vec{X} <_{lex} \vec{Y}$, and proves that AC($X_\alpha \leq Y_\alpha$) implies GAC($\vec{X} \leq_{lex} \vec{Y}$) when $\beta > \alpha + 1$. QED.

How do we modify `LexLeq` to obtain the filtering algorithm `LexLess` for propagating the strict lexicographic ordering constraint? Theorems 14-15 are identical to Theorems 8-9. Even though the definition of β is now slightly different, we have:

$$min(X_{\beta-1}) \neq max(Y_{\beta-1}) \wedge (\forall i \ \alpha \leq i < \beta \,.\, min(X_i) \leq max(Y_i) \ )$$

in both Definitions 28 and 29. This suggests that we do not need to change `LexLeq`, including the part where we update β. On the other hand, small modifications to `Initialise` are needed. Algorithm 5 shows how we modify Algorithm 1.

We fail if the vectors are initially ground and equal. In fact, similar modification is necessary also for `UpdateAlpha`. Note that the vectors can later become ground and equal only if the event queue contains a series of domain prunings which must be dealt with simultaneously. Otherwise, after initialisation, the lexicographic ordering constraint is either disentailed or GAC. In the second case, every value is supported. After every single assignment, by maintaining the generalised arc-consistent state, this property persists.

Moreover, we cancel line 7 for two reasons: first, we want to initialise β to n if there is no index starting from which the vectors are ordered the wrong way around whatever the remaining assignments are. Second, if there is no index k such that $min(X_k) > max(Y_k)$ but rather there is an index i such that for all $i \leq j < n$ we have $min(X_j) = max(Y_j)$ and i is the most significant such index, then we want to initialise β to i.

Algorithm 5: `Initialise`

```
Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
Result : $\alpha$ and $\beta$ are initialised, GAC($\vec{X} \leq_{lex} \vec{Y}$)
    ⋮
    if $i = n$ then fail;
    ⋮
    ~~if $i = n$ then $\beta := \infty$;~~
    ⋮
```

Procedure `UpdateAlpha(`i`)`

```
    ⋮
    if $i = n$ then fail;
    ⋮
```

5.4.2 Entailment

Even though `LexLeq` is a complete and correct filtering algorithm, it does not detect entailment. Incorporating entailment changes neither the worst-case nor the amortised complexity of the algorithm but is very useful for avoiding unnecessary work.

A constraint is said to be *entailed* when the constraint is *true*. If a constraint is entailed, then it is not necessary to execute the filtering algorithm: we can simply return without having to do any pruning. For this purpose, we introduce a Boolean flag called *entailed* which indicates whether $\vec{X} \leq_{lex} \vec{Y}$ is entailed. More formally:

Definition 30 *Given $\vec{X}$ and $\vec{Y}$, the flag* entailed *is set to true iff $\vec{X} \leq_{lex} \vec{Y}$ is true.*

It is not hard to show when $\vec{X} \leq_{lex} \vec{Y}$ is *true*.

Theorem 16 *$\vec{X} \leq_{lex} \vec{Y}$ is true iff* `ceiling`$(\vec{X}) \leq_{lex}$ `floor`$(\vec{Y})$.

Proof: ($\Rightarrow$) Since $\vec{X} \leq_{lex} \vec{Y}$ is *true* any combination of assignments, including $\vec{X} \leftarrow$ `ceiling`$(\vec{X})$ and $\vec{Y} \leftarrow$ `floor`$(\vec{Y})$, satisfies $\vec{X} \leq_{lex} \vec{Y}$. Hence, `ceiling`$(\vec{X}) \leq_{lex}$ `floor`$(\vec{Y})$.

($\Leftarrow$) Any $\vec{x} \in \vec{X}$ is lexicographically less than or equal to any $\vec{y} \in \vec{Y}$. Hence, $\vec{X} \leq_{lex} \vec{Y}$ is *true*. QED.

To know when we have `ceiling`$(\vec{X}) \leq_{lex}$ `floor`$(\vec{Y})$, we introduce a third pointer, called γ, whose purpose is dual to that of β.

Definition 31 *Given $\vec{X}$ and $\vec{Y}$, the pointer γ is set either to the index in $[\alpha, n)$ such that:*

$$\exists k \; \gamma \leq k < n \, . \, (max(X_k) < min(Y_k) \wedge \forall i \; \gamma \leq i < k \, . \, max(X_i) = min(Y_i) \;) \quad \vee$$

$$\forall i \; \gamma \leq i < n \, . \, max(X_i) = min(Y_i)$$

where:

$$max(X_{\gamma-1}) \neq min(Y_{\gamma-1}) \wedge (\forall i \; \alpha \leq i < \gamma \, . \, max(X_i) \geq min(Y_i) \;)$$

or (if this is not the case) to n.

Algorithm 7: `Initialise`

```
    Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
    Result : $\alpha$, $\beta$, entailed, and $\gamma$ are initialised, GAC($\vec{X} \leq_{lex} \vec{Y}$)
0       entailed := false;
        ⋮
3       if $i = n$ then
3.1         entailed := true;
3.2         return;
        end
        ⋮
10      $\gamma := -1$;
11      while $i \neq n \ \wedge\ max(X_i) \geq min(Y_i)$ do
11.1        if $max(X_i) = min(Y_i)$ then
11.1.1          if $\gamma = -1$ then $\gamma := i$;
            else
11.2.1          $\gamma := -1$;
            end
11.3        $i := i + 1$;
        end
12      if $\gamma = -1$ then $\gamma := i$;
13      if $\alpha = \gamma$ then
13.1        entailed := true;
13.2        return;
        end
14      LexLeq($\alpha$);
```

Informally, γ points to the most significant index in $[\alpha, n)$ such that $\vec{X}_{\gamma\to n-1} \leq_{lex} \vec{Y}_{\gamma\to n-1}$ is *true*. That is, starting from index γ, the vectors are ordered lexicographically no matter what values are assigned to the future variables. If this is not the case then γ points to n.

The purpose of the pointers β and γ are dual. The former points to the index below which the vectors are ordered the other way around whatever the remaining assignments are. We make use of it to prune inconsistent values as well as detect disentailment. The latter points to the index starting from which the vectors are ordered lexicographically independent of the future assignments. We can therefore use it to detect entailment.

Theorem 17 *$\vec{X} \leq_{lex} \vec{Y}$ is entailed iff $\gamma = \alpha$.*

Proof: ($\Rightarrow$) As $\vec{X}_{0\to n-1} \leq_{lex} \vec{Y}_{0\to n-1}$ is *true*, we have $\gamma = \alpha$.

($\Leftarrow$) By Definition 27, $\vec{X}_{0\to\alpha-1} = \vec{Y}_{0\to\alpha-1}$ is *true*, and by Definition 31 $\vec{X}_{\alpha\to n-1} \leq_{lex} \vec{Y}_{\alpha\to n-1}$ is *true*, whatever the remaining assignments are. Hence, $\vec{X} \leq_{lex} \vec{Y}$ is *true*. QED.

The lexicographic ordering constraint is entailed also when the vectors are ground and equal. In this case none of the variables are left uninstantiated and the constraint cannot wake up anymore. We nevertheless set the flag *entailed* to *true* for completeness.

We now need to integrate the entailment knowledge into the filtering algorithm of the lexicographic ordering constraint. First, we initialise the values of *entailed* and γ. In Algorithm 7, we show how we modify Algorithm 1 for this purpose. We add line 0 to initialise the flag *entailed* to $false$. We modify line 3 by setting *entailed* to *true* before returning. In fact, we do a similar modification in `UpdateAlpha`.

Procedure `UpdateAlpha(`i`)`

```
      ⋮
2     if $i = n$ then
2.1       $entailed := true$;
2.2       return;
      end
      ⋮
```

We replace line 10 of the original algorithm `Initialise` with lines 10-14. Now, lines 10-12 initialise γ in a dual manner of the initialisation of β. In case $\gamma = \alpha$, we set *entailed* to *true* and return (lines 13-13.2) by Theorem 17. Otherwise, we call `LexLeq(`α`)` for an initial pruning at α in case needed.

`LexLeq(`i`)` is called whenever $min(X_i)$ or $max(Y_i)$ of some i in $[\alpha, min\{n, \beta\})$ changes. The position of i with respect to that of α determines two different response. First, if $i = \alpha$ then we maintain the generalised arc-consistent state. After this, if we have $i = \alpha = \gamma$ then $\vec{X} \leq_{lex} \vec{Y}$ is entailed by Theorem 17. Second, if $i > \alpha$ then we do not have to maintain the generalised arc-consistent state. However, we might have to update the pointer γ, just like we might have to update the pointer β when $i > \alpha$. In Algorithm 10, we show how we modify Algorithm 2 in the presence of γ.

We add lines 4-4.1.1 for updating γ in case i is after α and i is a more significant position for γ. The condition for updating γ is derived from Definition 31: at i we either have $max(X_i) < min(Y_i)$, or i is $\gamma - 1$ and $max(X_i) = min(Y_i)$. The pointer γ is updated by calling `UpdateGamma(`$i - 1$`)`.

Procedure `UpdateGamma(`i`)`

```
1     if $i + 1 = \alpha$ then
1.1       $entailed := true$;
1.2       return;
      end
2     if $max(X_i) > min(Y_i)$ then $\gamma := i + 1$;
3     else if $max(X_i) = min(Y_i)$ then UpdateGamma($i - 1$);
```

In lines 2 and 3 of `UpdateGamma`, the vectors are traversed until the index k where $max(X_k) > min(Y_k)$ is found. The pointer γ is set to $k+1$ (line 2). During this traversal, if α is met then `UpdateGamma` sets *entailed* to *true* and then returns (lines 1.1 and 1.2).

We modify `LexLeq` further to catch entailment in case initially $i = \alpha$ and we get $i = \alpha = \gamma$ after maintaining the generalised arc-consistent state. In other words, γ moves all the way to α after executing the line 1.1 or 2.1. There are two cases. First, if initially $i = \alpha \ \wedge \ i + 1 = \beta$ then i cannot be $\gamma - 1$ because at $i + 1$ we have β, and by definition β cannot point to the same index as γ. We get $i = \alpha = \gamma$ if we have $max(X_i) < min(Y_i)$ after `AC(`$X_i < Y_i$`)`. Second, if initially $i = \alpha \ \wedge \ i+1 < \beta$ then we get $i = \alpha = \gamma$ provided that initially i was $\gamma - 1$ and we have $max(X_i) = min(Y_i)$ after `AC(`$X_i \leq Y_i$`)`. Another alternative is that after `AC(`$X_i \leq Y_i$`)` we have $max(X_i) < min(Y_i)$. In any case, whenever we get $i = \alpha = \gamma$ (lines 1.2 and 2.2, respectively), we set *entailed* to *true* (lines 1.2.1 and 2.2.1 resp.), and return from the algorithm (lines 1.2.2 and 2.2.2 resp.).

Algorithm 10: `LexLeq`(i)

```
      Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, Integer $i$
      Result : GAC($\vec{X} \leq_{lex} \vec{Y}$)
0      if entailed then return;
1      if $i = \alpha \ \wedge \ i+1 = \beta$ then
1.1        AC($X_i < Y_i$);
1.2        if $max(X_i) < min(Y_i)$ then
1.2.1          entailed := true;
1.2.2          return;
           end
       end
2      if $i = \alpha \ \wedge \ i+1 < \beta$ then
2.1        AC($X_i \leq Y_i$);
2.2        if $(i = \gamma - 1 \ \wedge \ max(X_i) = min(Y_i)) \ \vee \ max(X_i) < min(Y_i)$ then
2.2.1          entailed := true;
2.2.2          return;
           end
2.3        if $X_i \doteq Y_i$ then UpdateAlpha($i+1$);
       end
       ⋮
4      if $\alpha < i < \gamma$ then
4.1        if $(i = \gamma - 1 \ \wedge \ max(X_i) = min(Y_i)) \ \vee \ max(X_i) < min(Y_i)$ then
4.1.1          UpdateGamma($i-1$);
           end
       end
```

Finally, we add line 0, where we return if the constraint was previously entailed. If the constraint is entailed due to the latest modification at i, we catch this after `AC`. To avoid running `AC` in such a case, we can repeat lines 1.2-1.2.2 and 2.2-2.2.2 before `AC(`$X_i \leq Y_i$`)` and `AC(`$X_i < Y_i$`)`, respectively.

Initialisation and update of γ require, in the worst case, a complete scan of the vectors. Hence, the worst-case and amortised complexity of `LexLeq` remain $O(nb)$ and $O(b)$, respectively.

5.4.3 Vector Disequality and Equality

Two vectors of integers $\vec{x} = \langle x_0, \ldots, x_{n-1}\rangle$ and $\vec{y} = \langle y_0, \ldots, y_{n-1}\rangle$ are different $\vec{x} \neq \vec{y}$ iff $\vec{x} <_{lex} \vec{y}$ or $\vec{y} <_{lex} \vec{x}$. Therefore, we can decompose a vector disequality constraint $\vec{X} \neq \vec{Y}$ by insisting that one of the vectors must be lexicographically less than the other:

$$\vec{X} <_{lex} \vec{Y} \ \vee \ \vec{Y} <_{lex} \vec{X}$$

Most solvers will delay such a disjunction until one of the disjuncts becomes $false$ (see Chapter 5.5.2). At this point, we can enforce GAC on the other disjunct. As the following theorem shows, such a decomposition hinders constraint propagation.

Theorem 18 *GAC($\vec{X} \neq \vec{Y}$) is strictly stronger than $\vec{X} <_{lex} \vec{Y} \ \vee \ \vec{Y} <_{lex} \vec{X}$, assuming that GAC is enforced on the delayed disjunctive constraint.*

Proof: GAC($\vec{X} \neq \vec{Y}$) is as strong as its decomposition. To show strictness, consider $\vec{X} = \langle\{0,1,2\}\rangle$ and $\vec{Y} = \langle\{1\}\rangle$ where 1 in $\mathcal{D}(X_0)$ cannot be extended to a consistent assignment and therefore $\vec{X} \neq \vec{Y}$ is not GAC. Since neither of $\vec{X} <_{lex} \vec{Y}$ and $\vec{Y} <_{lex} \vec{X}$ is $false$ yet, the decomposition does not post any constraint, leaving the vectors unchanged. QED.

Even though we cannot directly use strict lexicographic ordering constraint to propagate the disequality constraint, we can slightly modify `LexLess` to obtain a filtering algorithm for the disequality constraint. In this new algorithm, the pointer α is defined as in Definition 27; but β now points to the most significant index in $[\alpha, n)$ starting from which the vectors are ground and equal.

Definition 32 *Given $\vec{X}$ and $\vec{Y}$, β is set either to the index in $[\alpha, n)$ such that:*

$$(\forall i \;\beta \leq i < n \,.\;\; X_i \doteq Y_i) \;\wedge\; \neg(X_{\beta-1} \doteq Y_{\beta-1})$$

or (if this is not the case) to n.

As in the lexicographic ordering case, the constraint is disentailed if $\beta = \alpha$.

Theorem 19 *Given $\beta = \alpha$, $\vec{X} \neq \vec{Y}$ is disentailed.*

Proof: If $\beta = \alpha$ then the vectors are ground and equal, hence cannot be made different. QED.

Now we give the necessary condition for GAC on $\vec{X} \neq \vec{Y}$.

Theorem 20 *Given $\beta > \alpha$, GAC($\vec{X} \neq \vec{Y}$) iff: AC($X_\alpha \neq Y_\alpha$) when $\beta = \alpha + 1$.*

Proof: ($\Rightarrow$) Assume $\vec{X} \neq \vec{Y}$ is GAC but $X_\alpha \neq Y_\alpha$ is not AC when $\beta = \alpha + 1$. Then there is a value in $\mathcal{D}(X_\alpha)$ or $\mathcal{D}(Y_\alpha)$ with which we cannot achieve $X_\alpha \neq Y_\alpha$ and the vectors can be made different at only position α. This contradicts that the disequality constraint is GAC.

($\Leftarrow$) If $\beta = \alpha + 1$ then there is only one index, α, where the vectors can take different values and hence differ. So AC($X_\alpha \neq Y_\alpha$) must be maintained for GAC on the disequality constraint. Otherwise, if $\beta > \alpha + 1$ then the assignment $X_\alpha \lessdot Y_\alpha$ or $X_\alpha \gtrdot Y_\alpha$ can always be extended to a consistent assignment as in either case the vectors already take a different value at α. Also, $X_\alpha \doteq Y_\alpha$ can be extended to a consistent assignment no matter what the domains of the variables between α and β are. Even if $X_i \doteq Y_i$ for all $\alpha < i < \beta - 1$, at $\beta - 1$ we have $\neg(X_{\beta-1} \doteq Y_{\beta-1})$ since β points to the most significant index. QED.

We can now easily modify `LexLess` based on the new definition of β and Theorem 20 so as to obtain a filtering algorithm `LexDiff` which either proves that $\vec{X} \neq \vec{Y}$ is disentailed or establishes GAC on $\vec{X} \neq \vec{Y}$.

Two vectors of integers $\vec{x} = \langle x_0, \ldots, x_{n-1}\rangle$ and $\vec{y} = \langle y_0, \ldots, y_{n-1}\rangle$ are equivalent $\vec{x} = \vec{y}$ iff $\vec{x} \leq_{lex} \vec{y}$ and $\vec{y} \leq_{lex} \vec{x}$. Therefore, we can decompose a vector equality constraint $\vec{X} = \vec{Y}$ by insisting that each of the vectors must be lexicographically less than or equal to the other:

$$\vec{X} \leq_{lex} \vec{Y} \;\wedge\; \vec{Y} \leq_{lex} \vec{X}$$

In this way, both of the constraints in the conjunction are posted (see Chapter 5.5.2). As the following theorem shows, such a decomposition hinders constraint propagation.

Theorem 21 *GAC($\vec{X} = \vec{Y}$) is strictly stronger than GAC($\vec{X} \leq_{lex} \vec{Y}$) and GAC($\vec{Y} \leq_{lex} \vec{X}$).*

Proof: GAC($\vec{X} = \vec{Y}$) is as strong as its decomposition. To show strictness, consider $\vec{X} = \langle\{0,1,2\}\rangle$ and $\vec{Y} = \langle\{0,2\}\rangle$ where 1 in $\mathcal{D}(X_0)$ cannot be extended to a consistent assignment and therefore $\vec{X} = \vec{Y}$ is not GAC. The constraints $\vec{X} \leq_{lex} \vec{Y}$ and $\vec{Y} \leq_{lex} \vec{X}$ are both GAC. The vectors are thus left unchanged. QED.

We cannot therefore make use of lexicographic ordering constraint for enforcing GAC on the vector equality constraint. On the other hand, another way of decomposing this constraint eliminates the need of designing a new filtering algorithm. Two vectors $\vec{x} = \langle x_0, \ldots, x_{n-1}\rangle$ and $\vec{y} = \langle y_0, \ldots, y_{n-1}\rangle$ are equivalent $\vec{x} = \vec{y}$ iff $x_i = y_i$ for all $0 \leq i < n$. We can therefore decompose a vector equality constraint $\vec{X} = \vec{Y}$ by insisting that:

$$\forall i \; 0 \leq i < n \, . \;\; X_i = Y_i$$

As the following theorem shows, GAC on the vector equality constraint does the same pruning as AC on this decomposition.

Theorem 22 *GAC($\vec{X} = \vec{Y}$) is equivalent to AC($X_i = Y_i$) for all $0 \leq i < n$.*

Proof: GAC($\vec{X} = \vec{Y}$) is as strong as its decomposition. To show the reverse, suppose that each of the constraints posted is AC but $\vec{X} = \vec{Y}$ is not GAC. Then there is an inconsistent value in the domain of a variable. In any assignment $\vec{x}$ and $\vec{y}$ to $\vec{X}$ and $\vec{Y}$ with the variable assigned this inconsistent value, there is some index α in $[0, n)$ such that for all $0 \leq i < \alpha$ we have $x_i = y_i$ and $x_\alpha \neq y_\alpha$. But this contradicts that AC($X_\alpha = Y_\alpha$). QED.

5.4.4 Vectors of Different Length

In Definition 13, we defined strict lexicographic ordering between two vectors of equal length, and in Theorem 15 we stated the necessary conditions for GAC on strict lexicographic ordering constraint posted on a pair of vectors of equal length. It is, however, straightforward to generalise the definition and the theorem for two vectors of *any*, not necessarily equal, length.

Definition 33 *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 $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1}\rangle$ with $m, n > 0$ and $k = min\{m, n\}$ holds iff either of the following conditions hold:*

1. $k = m < n \;\wedge\; \langle x_0, x_1, \ldots, x_{k-1}\rangle \leq \langle y_0, y_1, \ldots, y_{k-1}\rangle$
2. $\langle x_0, x_1, \ldots, x_{k-1}\rangle <_{lex} \langle y_0, y_1, \ldots, y_{k-1}\rangle$

In other words, we chop off 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 longer than or equal to $\vec{y}$ 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 $\langle X_0, X_1, \ldots, X_{m-1}\rangle <_{lex} \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$ is either GAC on lexicographic ordering constraint or GAC on strict lexicographic ordering constraint.

Theorem 23 *GAC($\langle X_0, X_1, \ldots, X_{m-1}\rangle <_{lex} \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$) iff:*

1. $m < n \rightarrow GAC(\langle X_0, X_1, \ldots, X_{m-1}\rangle \leq \langle Y_0, Y_1, \ldots, Y_{m-1}\rangle)$
2. $m \geq n \rightarrow GAC(\langle X_0, X_1, \ldots, X_{n-1}\rangle <_{lex} \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle)$

We can now easily generalise the filtering algorithm `LexLess` based on this theorem. If $m < n$ then we just consider the first m variables of $\vec{Y}$ and maintain GAC on $\langle X_0, X_1, \ldots, X_{m-1}\rangle \leq_{lex} \langle Y_0, Y_1, \ldots, Y_{m-1}\rangle$. If $m \geq n$ then we just consider the first n variables of $\vec{X}$ and run `LexLess` on $\langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$.

5.5 Alternative Approaches

There are at least two other ways of posting lexicographic ordering constraints: by posting arithmetic inequality constraints, or by decomposing them into smaller constraints. In this section, we study these alternative approaches in detail and argue why our filtering algorithms can be preferable to them.

5.5.1 Arithmetic Constraint

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

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

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:

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

Maintaining BC on such arithmetic constraints does the same pruning as GAC on the original lexicographic ordering constraints.

Theorem 24 *GAC($\vec{X} \leq_{lex} \vec{Y}$) and GAC($\vec{X} <_{lex} \vec{Y}$) are equivalent to BC on the corresponding arithmetic constraints.*

Proof: We just consider GAC($\vec{X} \leq_{lex} \vec{Y}$) as the proof for GAC($\vec{X} <_{lex} \vec{Y}$) is entirely analogous. As $\vec{X} \leq_{lex} \vec{Y}$ and the corresponding arithmetic constraint are logically equivalent, BC($\vec{X} \leq_{lex} \vec{Y}$) and BC on the arithmetic constraint are equivalent. By Theorem 9, BC($\vec{X} \leq_{lex} \vec{Y}$) is equivalent to GAC($\vec{X} \leq_{lex} \vec{Y}$). QED.

Maintaining BC on such arithmetic constraints can naively be achieved in $O(ndc)$ where n is the length of the vectors, d is the domain size, and c is the time required to check that a particular (upper or lower) bound of a variable is BC. At best, c is a constant time operation. However, when n and d get large, d^{n-1} will be much more than the word size of the computer and computing BC will be significantly more expensive. Hence, this method is only feasible when the vectors and domain sizes are small.

5.5.2 Decomposition

Arbitrary arithmetic constraints can be combined by traditional logical connectives ($\wedge, \vee, \rightarrow$ $, \leftrightarrow$, and $\neg$) to build complex logical constraint expressions. We can therefore compose simple arithmetic constraints between the variables of $\vec{X}$ and $\vec{Y}$ by using logical connectives so as to obtain the lexicographic ordering constraint between $\vec{X}$ and $\vec{Y}$. We refer to such a logical constraint as a decomposition of the lexicographic ordering constraint.

There are at least two ways of decomposing a lexicographic ordering constraint $\vec{X} \leq_{lex} \vec{Y}$. The first decomposition, which we call $\wedge$ decomposition, is a conjunction of $n-1$ constraints:

$$\begin{array}{rl} X_0 \leq Y_0 & \wedge \\ X_0 = Y_0 \rightarrow X_1 \leq Y_1 & \wedge \\ X_0 = Y_0 \ \wedge \ X_1 = Y_1 \rightarrow X_2 \leq Y_2 & \wedge \\ \vdots & \\ X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{n-2} = Y_{n-2} \rightarrow X_{n-1} \leq Y_{n-1} & \end{array}$$

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 $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \ldots \wedge X_{n-2} = Y_{n-2} \rightarrow X_{n-1} < Y_{n-1}$.

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

$$\begin{array}{rl} X_0 < Y_0 & \vee \\ X_0 = Y_0 \ \wedge \ X_1 < Y_1 & \vee \\ X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ X_2 < Y_2 & \vee \\ \vdots & \\ X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{n-2} = Y_{n-2} \ \wedge \ X_{n-1} \leq Y_{n-1} & \end{array}$$

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 $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{n-2} = Y_{n-2} \ \wedge \ X_{n-1} < Y_{n-1}$.

The declarative meaning and the operational semantics of the logical constraints that we here consider are:

- $C_1 \wedge C_2$: Both constraint expressions C_1 and C_2 are *true*. Hence, both C_1 and C_2 are imposed.
- $C_1 \vee C_2$: At least one of the constraint expressions C_1 and C_2 is *true*. Hence, if one of C_1 or C_2 becomes $false$, the other constraint is imposed.
- $C_1 \rightarrow C_2$: The constraint expression C_1 implies the constraint expression C_2. Hence, if C_1 becomes *true* then C_2 is imposed. If C_2 becomes $false$ then the negation of C_1 is imposed.

Assuming that AC is enforced on a binary constraint whenever it is imposed, the two decompositions are inferior to maintaining GAC on $\vec{X} \leq_{lex} \vec{Y}$ and they are themselves incomparable with respect to the inconsistent values being removed.

Theorem 25 *$GAC(\vec{X} \leq_{lex} \vec{Y})$ and $GAC(\vec{X} <_{lex} \vec{Y})$ are strictly stronger than the corresponding $\wedge$ and $\vee$ decompositions. Moreover, the two decompositions are incomparable.*

Proof: We only consider GAC($\vec{X} \leq_{lex} \vec{Y}$) but the proof works also for GAC($\vec{X} <_{lex} \vec{Y}$). Since $\vec{X} \leq_{lex} \vec{Y}$ is GAC, every value has a support. The vectors $\vec{x}$ and $\vec{y}$ supporting a value are lexicographically ordered ($\vec{x} \leq_{lex} \vec{y}$). By Definition 14, either $\vec{x} = \vec{y}$ or there is an index k in $[0, n)$ such that $x_k < y_k$ and for all $0 \leq i < k$ we have $x_i = y_i$. 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, GAC($\vec{X} \leq_{lex} \vec{Y}$) is as strong as any of its decomposition.

Consider $\vec{X} = \langle\{0, 1\}, \{1\}\rangle$ and $\vec{Y} = \langle\{0, 1\}, \{0\}\rangle$ where $\vec{X} \leq_{lex} \vec{Y}$ is not GAC. The $\wedge$ decomposition imposes both of $X_0 \leq Y_0$ and $X_0 = Y_0 \rightarrow X_1 \leq Y_1$. We have AC($X_0 \leq Y_0$). Since $X_1 \leq Y_1$ is $false$, $X_0 \neq Y_0$ is enforced. We have AC($X_0 \neq Y_0$) so no pruning is possible. The $\vee$ decomposition, however, imposes $X_0 < Y_0$ because $X_0 = Y_0 \ \wedge \ X_1 \leq Y_1$ is $false$. 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 $\vec{X} \leq_{lex} \vec{Y}$ is not GAC. The $\wedge$ decomposition removes 2 from $\mathcal{D}(X_0)$ by AC($X_0 \leq Y_0$). The $\vee$ decomposition, however, leaves the vectors unchanged since neither $X_0 < Y_0$ nor $X_0 = Y_0 \wedge X_1 \leq Y1$ is $false$ yet.

Due to the fact that the $\wedge$ decomposition does a pruning that is not done by the $\vee$ decomposition and vice versa, the two decompositions are incomparable. QED.

The two decompositions together behave similarly to the filtering algorithm of the lexicographic ordering constraint: they either prove that $\vec{X} \leq_{lex} \vec{Y}$ is disentailed or establish GAC($\vec{X} \leq_{lex} \vec{Y}$). However, this requires posting and propagating too many constraints. Considering that posting a single global constraint requires propagating only one constraint, we expect our approach to be the most efficient way of solving the lexicographic ordering constraint. Indeed, our experimental results in Section 5.9.1 confirm this expectation.

Theorem 26 *The $\wedge$ and $\vee$ decomposition of $\vec{X} \leq_{lex} \vec{Y}$ together either prove that $\vec{X} \leq_{lex} \vec{Y}$ is disentailed, or establish $GAC(\vec{X} \leq_{lex} \vec{Y})$.*

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 for all $\alpha \leq i < k$ we have $min(X_i) = max(Y_i)$. In the first case, the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leq Y_\alpha$ is $false$. In the second case, the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{k-1} = Y_{k-1} \rightarrow X_k \leq Y_k$ is $false$ because for all $\alpha \leq i < k$ we get $X_i \doteq Y_i$ due to the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \ \ldots \ \wedge X_{i-1} = Y_{i-1} \rightarrow X_i \leq Y_i$. In any case, $\vec{X} \leq_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 8. If, however, $\alpha < \beta$ then the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leq Y_\alpha$ makes sure that AC($X_\alpha \leq Y_\alpha$). Now consider the $\vee$ decomposition. If $\beta = \alpha$ then all the disjuncts of the decomposition are $false$, so $\vec{X} \leq_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 8. If $\beta = \alpha + 1$ then each of the disjuncts but $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{\alpha-1} = Y_{\alpha-1} \ \wedge \ X_\alpha < Y_\alpha$ is $false$. This makes sure that AC($X_\alpha < Y_\alpha$). Given $\beta > \alpha$, we have:

- $\beta = \alpha + 1 \rightarrow AC(X_\alpha < Y_\alpha)$
- $\beta > \alpha + 1 \rightarrow AC(X_\alpha \leq Y_\alpha)$

By Theorem 9, we have GAC($\vec{X} \leq_{lex} \vec{Y}$). QED.

Theorem 27 *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 the cases $\beta = n$ and $\beta < n \ \wedge \ \forall \beta \leq i < n \,.\, min(X_i) = max(Y_i)$, as for the remaining cases the proof of Theorem 26 proves also this theorem. 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 $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{n-2} = Y_{n-2} \rightarrow X_{n-1} < Y_{n-1}$, which is the last constraint of the conjunction, makes sure that AC($X_\alpha < Y_\alpha$). In the second case, the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leq Y_\alpha$ makes sure that AC($X_\alpha \leq Y_\alpha$).

Now assume $\beta < n \wedge \forall \beta \leq i < n \,.\, min(X_i) = max(Y_i)$. Consider the $\wedge$ decomposition. If $\alpha = \beta$ the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{n-2} = Y_{n-2} \rightarrow X_{n-1} < Y_{n-1}$ is *false* because for all $\alpha \leq i < n-1$ we get $X_i \doteq Y_i$ due to the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{i-1} = Y_{i-1} \rightarrow X_i \leq Y_i$. Hence, $\vec{X} \leq_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 8. If, however, $\alpha < \beta$ then the constraint $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leq Y_\alpha$ makes sure that AC($X_\alpha \leq Y_\alpha$). Now consider the $\vee$ decomposition. If $\beta = \alpha$ then all the disjuncts of the decomposition are *false*, so $\vec{X} \leq_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 8. If $\beta = \alpha + 1$ then each of the disjuncts but $X_0 = Y_0 \ \wedge \ X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_{\alpha-1} = Y_{\alpha-1} \ \wedge \ X_\alpha < Y_\alpha$ is *false*. This makes sure that AC($X_\alpha < Y_\alpha$).

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

- $\beta = \alpha + 1 \rightarrow AC(X_\alpha < Y_\alpha)$
- $\beta > \alpha + 1 \rightarrow AC(X_\alpha \leq Y_\alpha)$

By Theorem 15, we have GAC($\vec{X} <_{lex} \vec{Y}$). QED.

5.6 Multiple Vectors

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

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

Theorem 28 *GAC($\vec{X}_i \leq_{lex} \vec{X}_j$) for all $0 \leq i < j \leq n-1$ is strictly stronger than GAC($\vec{X}_i \leq_{lex} \vec{X}_{i+1}$) for all $0 \leq i < n-1$.*

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

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

We have GAC($\vec{X}_i \leq_{lex} \vec{X}_{i+1}$) for all $0 \leq i < 2$. For the vectors $\vec{X}_0$ and $\vec{X}_2$, we have $\beta = \alpha + 1$ but $X_{0,\alpha} <_{lex} X_{2,\alpha}$ is not AC. Therefore, GAC($\vec{X}_0 \leq_{lex} \vec{X}_2$) does not hold. QED.

Theorem 29 *GAC($\vec{X}_i <_{lex} \vec{X}_j$) for all $0 \leq i < j \leq n-1$ is strictly stronger than GAC($\vec{X}_i <_{lex} \vec{X}_{i+1}$) for all $0 \leq i < n-1$.*

Proof: The example in Theorem 28 shows strictness. QED.

Theorem 30 *GAC($\forall ij\ 0 \leq i < j \leq n-1\,.\ \ \vec{X}_i \leq_{lex} \vec{X}_j$) is strictly stronger than GAC($\vec{X}_i \leq_{lex} \vec{X}_j$) for all $0 \leq i < j \leq n-1$.*

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

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

We have GAC($\vec{X}_i \leq_{lex} \vec{X}_j$) for all $0 \leq i < j \leq 2$. The assignment $X_{0,0} \leftarrow 1$ is supported by $\vec{X}_0 = \langle\{1\},\{0\},\{1\},\{0,1\}\rangle$, $\vec{X}_1 = \langle\{1\},\{1\},\{0\},\{1\}\rangle$, and $\vec{X}_2 = \langle\{1\},\{1\},\{0\},\{0\}\rangle$. In this case, $\vec{X}_1 \leq_{lex} \vec{X}_2$ is $false$. Therefore, GAC($\forall ij\ 0 \leq i < j \leq 2\,.\ \ \vec{X}_i \leq_{lex} \vec{X}_j$) does not hold. QED.

The example in Theorem 30 corrects a mistake that appears in [FHK$^+$02]: GAC($\forall ij\ 0 \leq i < j \leq n-1\,.\ \ \vec{X}_i \leq_{lex} \vec{X}_j$) is strictly stronger than GAC($\vec{X}_i \leq_{lex} \vec{X}_j$) for all $0 \leq i < j \leq n-1$ even for 0/1 variables.

Theorem 31 *GAC($\forall ij\ 0 \leq i < j \leq n-1\,.\ \ \vec{X}_i <_{lex} \vec{X}_j$) is strictly stronger than GAC($\vec{X}_i <_{lex} \vec{X}_j$) for all $0 \leq i < j \leq n-1$.*

Proof: The example in Theorem 30 shows strictness. QED.

5.7 Lexicographic Ordering with Variable Sharing

`LexLeq` has been designed based on Theorem 9 which assumed that the variables in the vectors are all distinct, i.e. a variable in a vector is not repeated in any of the vectors. In this section we consider the case when there is at least one variable in a vector that occurs more than once in any of the vectors.

When do we have vectors with shared variables? As an example, consider the row and column symmetries of an $n \times m$ matrix. A way to break all such symmetry is to add one constraint for each symmetry [CGLR96]. Consider a 2×2 matrix X of variables:

$$\begin{pmatrix} X_0 & X_1 \\ X_2 & X_3 \end{pmatrix}$$

which we represent as a vector $\langle X_0, \ldots, X_3\rangle$. The complete set of symmetry breaking constraints is composed by imposing $X \leq X'$, where X' is a matrix obtained by permuting the rows and/or columns of X:

$$\langle X_0, X_1, X_2, X_3\rangle \leq_{lex} \langle X_2, X_3, X_0, X_1\rangle$$

$$\langle X_0, X_1, X_2, X_3\rangle \leq_{lex} \langle X_1, X_0, X_3, X_2\rangle$$

$$\langle X_0, X_1, X_2, X_3\rangle \leq_{lex} \langle X_3, X_2, X_1, X_0\rangle$$

In each of the constraints above, there are 4 variables shared between the vectors constrained to be lexicographically ordered.

Why is `LexLeq` weak when there are shared variables? The fact that $X_\alpha \doteq Y_\alpha$ is a consistent assignment when $\beta > \alpha + 1$ may not be valid anymore in this new context. Consider the vectors:

$$\begin{array}{rcllll}
\vec{X} & = & \langle\{0\}, & \{0\}, & \{1\}\rangle \\
\vec{Y} & = & \langle\{0,1\}, & \{0,1\}, & \{0\}\rangle \\
 & & \ \uparrow\alpha & & \ \uparrow\beta
\end{array}$$

and the constraint $\vec{X} \leq_{lex} \vec{Y}$. We have $\beta > \alpha + 1$ and AC($X_\alpha \leq Y_\alpha$). Assuming that the variables are not shared, the assignment $X_\alpha \doteq Y_\alpha \leftarrow 0$ is supported by $\vec{x} = \langle 0, 0, 1\rangle \leq_{lex} \vec{y} = \langle 0, 1, 0\rangle$. Now assume that Y_0 and Y_1 are shared. By $X_\alpha \doteq Y_\alpha \leftarrow 0$, we get $\vec{X} \leftarrow \langle 0, 0, 1\rangle$ and $\vec{Y} \leftarrow \langle 0, 0, 0\rangle$, which violates $\vec{X} \leq_{lex} \vec{Y}$. So AC($X_\alpha < Y_\alpha$) must be enforced.

5.7.1 Remedy

Assume $\alpha + 1 < \beta < n$ and AC($X_\alpha \leq Y_\alpha$). If there is a variable in the range $[\alpha, \beta)$ of any of the vectors which is shared by only some variables in $[\beta, n)$, then the support of the assignment $X_\alpha \doteq Y_\alpha$ remains valid. This may not be the case if a variable in the range $[\alpha, \beta)$ is shared by some variables in the same range. The assignment $X_\alpha \doteq Y_\alpha$ might not be supported anymore as shown above.

So what can we do? We start by slightly changing the definition of α: for all $0 \leq i < \alpha$ we have either $X_i \doteq Y_i$ or $X_i = Y_i$ (shared), and at α we have $\neg(X_\alpha \doteq Y_\alpha)$ and $X_\alpha \neq Y_\alpha$ (not shared). After setting the values of α and β, given that $\beta > \alpha + 1$ we first establish AC($X_\alpha \leq Y_\alpha$). Assume we still have $\neg(X_\alpha \doteq Y_\alpha)$. We verify that $X_\alpha \doteq Y_\alpha$ is a consistent assignment by propagating the support gained from every variable to those shared by the variable. Let us illustrate this on an example. Consider the vectors:

$$\begin{array}{rccccccc}
 & & & & * & \star & \star & \\
\vec{X} & = & \langle\{0,1\}, & \{1\}, & \{0,1\}, & \{0,1\}, & \{0,1\}, & \{1\}\rangle \\
\vec{Y} & = & \langle\{0,1\}, & \{0,1\}, & \{0\}, & \{1\}, & \{0,1\}, & \{0\}\rangle \\
 & & \uparrow\alpha & \star & & & * & \uparrow\beta
\end{array}$$

where Y_1, X_3, and X_4 are shared (indicated by a $\star$), as well as X_2 and Y_4 (indicated by a $*$). We have $\beta = 5$, $\alpha = 0$, AC($X_\alpha \leq Y_\alpha$) and $\neg(X_\alpha \doteq Y_\alpha)$. We maintain two vectors $\vec{sx}$ and $\vec{sy}$ of length $\beta - \alpha - 1$ where we will place the support we are looking for by taking into account the shared variables.

$$\begin{array}{rcllll}
 & & & * & \star & \star \\
\vec{sx} & = & \langle\_, & \_, & \_, & \_\rangle \\
\vec{sy} & = & \langle\_, & \_, & \_, & \_\rangle \\
 & & \ \star & & & *
\end{array}$$

Since X_α and Y_α are not shared by any other variables, we look for a support for the assignment $X_\alpha \doteq Y_\alpha$ without having to know what value is assigned to these variables.

We start at $\alpha + 1$ where the only support for $X_\alpha \doteq Y_\alpha$ is $X_{\alpha+1} \doteq Y_{\alpha+1} \leftarrow 1$. We place this support on $\vec{sx}$ and $\vec{sy}$ taking into account the shared variables.

$$\begin{array}{rcllll} & & & * & \star & \star \\ \vec{sx} & = & \langle 1, & \_, & 1, & 1\rangle \\ \vec{sy} & = & \langle 1, & \_, & \_, & \_\rangle \\ & & \star & & & * \end{array}$$

We move to $\alpha + 2$. The only support is $X_{\alpha+2} \doteq Y_{\alpha+2} \leftarrow 0$. Considering the shared variables we get:

$$\begin{array}{rcllll} & & & * & \star & \star \\ \vec{sx} & = & \langle 1, & 0, & 1, & 1\rangle \\ \vec{sy} & = & \langle 1, & 0, & \_, & 0\rangle \\ & & \star & & & * \end{array}$$

At $\alpha + 3$, the support is $X_{\alpha+3} \doteq Y_{\alpha+3} \leftarrow 1$.

$$\begin{array}{rcllll} & & & * & \star & \star \\ \vec{sx} & = & \langle 1, & 0, & 1, & 1\rangle \\ \vec{sy} & = & \langle 1, & 0, & 1, & 0\rangle \\ & & \star & & & * \end{array}$$

Since we have now $\vec{sx} >_{lex} \vec{sy}$, the assignment $X_\alpha \doteq Y_\alpha$ lacks support. We thus enforce AC($X_\alpha < Y_\alpha$) on the original vectors.

5.7.2 An Alternative Filtering Algorithm

A filtering algorithm for $\vec{X} \leq_{lex} \vec{Y}$ in the presence of shared variables is given in Algorithm 11. This algorithm is similar to the original algorithm `LexLeq` except for the case $i = \alpha$ and $i + 1 < \beta$. Note that at α we have $X_\alpha \neq Y_\alpha$ (not shared).

In addition to the original steps of `LexLeq` in lines 2.1 and 2.2, we add lines **a-k** to the case when $i = \alpha \ \wedge \ i + 1 < \beta$. If $max(X_\alpha) < max(Y_\alpha)$ after executing line 2.1 then every value a in $\mathcal{D}(X_\alpha)$ has a support in $\mathcal{D}(Y_\alpha)$ which is greater than a. Similarly, if $min(X_\alpha) < min(Y_\alpha)$ then every value a in Y_α has a support in $\mathcal{D}(X_\alpha)$ which is less than a. Variable sharing is irrelevant when $max(X_\alpha) < max(Y_\alpha)$ and $min(X_\alpha) < min(Y_\alpha)$, and therefore the algorithm returns in line **a**.

If, however, $max(X_\alpha) = max(Y_\alpha)$ then $max(X_\alpha)$ is supported by only $max(Y_\alpha)$ in $\mathcal{D}(Y_\alpha)$, therefore we need to make sure that the assignment $X_\alpha \doteq Y_\alpha \leftarrow max(X_\alpha)$ is consistent. Similarly, if $min(X_\alpha) = min(Y_\alpha)$ then $min(Y_\alpha)$ is supported by only $min(X_\alpha)$ in $\mathcal{D}(X_\alpha)$, therefore we need to verify that $X_\alpha \doteq Y_\alpha \leftarrow min(X_\alpha)$ is consistent. We start by checking in line **b** whether there is at least one variable in $[\alpha, min\{n, \beta\})$ being shared by another variable within the same interval. If there is not then we are done. Otherwise, we start seeking support for the assignment $X_\alpha \doteq Y_\alpha$ in the subvectors $\vec{X}_{\alpha+1 \to min\{n,\beta\}-1}$ and $\vec{Y}_{\alpha+1 \to min\{n,\beta\}-1}$ (lines **c-k**).

We distinguish between two cases. First, if X_α and Y_α are not shared by any other variables in $[\alpha + 1, min\{n, \beta\})$ then we do not need to know the value assigned to these variables while seeking support for the assignment $X_\alpha \doteq Y_\alpha$ (lines **c-e**). Second, if any of X_α and Y_α is shared in $[\alpha + 1, min\{n, \beta\})$ then we need to know whether we seek support

Algorithm 11: `LexLeq`(i)

```
    Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, Integer $i$
    Result : GAC($\vec{X} \leq_{lex} \vec{Y}$)

    ⋮
2    if $i = \alpha \ \wedge \ i + 1 < \beta$ then
2.1      AC($X_i \leq Y_i$);
2.2      if $X_i \doteq Y_i$ then UpdateAlpha($i + 1$);
a        if $max(X_i) < max(Y_i) \ \wedge \ min(X_i) < min(Y_i)$ then return;
b        if there are shared variables in $[i, min\{n, \beta\})$ then
c            if $X_i$ and $Y_i$ are not shared in $[i + 1, min\{n, \beta\})$ then
d                if ¬SeekSupport($-1$) then
e                    AC($X_i < Y_i$);
                 end
             else
f                if $max(X_i) = max(Y_i)$ then
g                    if ¬SeekSupport($max(X_i)$) then
h                        NC($X_i < max(X_i)$);
                     end
                 end
i                if $min(Y_i) = min(X_i)$ then
j                    if ¬SeekSupport($min(Y_i)$) then
k                        NC($Y_i > min(Y_i)$);
                     end
                 end
             end
         end
     end
     ⋮
```

for $X_\alpha \doteq Y_\alpha \leftarrow max(X_\alpha)$ (lines **f**-**h**) or for $X_\alpha \doteq Y_\alpha \leftarrow min(X_\alpha)$ (lines **i**-**k**). In the first case, we pass -1 to `SeekSupport`. If no support is found for $X_\alpha \doteq Y_\alpha$ (line **d**) then we enforce AC($X_\alpha < Y_\alpha$) in line **e**. In the second case, we pass $max(X_\alpha)$ to `SeekSupport`. If no support is found for $X_\alpha \doteq Y_\alpha \leftarrow max(X_\alpha)$ (line **g**) then we decrease $max(X_\alpha)$ by maintaining node consistency on $X_\alpha < max(X_\alpha)$ via the call `NC` (line **h**). Similarly, after passing $min(X_\alpha)$ to `SeekSupport`, if no support is found for $X_\alpha \doteq Y_\alpha \leftarrow min(X_\alpha)$ (line **j**) then we increase $min(Y_\alpha)$ (line **k**).

We call `SeekSupport` for constructing a pair of vectors $\vec{Sx}$ and $\vec{Sy}$ to check whether the assignment $X_\alpha \doteq Y_\alpha$ is supported. The input *value* is -1 if the value assigned to X_α and Y_α is irrelevant. Otherwise, *value* is either $max(X_\alpha)$ or $min(X_\alpha)$.

These vectors are of length $Limit - 1 - \alpha$, where $Limit$ is β when $\beta < n$, but is n when $\beta = \infty$. To be able to propagate the values assigned to shared variables, $\vec{Sx}$ and $\vec{Sy}$ are vectors of variables, and are the copies of the original vectors $\vec{X}_{\alpha+1 \to Limit-1}$ and $\vec{Y}_{\alpha+1 \to Limit-1}$. So, whenever we assign a value to one of the variables in $\vec{Sx}$ and $\vec{Sy}$, the same value will be assigned to the shared variables. Lines 1-3 construct the vectors.

We assign values to Sx_i and Sy_i starting from $\alpha + 1$ as follows. At i, if Sx_i is not yet ground and X_i is shared by any of X_α and Y_α (lines 6-7), then we assign Sx_i either

```
Procedure Boolean SeekSupport($value$)
1   $Limit := \beta$;
2   if $Limit > n$ then $Limit := n$;
3   $\langle \vec{Sx}_{\alpha+1 \to Limit-1}, \vec{Sy}_{\alpha+1 \to Limit-1} \rangle := copyterm(\langle \vec{X}_{\alpha+1 \to Limit-1}, \vec{Y}_{\alpha+1 \to Limit-1} \rangle)$;
4   $i := \alpha + 1$;
5   while $i < Limit$ do
6       if $value \neq -1 \ \wedge \ \neg$Bound($Sx_i$) then
7           if $X_i = X_\alpha \ \vee \ X_i = Y_\alpha$ then
8               $Sx_i \leftarrow value$;
            end
        end
9       if $value \neq -1 \ \wedge \ \neg$Bound($Sy_i$) then
10          if $Y_i = X_\alpha \ \vee \ Y_i = Y_\alpha$ then
11              $Sy_i \leftarrow value$;
            end
        end
12      if $\neg$Bound($Sx_i$) then $Sx_i \leftarrow min(X_i)$;
13      if $\neg$Bound($Sy_i$) then $Sy_i \leftarrow max(Y_i)$;
14      if $Sx_i \lessdot Sy_i$ then return $true$;
15      if $Sx_i \gtrdot Sy_i$ then return $false$;
16      if $Sx_i \doteq Sy_i$ then $i := i + 1$;
    end
17  return($Limit = n$);
```

$max(X_\alpha)$ or $min(X_\alpha)$ in line 8 depending on the input *value*. On the other hand, if Sx_i is not yet ground and X_i is not shared by any of X_α and Y_α (line 12), then we assign $min(X_i)$ to Sx_i. This is the best support we can get from X_i. In lines 9-11 and 13, we repeat a similar procedure for Sy_i except that we assign $max(Y_i)$ if Sy_i is not yet ground and Y_i is not shared by any of X_α and Y_α (line 13).

Finally, we either return *true* if $Sx_i \lessdot Sy_i$ (line 14), or return *false* if $Sx_i \gtrdot Sy_i$ (line 15), or continue if $Sx_i \doteq Sy_i$ (line 16). At the end of the vectors (line 17), we return *true* if $\beta > n$ but *false* otherwise.

Procedure `SeekSupport` runs in time $O(n)$ in the worst case. Checking whether there are any shared variables in $[\alpha, min\{n, \beta\})$ can also be done in time $O(n)$ by scanning a data structure in linear time which records which variables are shared. Hence, the worst-case complexity of `LexLeq` remains $O(nb)$.

5.8 Related Work

The ECLiPSe constraint solver [WNS97] provides a global constraint, called **lexico_le**, for imposing lexicographic ordering constraint on two vectors. It is not documented in [ECL03] what level of consistency is enforced with this constraint. With some experimentation, we believe that **lexico_le** is stronger than each of the decompositions discussed in Section 5.5.2. For instance, $\vec{X} = \langle \{0,1\}, \{1\} \rangle \ \wedge \ \vec{Y} = \langle \{0,1\}, \{0\} \rangle \ \wedge$ **lexico_le**$(\vec{X}, \vec{Y})$ gives $\vec{X} = \langle \{0\}, \{1\} \rangle$ and $\vec{Y} = \langle \{1\}, \{0\} \rangle$ but the $\wedge$ decomposition leaves the vectors unchanged. Likewise, $\vec{X} = \langle \{0,1,2\}, \{0,1\} \rangle \ \wedge \ \vec{Y} = \langle \{0,1\}, \{0,1\} \rangle \ \wedge$ **lexico_le**$(\vec{X}, \vec{Y})$ 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. On the other hand, **lexico_le**$(\vec{X},\vec{Y})$ does not maintain GAC on $\vec{X} \leq_{lex} \vec{Y}$. For instance $\vec{X} = \langle\{0,1\},\{0,1\},\{1\}\rangle \ \wedge \ \vec{Y} = \langle\{0,1\},\{0\},\{0\}\rangle \ \wedge$ **lexico_le**$(\vec{X},\vec{Y})$ leaves the vectors unchanged, even though $\vec{X} \leq_{lex} \vec{Y}$ is not GAC.

An alternative way of propagating a global constraint is to pose a set of constraints as opposed to designing a special-purpose filtering algorithm. This approach is referred to as constraint encoding and amounts to simulating the behaviour of an algorithm by enforcing an appropriate level of consistency on the posted constraints. The success of such an approach was demonstrated in [GIM$^+$01] by showing that arc-consistency on the CSP representation of the stable marriage problem gives reduced domains which are equivalent to the GS-lists produced by the Extended Gale-Shapley algorithm.

Following this line of research, an encoding of the lexicographic ordering constraint by Gent *et al.* is described in [GPS02]. Firstly, they assume that the vectors $\vec{X}$ and $\vec{Y}$ are indexed from 1 to n, and introduce a new vector α of 0/1 variables indexed from 0 to n. The intended meaning of α is that:

- if $\alpha_i = 1$ then $X_j = Y_j$ for all $1 \leq j \leq i$;
- if $\alpha_{i+1} = 0$ but $\alpha_i = 1$ then $X_{i+1} < Y_{i+1}$.

Secondly, they pose the following constraints:

$$\alpha_0 = 1 \tag{5.1}$$
$$\forall i \ 0 \leq i \leq n-1\,. \quad \alpha_i = 0 \rightarrow \alpha_{i+1} = 0 \tag{5.2}$$
$$\forall i \ 1 \leq i \leq n\,. \quad \alpha_i = 1 \rightarrow X_i = Y_i \tag{5.3}$$
$$\forall i \ 0 \leq i \leq n-1\,. \quad \alpha_i = 1 \ \wedge \ \alpha_{i+1} = 0 \rightarrow X_{i+1} < Y_{i+1} \tag{5.4}$$
$$\forall i \ 0 \leq i \leq n-1\,. \quad \alpha_i = 1 \rightarrow X_{i+1} \leq Y_{i+1} \tag{5.5}$$

If strict lexicographic ordering between the vectors is wanted then it suffices to add $\alpha_n = 0$.

Warwick Harvey suggests another encoding [Har02]. To ensure $\vec{X} \leq_{lex} \vec{Y}$, he poses:

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

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:

$$1 = (X_1 < Y_1 + (X_2 < Y_2 + (... + (X_n < Y_n + 0)...)))$$

which disallows $X_n \doteq Y_n$ in case the vectors are ground and equal until the last index.

Both of the encodings were proposed in knowledge of our algorithm `LexLeq` and they simulate the behaviour of `LexLeq`. Whenever $\alpha = \beta$, the posted constraints fail. If $\beta > \alpha + 1$ then $X_\alpha \leq Y_\alpha$, but if $\beta = \alpha + 1$ then $X_\alpha < Y_\alpha$ is enforced. Assuming that arc-consistency is established on $X_\alpha \leq Y_\alpha$ and $X_\alpha < Y_\alpha$, we get GAC on $\vec{X} \leq_{lex} \vec{Y}$. The advantage of such an approach is that we can avoid special-purpose filtering algorithms and instead rely on the existing and more general filtering algorithms. On the other hand, as our experimental results in Section 5.9.2 show, the introduction of extra variables and constraints may result in a less efficient way of propagating the constraint. Hence, a specialised algorithm might be the most efficient way of posting and solving a constraint.

Subsequent to [FHK$^+$02], an alternative filtering algorithm for the lexicographic ordering constraint, which is derived from a finite automaton operating on a signature of

$\vec{X} \leq_{lex} \vec{Y}$, is presented by Carlsson and Beldiceanu in [CB02b]. The algorithm maintains generalised arc-consistency or detects (dis)entailment, and runs in linear time for posting plus amortised constant time per propagation event. Both our algorithm and the algorithm of Carlsson and Beldiceanu record the position of the vectors above which the vectors are ground and equal as α and q, respectively. There is, however, no counterpart of β in the latter algorithm. In a following report, Carlsson and Beldiceanu introduce a new global constraint, called **lex_chain**, for imposing lexicographic ordering constraints on a chain of vectors [CB02a]. The filtering algorithm is derived from a finite automaton operating on a signature of $\vec{l_i} \leq_{lex} \vec{X_i} \leq_{lex} \vec{u_i}$ where X_i is a vector in the chain, and $\vec{l_i}$ and $\vec{u_i}$ are the feasible lower and upper bounds of $\vec{X_i}$. 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 the corresponding bounds. Given m vectors, the algorithm maintains generalised arc-consistency or detects (dis)entailment, and runs in time $O(nmd)$ where d is the cost of necessary domain operations. As Theorem 30 shows, such an algorithm can yield more pruning compared to propagating lexicographic ordering constraints between adjacent or every pair of vectors in the chain. Unfortunately, no experimental results are provided in [CB02a] to show the benefits of **lex_chain**. We have therefore conducted some experiments to judge how much more propagation in practice is achieved with this new global constraint. Our results indicate no gain in the amount of constraint propagation. For a detailed discussion, see Chapter 6.9.

5.9 Experimental Results

We implemented our global constraints $\leq_{lex}$ and $<_{lex}$ in C++ using ILOG Solver 5.3 [ILO02]. The global constraints encapsulate the corresponding filtering algorithm that either maintains GAC on (strict) lexicographic ordering constraint or establishes failure at each node of the search tree.

We performed a wide range of experiments to compare our global constraints with (1) the alternative ways of posing lexicographic ordering constraints which are presented in Section 5.5; (2) the constraint encodings of lexicographic ordering constraints which are presented in Section 5.8. The experiments are done using some of the problems discussed in Chapter 3, in which we either look for one solution or the optimal solution. Each of the problems can be modelled by matrices of decision variables where the rows and/or columns are (partially) symmetric. We can therefore pose lexicographic ordering constraints on the corresponding rows and/or columns to break much of this symmetry.

The results of the experiments are shown in tables where a "-" means no result is obtained in 1 hour (3600 secs). Whilst the number of choice points gives the number of alternatives explored in the search tree, the number of fails gives the number of incorrect decisions at choice points. The best result of each entry in a table is typeset in bold. If posing lexicographic ordering on the rows is done via a technique called $Tech$ then we write $Tech$ R. Similarly, posing lexicographic ordering on the columns using $Tech$ is specified as $Tech$ C, and on the rows and columns as $Tech$ RC. In theory posing lexicographic ordering constraints between every pair of rows (similarly for columns) leads to more pruning than posing between adjacent rows (see Section 5.6). We could not see any evidence of this in practise, therefore lexicographic ordering constraints are enforced just between the adjacent rows.

The experiments are conducted using ILOG Solver 5.3 on a 1Ghz pentium III processor with 256Mb RAM under Windows XP.

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 5.1: Instance specification for the progressive party problem.

5.9.1 Comparison with Alternative Approaches

We designed some experiments to test two goals. First, does our filtering algorithm(s) do more inference in practice than the $\wedge$ and $\vee$ decompositions? Similarly, is the algorithm more efficient in practice than the decompositions? Second, how does our algorithm compare to combining the decompositions, and BC on the arithmetic constraint? We propagate the arithmetic constraint via **IloScalProd** which maintains BC on the scalar product of two vectors.

We tested our global constraints on three problem domains: the progressive party problem, the template design problem, and the balanced incomplete block design problem.

Progressive Party Problem This was introduced in Chapter 3.2.5. In Figure 3.10, one way of modelling the problem is given. The time periods as well as the guests with equal crew size are indistinguishable. Hence, this model of the problem has partial row symmetry between the indistinguishable guests of H, and column symmetry.

Due to the problem constraints, no pair of rows/columns can be equal. Given a set of indistinguishable guests $\{g_i, g_{i+1}, \ldots, g_j\}$, we break the partial row symmetry by enforcing that the rows corresponding to such guests $\vec{R_i}, \vec{R}_{i+1}, \ldots, \vec{R_j}$ are strict anti-lexicographically ordered: $\vec{R_i} >_{lex} \vec{R}_{i+1} \ldots >_{lex} \vec{R_j}$. As for the column symmetry, we enforce that the columns $\vec{C_0}, \vec{C_1}, \ldots, \vec{C}_{p-1}$ corresponding to the p time periods are strict anti-lexicographically ordered: $\vec{C_0} >_{lex} \vec{C_1} \ldots >_{lex} \vec{C}_{p-1}$. We enforce the lexicographic ordering constraints either by using our filtering algorithm `LexLess` or the various alternative approaches.

We consider several instances of the progressive party problem, including the one mentioned in Chapter 3.2.5. 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. Table 5.1 shows the data. The last column of Table 5.1 gives the percentage of the total capacity used, which is a measure of constrainedness [Wal99].

As in [SBHW96], we give priority to the largest crews, so the guest boats are ordered in descending order of their size. Also, when assigning a host to a guest, we try a value first which is most likely to succeed. We therefore order the host boats in descending order of their spare capacity. In terms of variable ordering, we use *smallest-domain first* principle and choose next the variable that has the smallest domain size.

Inst.	LexLess RC			∧ RC			∨ RC		
#	Fails	Choice	Time	Fails	Choice	Time	Fails	Choice	Time
		points	(secs.)		points	(secs.)		points	(secs.)
1	**446**	**538**	**0.8**	-	-	-	-	-	-
2	**445**	**557**	**0.9**	445	557	1.1	-	-	-
3	**2,380**	**2,489**	**2.1**	3,651	3,766	2.9	-	-	-
4	**459**	**569**	**0.9**	-	-	-	-	-	-
5	**443**	**554**	**1.0**	443	554	1.1	-	-	-
6	8,481	8,598	5.7	**604**	**721**	**1.4**	-	-	-
7	**782**	**892**	**1.2**	-	-	-	-	-	-
8	33,849	33,951	15.5	**773**	**885**	**1.3**	-	-	-
9*	**211,075**	**211,171**	**112.2**	213,472	213,568	149.0	-	-	-

Table 5.2: Progressive party problem: LexLeq vs ∧ and ∨ decompositions.

Instance	LexLess RC			∧ ∨ RC
#	Fails	Choice	Time	Time
		points	(secs.)	(secs.)
1	446	538	**0.8**	1.1
2	445	557	**0.9**	1.2
3	2,380	2,489	**2.1**	3.0
4	459	569	**0.9**	1.1
5	443	554	**1.0**	1.2
6	8,481	8,598	**5.7**	8.0
7	782	892	**1.2**	1.5
8	33,849	33,951	**15.5**	21.0
9*	211,075	211,171	**112.2**	155.0

Table 5.3: Progressive party problem: LexLeq vs ∧ ∨ decomposition.

The results of the experiments are shown in Tables 5.2 and 5.3. 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.

According to the results in Table 5.2, LexLess is superior to the ∨ decomposition: none of the instances could be solved within an hour by the ∨ decomposition. However, it is difficult to judge which one of LexLess and the ∧ decomposition is superior to the other. LexLess solves instances 1, 4 and 7 very quickly, but the ∧ decomposition fails to return an answer in one hour. Also, instances 3 and 9 are solved with less failures by LexLess. On the other hand, ∧ decomposition is superior to LexLess 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 GAC on $\vec{X} <_{lex} \vec{Y}$ is strictly stronger than the ∧ decomposition of $\vec{X} <_{lex} \vec{Y}$, maintaining the latter at every choice point may lead to a smaller search tree than maintaining the former due the dynamic nature of the variable ordering.

In Table 5.3, we compare LexLess with combining the decompositions. Even though the same search tree is explored by the two, LexLess is more efficient especially on rather

Slots per Template	Variations	Order Quantities (/1000)
42	30	60, 60, 70, 70, 70, 70 ,70, 70, 70, 80, 80, 80, 80, 90, 90, 90, 90, 90, 90, 100, 100, 100, 100, 150, 230, 230, 230, 230, 280, 280

Table 5.4: The data for the herbs problem in [PS98].

difficult instances 8 and 9. Note that posing the arithmetic constraint is not feasible for this problem, as the largest coefficient necessary is 13^{28}, which is larger than 2^{31}, the maximum integer size allowed in Solver 5.3.

In summary, for the progressive party problem, posing the arithmetic constraint is not feasible for the instances that we considered. With the dynamic labelling heuristic that chooses next the variable with smallest domain size, LexLess is clearly superior to $\vee$ decomposition. We, however, observe that the $\wedge$ decomposition could give either smaller or larger search trees than LexLess. By combining the decompositions, we achieve the same pruning as LexLess but this may result in longer run-times.

Template Design Problem This was introduced in Chapter 3.2.2. In Figure 3.4, one way of modelling the problem is given. In this model the templates, as well as the variations with equal demands are indistinguishable. Hence, this model of the problem has symmetry between the variables of *Run*, and partial row symmetry between the indistinguishable variations of T. Note that there is also partial column symmetry between the templates of T with equal pressings, but this will not be considered here.

Given a set of indistinguishable variations $\{v_i, v_{i+1}, \ldots, v_j\}$, we break the partial row symmetry by enforcing that the rows corresponding to such variations $\vec{R_i}, \vec{R}_{i+1}, \ldots, \vec{R_j}$ are lexicographically ordered: $\vec{R_i} \leq_{lex} \vec{R}_{i+1} \ldots \leq_{lex} \vec{R_j}$. We enforce the lexicographic ordering constraints between the indistinguishable variations of T by either using our filtering algorithm LexLeq or the various alternative approaches.

We consider an instance of the template design problem, which is the herbs problem [PS98], where herb cartoons for a variety of herbs are to be printed on templates. The specification of the problem is in Table 5.4. We extend the basic model given in Chapter 3.2.2 by taking into account the additional constraints proposed in [PS98]. These constraints are (1) the symmetry breaking constraints which distinguish between the templates:

$$\forall i\ 0 \leq i < t-1\ .\ Run_i \leq_{lex} Run_{i+1}$$

and between the variations with equal demands when $t = 2$:

$$\forall j\ 0 \leq j < v-1\ .\ d_j = d_{j+1}\ \wedge\ T_{0,j} < T_{0,j+1} \rightarrow T_{1,j} > T_{1,j+1}$$

(2) the "pseudo-symmetry" breaking constraints:

$$\forall j\ 0 \leq j < v-1\ .\ d_j < d_{j+1} \rightarrow \sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} \leq \sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j+1}$$

(3) the implied constraints which provide an upper bound on the cost function:

$$\forall j \in \mathcal{V}ariations\ .\ \sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} - d_j \leq Surplus$$

t	Goal	`LexLeq` R			$\wedge$ R			$\vee$ R		
		Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
2	find	**22**	**49**	**0.1**	22	49	0.2	22	49	0.2
	prove	**49**	**49**	**0.1**	49	49	0.2	49	49	0.2
3	find	**5**	**52**	**0.1**	18,285	18,341	7.3	18,786	18,842	9.0
	prove	**52**	**52**	**0.1**	18,341	18,341	7.3	18,842	18,842	9.0
4	find	**6**	**70**	**0.1**	-	-	-	-	-	-
	prove	**70**	**70**	**0.1**	-	-	-	-	-	-
5	find	**4**	**77**	**0.1**	-	-	-	-	-	-
	prove	**77**	**77**	**0.1**	-	-	-	-	-	-

Table 5.5: Herbs problem with 10% over- and under-production limit: `LexLeq` vs $\wedge$ and $\vee$ decompositions.

$$\forall k\ 0 \leq k < n-1\,.\ \sum_{0\leq j\leq k}\left(\sum_{i\in\mathcal{T}emplates} Run_i * T_{i,j} - d_j\right) \leq Surplus$$

(4) the implied constraints on the number of pressings when $t = 2$:

$$Run_0 \leq \sum_{i\in\mathcal{T}emplates} Run_i/2$$

$$Run_1 \geq \sum_{i\in\mathcal{T}emplates} Run_i/2$$

and when $t = 3$:

$$Run_0 \leq \sum_{i\in\mathcal{T}emplates} Run_i/3$$

$$Run_1 \leq \sum_{i\in\mathcal{T}emplates} Run_i/2$$

$$Run_2 \geq \sum_{i\in\mathcal{T}emplates} Run_i/3$$

where we have $t = |\mathcal{T}emplates|$, $v = |\mathcal{V}ariations|$, and $Surplus = \sum_{i\in\mathcal{T}emplates} Run_i * s * - \sum_{j\in\mathcal{V}ariations} d_j$.

As for labelling heuristic, we adopt the static variable ordering proposed in [PS98]: we first label the variations of T, and then the variables of Run.

As in [PS98], 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 `LexLeq` 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 around 23 seconds by `LexLeq`, the $\wedge$ decomposition and the arithmetic constraint, but all of them fail to prove optimality within an hour. Changing the labelling heuristic by assigning first the Run variables and then the variations of T 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%.

t	Goal	`LexLeq` R			$\vee \wedge$ R	Arithmetic Constraint R
		Fails	Choice	Time	Time	Time
			points	(secs.)	(secs.)	(secs.)
2	find	22	49	**0.1**	0.2	0.2
	prove	49	49	**0.1**	0.2	0.2
3	find	5	52	**0.1**	0.2	0.2
	prove	52	52	**0.1**	0.2	0.2
4	find	6	70	**0.1**	0.2	0.2
	prove	70	70	**0.1**	0.2	0.2
5	find	4	77	**0.1**	0.2	0.2
	prove	77	77	**0.1**	0.2	0.2

Table 5.6: Herbs problem with 10% over- and under-production limit: `LexLeq` vs $\wedge$ $\vee$ decomposition and the arithmetic constraint.

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 [PS98], 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 Tables 5.5 and 5.6.

We observe in Table 5.5 that as the number of templates increase, the search effort and time required to find a solution and prove the optimality dramatically increase for the $\wedge$ and $\vee$ decompositions. On the other hand, `LexLeq` finds and proves solutions very quickly with much less effort. In particular, the 4 and 5 template problems can only be solved by `LexLeq`. In Table 5.6, we compare `LexLeq` with combining the decompositions, as well as with maintaining BC on the arithmetic constraint. In this case, the search tree generated is all the same, with some minor difference in solving times.

In summary, with the static labelling heuristic that assigns first the variables of *Run* and then T, `LexLeq` is clearly superior to both of the decompositions for the herbs problem with %10 allowed over- and under-production. On the other hand, the fact that the rows of the model are not too long and that the domain size is not too large makes it feasible to pose either the combination of the decompositions or the arithmetic constraint, both of which are highly competitive with our algorithm.

Balanced Incomplete Block Design Problem This was introduced in Chapter 3.2.1. In Figure 3.1, one way of modelling the problem is given. Since the elements as well as the subsets containing the elements are indistinguishable, the matrix X modelling the problem has row and column symmetry.

Due to the constraints on the rows, no pair of rows can be equal unless $r = \lambda$. To break the row symmetry, we enforce that the rows $\vec{R_0}, \vec{R_1}, \ldots, \vec{R_{v-1}}$ corresponding to the v elements are strict anti-lexicographically ordered: $\vec{R_0} >_{lex} \vec{R_1} \ldots >_{lex} \vec{R_{v-1}}$. As for the column symmetry, we enforce that the columns $\vec{C_0}, \vec{C_1}, \ldots, \vec{C_{b-1}}$ corresponding to the b subsets of $\mathcal{V}$ are anti-lexicographically ordered: $\vec{C_0} \geq_{lex} \vec{C_1} \ldots \geq_{lex} \vec{C_{b-1}}$. We pose the lexicographic ordering constraints either by using `LexLess` and `LexLeq`, or the corresponding alternative approaches.

In our experiments, we select some large instances from [CD96]. Note that these instances require matrices with very long columns, therefore posting the arithmetic con-

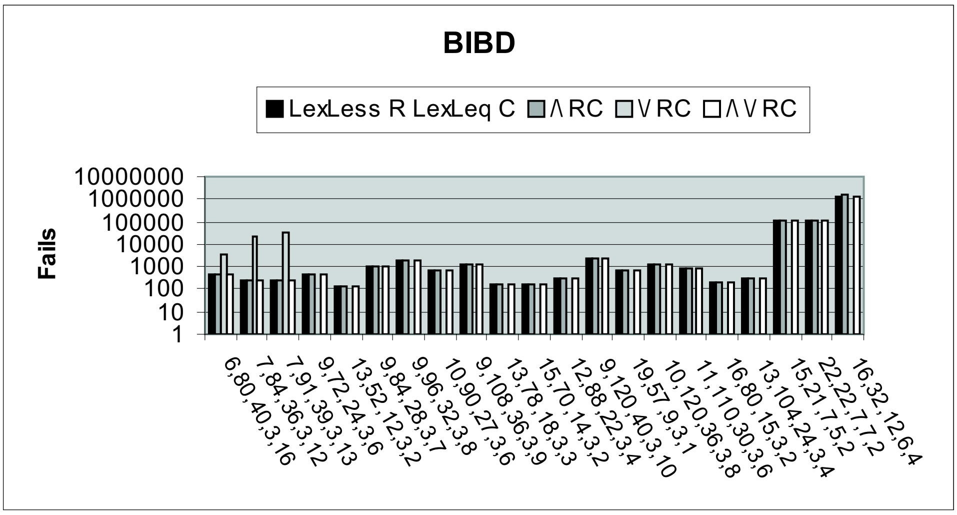

Figure 5.1: BIBD: `LexLeq`/`LexLess` vs $\wedge$, $\vee$, $\wedge\vee$ decompositions in terms of fails.

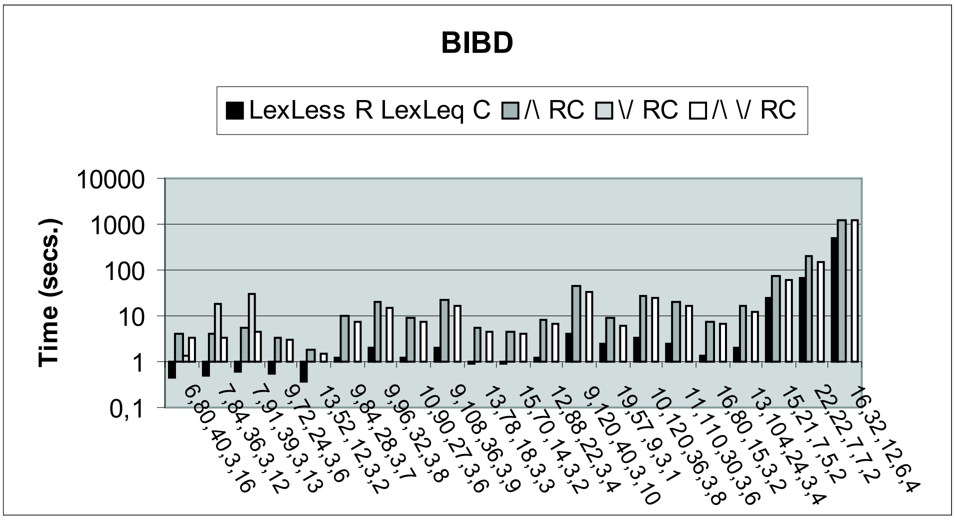

Figure 5.2: BIBD: `LexLeq`/`LexLess` vs $\wedge$, $\vee$, $\wedge\vee$ decompositions in terms of run-times.

straint is not feasible. As for the labelling heuristic, we adopt a static variable ordering, by instantiating the matrix X along its rows from top to bottom and exploring the domain of each variable in ascending order.

The results of the experiments along with the instances we have used are shown in Figures 5.1 and 5.2, where we contrast our algorithms with the alternative approaches in terms of failures and run-times respectively using a logarithmic scale. As seen in Figure 5.1, our algorithms and the $\wedge$ decomposition do exactly the same inference when solving every instance. An exception to this is the instances $\langle 15, 21, 7, 5, 2\rangle$ and $\langle 16, 32, 12, 6, 4\rangle$, for which the algorithms fail slightly less than the $\wedge$ decomposition. The $\vee$ decomposition, however, can solve only the first 3 instances within an hour limit, with many more failures.

In Figure 5.2, we observe a substantial gain in efficiency by using our algorithms in preference to all the other approaches considered. Even though the $\wedge$ decomposition and our algorithms explore the same search tree, the efficiency of the algorithms dramatically reduce the run-times. By combining the decompositions, we decrease the run-times com-

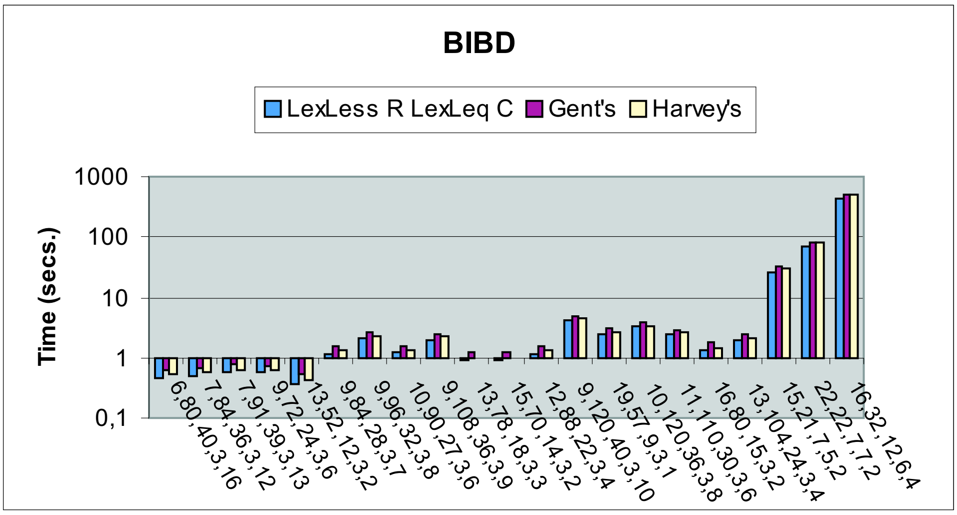

Figure 5.3: BIBD: LexLeq/LexLess vs constraint encodings in terms of run-times.

pared to the $\wedge$ decomposition. However, our algorithms remain the most efficient way of propagating the lexicographic ordering constraints.

In summary, for the BIBD, posing the arithmetic constraint is not feasible for the instances that we considered. By using the static labelling heuristic that instantiates the matrix along its rows from top to bottom, the filtering algorithms do much more pruning than the $\vee$ decomposition, but explore the same search tree as the $\wedge$ decomposition. On the other hand, our algorithms are much more efficient than all the decompositions including the combination of the decompositions.

5.9.2 Comparison with Constraint Encodings

We have shown in Section 5.8 that the filtering algorithm of the (strict) lexicographic ordering constraint can be simulated by posting a set of constraints. The filtering algorithm or the constraint encodings either maintain GAC or establish failure if the constraint cannot be satisfied. Therefore, the same search tree is generated by using either of them at each node of the search tree. Since the encodings introduce extra variables and constraints, we want to know which way of propagating $\leq_{lex}$ is the most efficient. How do our filtering algorithms compare to their corresponding encodings in terms of run-times, the number of variables used, the number of constraints posted, and the total memory used? For this purpose, we ran some experiments on the BIBD problem.

In our experiments, we use the same model, labelling heuristic, lexicographic ordering constraints, and the instances of BIBD described in Section 5.9.1. We pose the lexicographic ordering constraints either by using our filtering algorithms LexLess and LexLeq, or the corresponding constraint encodings of Gent *et al.* [GPS02] and Harvey [Har02]. Figures 5.3 to 5.6 show the results.

We observe in Figure 5.3 that the instances are solved quicker by the algorithmic approach (note the logarithmic scale), though the difference is not as much as the difference between the algorithms and the $\wedge$ decomposition in Figure 5.2. The constraint encodings are therefore competitive with the algorithms in terms of run-times. On the other hand, the introduction of extra variables results in dramatic difference in the total number of variables used (see Figure 5.4), constraints posted (see Figure 5.5), and the memory used

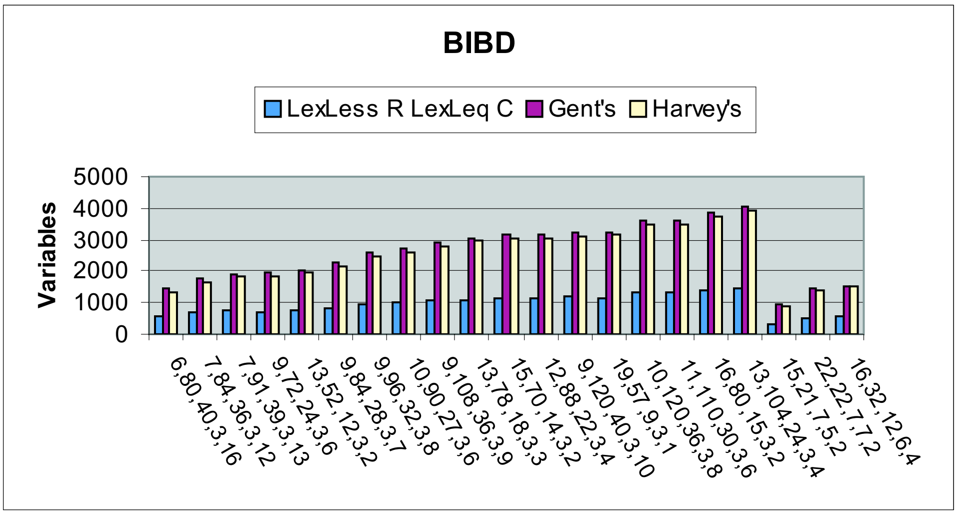

Figure 5.4: BIBD: `LexLeq`/`LexLess` vs constraint encodings in terms of number of variables.

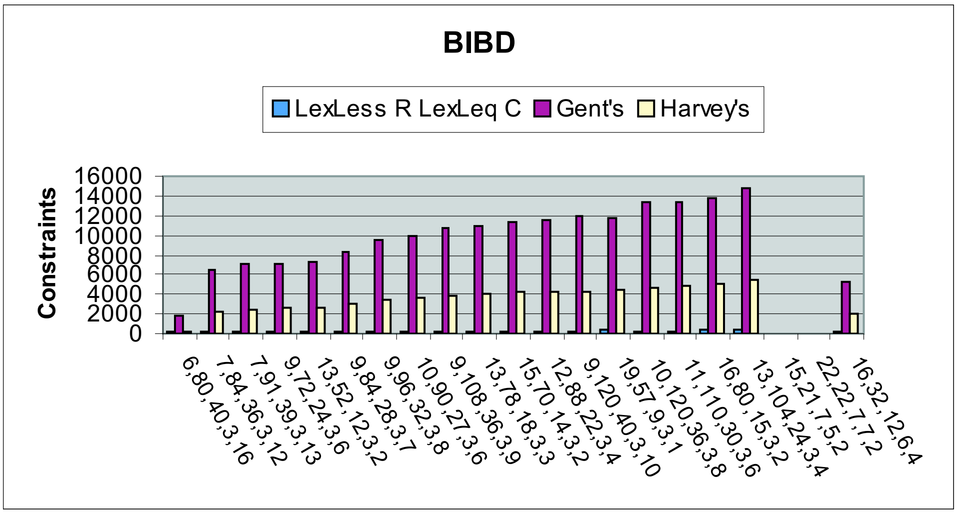

Figure 5.5: BIBD: `LexLeq`/`LexLess` vs constraint encodings in terms of constraints.

(see Figure 5.6) to solve the instances. This shows that propagating the lexicographic ordering constraint by a filtering algorithm is more (space) efficient than by posing a set of constraints.

5.10 Implementation: Incremental or Non-Incremental Algorithm?

We implemented our global constraints in C++ using ILOG Solver 5.3 [ILO02]. The implementation raised a number of questions, such as:

- at which propagation events to wake up the constraints;
- when to propagate the constraints.

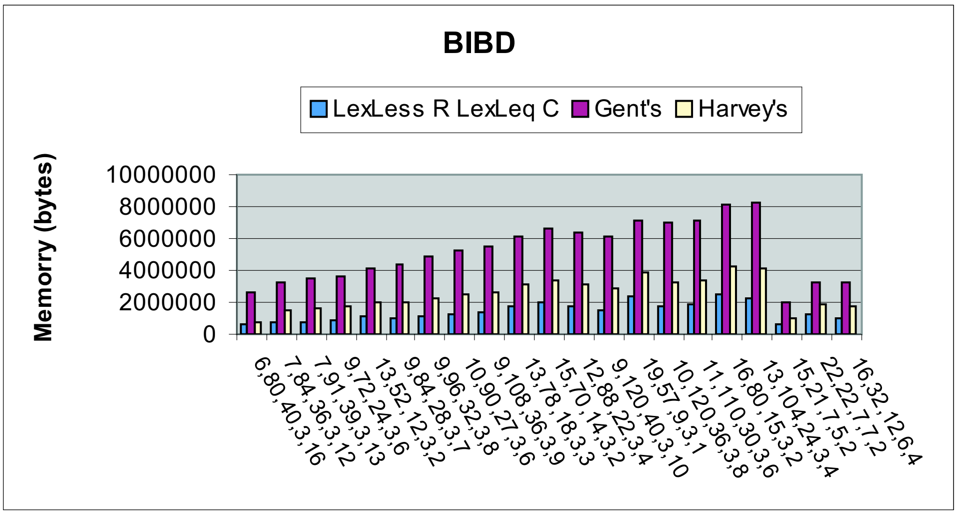

Figure 5.6: BIBD: `LexLeq`/`LexLess` vs constraint encodings in terms of memory usage.

In this section, we answer these questions by focusing on the implementation of $\leq_{lex}$.

In Solver, three propagation events are available for an integer variable:

- **whenValue:** The variable is assigned a value.
- **whenRange:** Either the minimum or the maximum element in the domain of the variable is modified.
- **whenDomain:** The domain of the variable is modified.

When $\vec{X} \leq_{lex} \vec{Y}$ is GAC, every value a in $\mathcal{D}(X_\alpha)$ is supported by $max(Y_\alpha)$, and also by $\langle min(X_{\alpha+1}), \ldots, min(X_{min\{n,\beta\}-1})\rangle$ and $\langle max(Y_{\alpha+1}), \ldots, max(Y_{min\{n,\beta\}-1})\rangle$ if $a = max(Y_\alpha)$. In a similar way, every value a in $\mathcal{D}(Y_\alpha)$ is supported by $min(X_\alpha)$, and also by $\langle min(X_{\alpha+1}), \ldots, min(X_{min\{n,\beta\}-1})\rangle$ and $\langle max(Y_{\alpha+1}), \ldots, max(Y_{min\{n,\beta\}-1})\rangle$ if $a = min(X_\alpha)$. Any modification to the bounds of the variables in the range $[\alpha, min\{n, \beta\})$ should wake up the constraint. Therefore, we attach **whenRange** propagation event to all the variables in $[\alpha, min\{n, \beta\})$ using the initial values of the pointers. This is in fact not satisfactory for two reasons. First, we want to wake up the constraint only when $min(X_i)$ or $max(Y_i)$ of some i in $[\alpha, min\{n, \beta\})$ changes. Thanks to the domain-delta facility[2] of Solver, we can discard the events triggered by modifications to $max(X_i)$ or $min(Y_i)$. Second, as variables are assigned the pointers α and β move inwards. The constraint therefore can be woken up by events triggered outside the current $[\alpha, min\{n, \beta\})$. We can, however, easily discard the events triggered by the variables outside the range of interest.

When do we propagate the constraint? In Solver, there are two ways:

1. respond to each propagation event individually, i.e. propagate the constraint after every event;
2. wait until all propagation events attached to the constraint accumulate and then propagate all at once.

[2] The domain-delta is a special set where the modifications of the domain of a variable are stored. This domain-delta can be accessed during the propagation of the constraints posted on the variable.

The first method is often used with a highly incremental filtering algorithm, in which most of the data structures can be restored easily and efficiently at each propagation step. In this case, the global constraint calls a special algorithm for its very first propagation to initialise the data structures, but calls another algorithm in future propagations which builds the data structures based on their previous state. The second method is often used when the filtering algorithm is costly, such as when maintaining the data structures incrementally is more costly than computing them from scratch.

In the case of our global constraint $\leq_{lex}$, the best choice at first sight seemed to be using method 1 for a number of good reasons. First, the filtering algorithm of $\leq_{lex}$ is incremental by construction. Second, we have only pointers (and flags) in terms of data structure and they can be maintained incrementally in an easy and efficient way. Third, we have `Initialise` which initialises the pointers and performs an initial propagation of the constraint, and `LexLeq(`i`)` which propagates the constraint in future propagation events and keeps the pointers up to date if necessary. With this motivation, we implemented our global constraint using the first method of propagation.

On the other hand, we do not know how multiple events are scheduled in Solver and we do not have any control over the event scheduling system. When `LexLeq(`i`)` is triggered, the pointers might be pointing to the wrong indices if several other propagation events occurred by the time X_i or Y_i was modified. In particular, α might be too far left and β might be too far right. This is not a big problem since Solver guarantees that constraint propagation triggered by every propagation event will eventually (before Solver makes another labelling decision) take place. Hence, α and β will eventually be set to their correct values. This, however, means that there may be many unnecessary propagations which may result in increased run-times. For instance, consider the vectors:

$$\begin{array}{rcllllll}
\vec{X} & = & \langle\{0,1\}, & \{0,1\}, & \{0,1\}, & \{0,1\}\rangle & & \\
\vec{Y} & = & \langle\{0,1\}, & \{0,1\}, & \{0,1\}, & \{0\}\rangle & & \\
 & & \quad\uparrow \alpha & & & & & \uparrow \beta
\end{array}$$

constrained as $\vec{X} \leq_{lex} \vec{Y}$. We have $\alpha = 0$ and $\beta = \infty$. Assume that due to the other constraints on the variables of $\vec{X}$, the following modifications occurred simultaneously:

$$\begin{array}{rcllllll}
\vec{X} & = & \langle\{0,1\}, & \{\xcancel{0},1\}, & \{\xcancel{0},1\}, & \{\xcancel{0},1\}\rangle & & \\
\vec{Y} & = & \langle\{0,1\}, & \{0,1\}, & \{0,1\}, & \{0\}\rangle & & \\
 & & \quad\uparrow \alpha & & & & & \uparrow \beta
\end{array}$$

These modifications all affect the minimums of some variables below α, and hence we have three propagation events E_1, E_2, and E_3 triggering `LexLeq(1)`, `LexLeq(2)`, and `LexLeq(3)`, respectively. Let us analyse what happens if the events are scheduled in the order E_1, E_2 and then E_3. With the call `LexLeq(1)`, β is not updated because we have $1 \neq \beta - 1$ and $min(X_1) \leq max(Y_1)$. Moreover, no pruning takes place because we have $\alpha < 1 < \beta$. For similar reasons, `LexLeq(2)` returns without any prunings nor update of β. Finally, the call `LexLeq(3)` updates β to 1 and then ensures AC$(X_\alpha < Y_\alpha)$.

While propagating the constraint, we have to deal with events that have no impact. If we instead have a non-incremental algorithm and use the second method of propagation, we would wait until all the events accumulate and respond only once by doing essentially what `LexLeq(3)` did. With the hope that waiting for all events to accumulate could result in faster running times, we slightly changed the filtering algorithm to obtain a non-incremental algorithm and used method 2 for implementing the propagation mechanism.

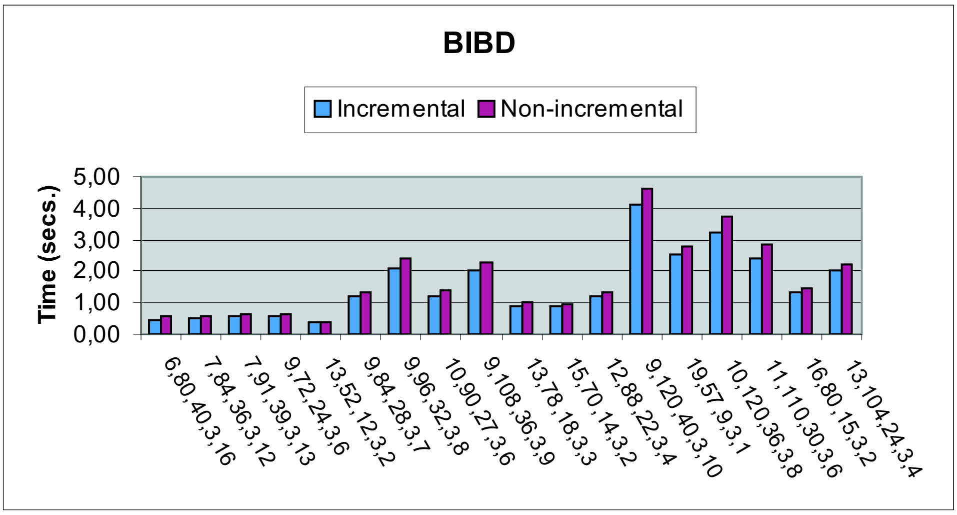

Figure 5.7: BIBD: Incremental vs non-incremental algorithm in terms of run-times.

In order to compare the two algorithms, we ran some experiments on the BIBD problem by using the same model, labelling heuristic, and the lexicographic ordering constraints described in Section 5.9.1. We pose the lexicographic ordering constraints by using either the incremental or the non-incremental implementations. Surprisingly, the former proved to be faster than latter. This hints that even if we have many propagation events, the efficiency of the original algorithm can overcome the cost of delaying propagation events and maintaining a propagation queue. Figure 5.7 shows the results.

In the rest of this section, we first explain how we implemented the incremental algorithm and then show how we can easily transform it into a non-incremental algorithm and implement it using the second method of propagation.

5.10.1 Incremental Algorithm

We first implement **post** which is called by Solver when the constraint is first posted. This procedure initialises the data structures (e.g. the pointers), defines on which events the constraint propagates, and then propagates the constraint. Since the data structures are maintained incrementally, they are defined in Solver as reversible objects. That is, their values will be restored automatically by Solver when it backtracks. Instead of implementing the propagation algorithm using **propagate**, we post different demons on the variables of $\vec{X}$ and $\vec{Y}$ to be able to propagate selectively.

Procedure post

```
1     $i := 0$;
      ⋮
9     if $\alpha = \beta$ then fail;
10    foreach $i \in [\alpha, min\{n, \beta\})$ do
10.1      $X_i$.whenRange(EventDemonForX($i$));
10.2      $Y_i$.whenRange(EventDemonForY($i$));
      end
11    LexLeq($\alpha$);
```

`Initialise` in Algorithm 1 is in fact what **post** deploys except that after initialing the pointers and before calling the filtering algorithm, we attach an event demon to a **whenRange** event for each variable of the vectors in the range $[\alpha, min\{n, \beta\})$.

`EventDemonForX(`i`)` is triggered whenever $min(X_i)$ or $max(X_i)$ in the initial range $[\alpha, min\{n, \beta\})$ is modified. The demon then propagates the constraint. `LexLeq(`i`)` in Algorithm 2 is in fact what `EventDemonForX(`i`)` deploys; however, before propagation, events that are not of interest are filtered.

Procedure `EventDemonForX(`i`)`

```
1      if $X_i$.getMinDelta()$\neq 0$ then
1.1        if $\alpha \leq i < \beta$ then
1.1.1          LexLeq($i$);
           end
       end
```

In Solver, $\delta(V)$ (reads as domain-delta) is a special set where the modifications of $\mathcal{D}(V)$ are stored. The member function **getMinDelta()** returns $min(X_i) - min(\delta(X_i))$. To know whether the propagation event that triggered the demon is due to a change to $min(X_i)$, it suffices to test the value of X_i.**getMinDelta()**. If the test returns 0 then $min(X_i)$ has not been modified.

`EventDemonForY(`i`)` is similar to `EventDemonForX(`i`)` except that we now discard the events triggered by modifications to $min(Y_i)$, using the member function **getMaxDelta()**.

Procedure `EventDemonForY(`i`)`

```
1      if $X_i$.getMaxDelta()$\neq 0$ then
1.1        if $\alpha \leq i < \beta$ then
1.1.1          LexLeq($i$);
           end
       end
```

We attach `EventDemonForX` or `EventDemonForY` to a **whenRange** event for each variable in $[\alpha, min\{n, \beta\})$ using the initial values of the pointers. As the variables are assigned, β moves monotonically left. Some index i which was initially less significant than β could in the future be more significant. Hence, if a propagation event is triggered at this index then it can be discarded.

Ideally, no propagation events at indices more significant than α can pop up because all the variables above α are already ground. Unfortunately, due to the propagation of the constraint or the presence of other constraints, we may have multiple modifications triggering multiple events simultaneously. This may result in events which are invoked at more significant indices than the current value of α and which are still waiting in the event queue. As an example, consider:

$$\begin{array}{rcllll} \vec{X} & = & \langle\{0,1\}, & \{0,1\}, & \{0,1\}, & \{0,1\}\rangle \\ \vec{Y} & = & \langle\{1\}, & \{1\}, & \{1\}, & \{1,2\}\rangle \\ & & \ \uparrow \alpha & & & & \uparrow \beta \end{array}$$

constrained as $\vec{X} \leq_{lex} \vec{Y}$. We have $\alpha = 0$ and $\beta = \infty$. Assume that $\sum_i X_i = 4$, that is $\vec{X}$ is constrained to have four 1s. The propagation of the sum constraint removes all the zeros in $\vec{X}$:

$$\begin{array}{rclllll}
\vec{X} & = & \langle\{\xcancel{0},1\}, & \{\xcancel{0},1\}, & \{\xcancel{0},1\}, & \{\xcancel{0},1\}\rangle & \\
\vec{Y} & = & \langle\{1\}, & \{1\}, & \{1\}, & \{1,2\}\rangle & \\
 & & \uparrow\alpha & & & & \uparrow\beta
\end{array}$$

Now we have 4 events invoking `EventDemonForX`. If `EventDemonForX(0)` is handled first then α is updated to 3. But we still have `EventDemonForX(1)` and `EventDemonForX(2)` in the propagation queue and they are invoked at more significant indices than α.

5.10.2 Non-incremental Algorithm

In this version of the algorithm, instead of maintaining the values of α and β incrementally, we recompute their values every time the constraint is propagated. Due to this nature, the pointers need not be defined reversible in Solver. We again post different demons on the variables of $\vec{X}$ and $\vec{Y}$ to be able to propagate selectively. However, the propagation algorithm is implemented using **propagate**, because every demon now delays its response and one propagation takes place after all the events are accumulated.

In **post**, we only need to initialise α to index 0 and attach an event demon to a **whenRange** event for every variable in the vectors.

Procedure post

```
$\alpha := 0$;
foreach $i \in [0, n)$ do
    $X_i$.whenRange(EventDemonForX($i$));
    $Y_i$.whenRange(EventDemonForY($i$));
end
```

We now use Solver's **push** to delay our response to each propagation event of interest.

Procedure `EventDemonForX(`i`)`

```
if $X_i$.getMinDelta()$\neq 0$ then
    if $i < \beta$ then
        push();
    end
end
```

Procedure `EventDemonForY(`i`)`

```
if $Y_i$.getMaxDelta()$\neq 0$ then
    if $i < \beta$ then
        push();
    end
end
```

Note that as the variables are assigned, α and β move inwards. Even if we recompute β at every propagation step, we can discard the events invoked at less significant indices than the previous value of β. Since all the variables before α are ground and we recompute

α at every propagation step, no propagation events at more significant indices than the previous value of α can pop up.

Solver automatically aggregates all such calls to **push** into a single invocation of the **propagate** procedure which propagates the constraint. Note that **propagate** is also automatically called once the constraint is posted.

Procedure propagate

```
1      while $\alpha < n \wedge X_\alpha \doteq Y_\alpha$ do $\alpha := \alpha + 1$;
2      if $\alpha = n$ then return;
3      else $i := \alpha$;
4      $\beta := -1$;
5      while $i \neq n \wedge min(X_i) \leq max(Y_i)$ do
5.1        if $min(X_i) = max(Y_i)$ then
5.1.1          if $\beta = -1$ then $\beta := i$;
           else
5.2.1          $\beta := -1$;
           end
5.3        $i := i + 1$;
       end
6      if $i = n$ then $\beta := \infty$;
7      else if $\beta = -1$ then $\beta := i$;
8      if $\alpha = \beta$ then fail;
9      else Prune;
```

In **propagate**, we first initialise α. We know that α moves monotonically right. Hence, line 1 traverses the vectors starting at the previous value of α. When looking for β, we have two choices: (1) as $\beta \geq \alpha$, we can start at α; (2) as β moves monotonically left, we can start at the previous value of β and traverse the vectors towards α. Using the second choice, we must *always* travel from the previous β to the current α to be sure to find the most significant index for β. However, using the first choice, we need not always travel all the way from the current α to the previous β: we may stop at an index $i < \beta$ because we have $min(X_i) > max(Y_i)$. If $\beta > \alpha$ then we call `Prune`.

Procedure `Prune`

```
1      if $\alpha + 1 = \beta$ then
1.1        AC($X_i < Y_i$);
       end
2      if $\alpha + 1 < \beta$ then
2.1        AC($X_i \leq Y_i$);
2.2        if $X_i \doteq Y_i$ then UpdateAlpha;
       end
```

propagate can be seen as a combination of Algorithms 1 and 2 as we first recompute α and β, and then prune the inconsistent values. The exception is that we are no longer interested in the index at which an event was triggered. This also requires a small modification to `UpdateAlpha`. Note that we no longer need a procedure to update β.

Procedure UpdateAlpha

```
$\alpha := \alpha + 1$;
if $\alpha = n$ then return;
if $\neg(X_i \doteq Y_i)$ then Prune;
else UpdateAlpha;
```

5.11 Summary

In this chapter, we have proposed two new global constraints for lexicographic orderings which are useful for breaking row and column symmetries of a matrix of decision variables.

There are at least two ways of decomposing a lexicographic ordering constraint. Both of the decompositions are inferior to maintaining GAC on the constraint. Combining the decompositions is in fact equivalent to maintaining GAC; however, this carries a penalty in the cost of constraint propagation. Alternatively, by using the domain size of the variables in the vectors, one can pose arithmetic inequality constraints to ensure lexicographic orderings. This approach is feasible only if the vectors are not too long and the domain size is not too large. We have therefore developed an efficient filtering algorithm which either proves that $\vec{X} \leq_{lex} \vec{Y}$ is disentailed, or ensures GAC on $\vec{X} \leq_{lex} \vec{Y}$. The algorithm runs in time $O(nb)$ where b is the cost of adjusting the bounds of a variable, but runs amortised in time $O(b)$. Since adjusting the bounds is a constant time operation, b is always a constant. The complexity of the algorithm is optimal as there are $O(n)$ variables to consider.

The filtering algorithm exploits two pointers which save us from repeatedly traversing the vectors. In the absence of such pointers, the worst-case complexity would be quadratic, as opposed to linear, because we would then have to traverse the whole vectors every time we propagate the constraint. These pointers are useful also for extending the algorithm to obtain new filtering algorithms. By slightly changing the definitions of the pointers, we can easily obtain similar algorithms for $\vec{X} <_{lex} \vec{Y}$ and $\vec{X} \neq \vec{Y}$. Moreover, the pointer approach makes it possible to detect entailment in a dual manner to detecting disentailment. Furthermore, the algorithm can easily be generalised for vectors of any length, as well as for vectors whose variables are repeated and shared.

We have studied the propagation of vector equality and disequality constraints using lexicographic ordering constraints, and demonstrated that decomposing a chain of lexicographic ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. We have also compared with related work.

Even though the filtering algorithm can in general do more pruning than propagating the lexicographic ordering constraint via its decompositions, in practice we may observe no major difference. Similarly, in some cases, posting the corresponding arithmetic constraint can solve the lexicographic ordering constraint as efficiently as the algorithm. In Table 5.7, we summarise the experiments conducted on three problem domains. An entry is marked as $-$ (resp. $+$) if the corresponding method of propagation is inferior (resp. superior) to the filtering algorithm. If we cannot decide which of the two is superior to the other then we mark the entry as $-+$. If the method behaves similarly to the filtering algorithm then the mark is $\approx$.

As the table shows, in most of the cases, the filtering algorithm is preferable to its decompositions or the corresponding arithmetic constraint except for some cases. When solving the progressive party problem, the $\wedge$ decomposition gives smaller search trees for some instances of the problem. This is due to the dynamic nature of the labelling

Approach	Progressive Party Problem	Template Design Problem	BIBD
$\vee$	$-$	$-$	$-$
$\wedge$	$-+$	$-$	$-+$
$\wedge\ \vee$	$-$	$\approx$	$-$
Arithmetic	$-$	$\approx$	$-$

Table 5.7: Summary of the experiments in Chapter 5.

heuristic chosen. Hence, we conclude that a decomposition method may be superior to a filtering algorithm with a dynamic labelling heuristic. In BIBDs, the labelling heuristic used is static; however, the $\wedge$ decomposition and the algorithm give the same search tree for almost all instances. On the other hand, the algorithmic approach solves the instances much quicker. Finally, propagating the lexicographic ordering constraints by combining the decompositions or the arithmetic constraint is as efficient as the filtering algorithm in the template design problem whose instances do not require long vectors with large domains.

We have also learnt some general lessons regarding the design and implementation of global constraints. First, if a data structure is easy to restore using its previous value, then it is more efficient to maintain it incrementally than computing it from scratch every time we need it. Second, if a filtering algorithm is incremental (i.e. its data structures can be maintained incrementally) and this comes with a low cost, then propagating the constraint is more efficient by responding to each propagation event individually than by responding only once after all events accumulate. Even if we have to deal with many propagation events, the efficiency of the filtering algorithm can overcome the cost of delaying events and maintaining a propagation queue. Therefore, when designing a new global constraint, we need to seek ways of maintaining the data structures incrementally in an easy way so that we can propagate our constraint very efficiently.

Chapter 6

Lexicographic Ordering with Sum Constraints

6.1 Introduction

Lexicographic ordering constraints are useful for breaking row and column symmetries. In this chapter, we take into account some additional constraints posted on the vectors constrained to be lexicographically ordered. In particular, we introduce two new global constraints `LexLeqAndSum` and `LexLessAndSum` on 0/1 variables. Each of the constraints is an ordering constraint and combines together a lexicographic ordering constraint with two sum constraints. Given two vectors $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1} \rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1} \rangle$, `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ ensures that $\vec{X} \leq_{lex} \vec{Y}$, and that $\sum_i X_i = Sx$ and that $\sum_i Y_i = Sy$; `LexLessAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ ensures that $\vec{X} <_{lex} \vec{Y}$, $\sum_i X_i = Sx$, and $\sum_i Y_i = Sy$. Lexicographic ordering and sum constraints on 0/1 variables frequently occur together in problems involving demand, capacity or partitioning that are modelled with symmetric matrices of decision variables. To post and propagate such combinations of constraints efficiently and effectively, we design global constraints for lexicographic ordering with sums, each of which encapsulates its own filtering algorithm.

Combining a lexicographic ordering constraint with two sum constraints is useful in two situations. First, it is of benefit when there is a very large space to explore, such as when the problem is not satisfiable. Second, it is useful when we lack a good labelling heuristic. Even if the labelling heuristic we have is not the best to solve the problem, earlier detection of dead-ends or inconsistent values can greatly decrease the search effort. One of the major disadvantages of posting ordering constraints to break variable symmetry is the possibility that the labelling heuristic and the symmetry breaking constraints clash. In this case, many solutions may be rejected as they do not agree with the ordering of the variables imposed by the symmetry breaking constraints, resulting in larger search trees and longer run-times compared to no symmetry breaking. A combined constraint gives additional pruning, and this can help compensate for the labelling heuristic trying to push the search in a different direction to the symmetry breaking constraints.

This chapter is organised as follows. We motivate in Section 6.2 the need of our new global constraints. In Section 6.3, we present a filtering algorithm for the constraint `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ assuming that the sums Sx and Sy are ground. Then in Section 6.4, we discuss the complexity of the algorithm, and prove that the algorithm is correct and complete. In Section 6.5, we show how we can easily modify the algorithm to obtain a filtering algorithm for `LexLessAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$, to detect entailment, and to deal with sums that are not ground but bounded. We demonstrate in Section 6.6 that decomposing a chain of our global constraints into constraints between adjacent or all pairs of vectors hinders constraint propagation. We discuss the related work in Section 6.7 and present our computational results in Section 6.8. Combining lexicographic ordering constraint with other constraints is discussed in Section 6.9. Finally, before summarising in Section 6.11, we give in Section 6.10 the details of the implementation.

6.2 Lexicographic Ordering with Sum Constraints

Lexicographic ordering and sum constraints on 0/1 variables frequently occur together in problems involving demand, capacity or partitioning that are modelled with symmetric matrices of decision variables.

Demand Constraints As an example for a problem involving demand, consider the steel mill slab design problem introduced in Chapter 3.2.2. Every order is to be packed onto exactly one slab. One way of modelling this problem is given in Figure 3.3. In this model, a 2-d 0/1 matrix O of decision variables is used to determine which orders are packed onto which slabs. Sum constraints on the columns of O ensure that every order is packed onto one slab:

$$\forall i \in \mathcal{O}rders \,.\, \sum_{j \in \mathcal{S}labs} O_{i,j} = 1$$

This matrix has partial column symmetry because orders of the same size and colour are indistinguishable. We break this symmetry by posting lexicographic ordering constraints. Given a set of indistinguishable orders $\{o_i, o_{i+1}, \ldots, o_j\}$, we enforce that the columns corresponding to such orders $\vec{O}_i, \vec{O}_{i+1}, \ldots, \vec{O}_j$ are lexicographically ordered:

$$\vec{O}_i \leq_{lex} \vec{O}_{i+1} \ldots \leq_{lex} \vec{O}_j$$

The indistinguishable columns are thus constrained by both lexicographic ordering and sum constraints.

A similar problem is the rack configuration problem introduced in Chapter 3.2.3. Every card is to be plugged into exactly one rack. In the second model of the problem shown in Figure 3.6, we manipulate the cards rather than the number of cards of a given type plugged into a rack. Hence, a 2-d 0/1 matrix C of decision variables is used to determine which cards are plugged into which racks. Sum constraints on the columns of C ensure that every card is plugged into one rack:

$$\forall i \in \mathcal{C}ards \,.\, \sum_{j \in \mathcal{R}acks} C_{i,j} = 1$$

Since the cards of the same type are indistinguishable, this model suffers from partial column symmetry. Given a set of indistinguishable cards $\{c_i, c_{i+1}, \ldots, c_j\}$, we enforce that the columns corresponding to such cards $\vec{C}_i, \vec{C}_{i+1}, \ldots, \vec{C}_j$ are lexicographically ordered:

$$\vec{C}_i \leq_{lex} \vec{C}_{i+1} \ldots \leq_{lex} \vec{C}_j$$

We again have both lexicographic ordering and sum constraints posted on the symmetric columns.

In fact, any assignment problem can be reformulated in terms of a 2-d matrix of variables with sum constraints. Suppose we have a set $\mathcal{V}$ of variables each taking a value from a set $\mathcal{W}$ of values. We can reformulate this problem by introducing a 2-d matrix A of $\mathcal{V} \times \mathcal{W}$, where each element takes a value between 0 and 1. A variable V_i is assigned w_j iff the corresponding entry in the matrix is assigned 1. Since every variable is to be assigned only one value, we insist that:

$$\forall i \in \mathcal{V} \,.\; \sum_{j \in \mathcal{W}} A_{i,j} = 1$$

Such a reformulation is desirable when we want to deploy ILP techniques for optimisation problems (see for instance the matrix model of the warehouse location problem in Chapter 3.2.3) or when we want to transform value symmetry into variable symmetry (see Chapter 4.4.3).

If the variables of the original formulation are symmetric then the 2-d matrix has column symmetry. To break this symmetry, we post lexicographic ordering constraints on the columns. Consequently, we have columns constrained by both lexicographic ordering and sum constraints.

Partitioning/Capacity Constraints As an example for a problem involving partitioning or capacity, consider the ternary Steiner problem introduced in Chapter 3.2.1. The problem is to partition n elements into $b = n(n-1)/6$ subsets of size 3. In Figure 3.2, we have shown one way of modelling this problem, where a 2-d matrix X of decision variables is used to represent which element goes into which subsets. Sum constraints on the columns of X ensure that every subset is of size 3:

$$\forall i \in \mathcal{B} \,.\; \sum_{j \in \mathcal{N}} X_{i,j} = 3$$

This model has column symmetry because the subsets are indistinguishable. Due to the problem constraints, no pair of subsets can be equal. We can therefore break subset symmetry by posting strict lexicographic ordering constraints on the columns:

$$\vec{X}_0 <_{lex} \vec{X}_1 \ldots <_{lex} \vec{X}_{b-1}$$

Now we have both strict lexicographic ordering and sum constraints posted on the columns.

As another example, consider the social golfers problem introduced in Chapter 3.2.4. The golfers must be partitioned into g groups in every week and every group must contain s golfers. Figure 3.8 shows a modification of the classical set variable based model given in Figure 3.7. In this model, a 3-d 0/1 matrix T of decision variables is used to decide which golfer plays in which group of which week. Since every week is a 2-d matrix, sum constraints are posted on the columns (i.e. the group dimension) of every week to ensure that every group is of size s:

$$\forall j \in \mathcal{W}eeks \,.\; \forall i \in \mathcal{G}roups \,.\; \sum_{k \in \mathcal{G}olfers} T_{i,j,k} = s$$

This model suffers from too many symmetries including the symmetry between the groups of golfers. The groups are indistinguishable and the contents of a group from one week to the next are independent of each other. Due to the problem constraints, the groups within a week must be disjoint. Hence, we can break this symmetry by insisting that the groups within each week are strict lexicographically ordered. That is, we enforce strict lexicographic ordering constraints on the columns of every week:

$$\forall j \in \mathcal{W}eeks\,.\ \vec{T}_{0,j} <_{lex} \vec{T}_{1,j} \ldots <_{lex} \vec{T}_{g-1,j}$$

Now, we have both strict lexicographic ordering and sum constraints posted on the columns of every week.

Demand and Partitioning/Capacity Constraints Some problems have both demand and partitioning requirements, like the BIBD problem introduced in Chapter 3.2.1. In this problem, we need to partition the elements of a set $\mathcal{V}$ into b subsets in such a way that every element appears in r subsets and every subset contains k elements. In Figure 3.1 we give a model of this problem, where a 2-d matrix X is used to determine which element goes into which subset. The sum constraints along the rows ensure that every element appears in r subsets:

$$\forall j \in \mathcal{V}\,.\ \sum_{i \in \mathcal{B}} X_{i,j} = r$$

while the sum constraints along columns impose that every subset contains k elements:

$$\forall i \in \mathcal{B}\,.\ \sum_{j \in \mathcal{V}} X_{i,j} = k$$

Both the subsets and the elements of $\mathcal{V}$ are indistinguishable. Therefore, X has both row and column symmetry. We break this symmetry by insisting that the rows and columns are lexicographically ordered. Due to the problem constraints, no pair of rows can be equal unless $r = \lambda$, so we can impose strict lexicographic ordering on the rows:

$$\vec{X}_0 \leq_{lex} \vec{X}_1 \ldots \leq_{lex} \vec{X}_{b-1}$$

$$\langle X_{0,0}, \ldots, X_{b-1,0}\rangle <_{lex} \langle X_{0,1}, \ldots, X_{b-1,1}\rangle \ldots <_{lex} \langle X_{0,v-1}, \ldots, X_{b-1,v-1}\rangle$$

Now we have both (strict) lexicographic ordering and sum constraints posted on the rows and columns.

As we have seen from the examples, a wide range of CSPs can be modelled using 0/1 matrices with both lexicographic ordering and sum constraints on the rows and/or columns. Given two vectors of 0/1 variables $\vec{X}$ and $\vec{Y}$, $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ ensures that $\vec{X} \leq_{lex} \vec{Y}$, and that $\sum_i X_i = Sx$ and that $\sum_i Y_i = Sy$. Since the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ need to be lexicographically ordered, $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ is an ordering constraint.

Theorem 32 $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ *is an ordering constraint.*

Proof: The set of solutions $\mathcal{S}'$ satisfying $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ is a subset of the set of solutions $\mathcal{S}$ satisfying $\vec{X} \leq_{lex} \vec{Y}$. As any subset of $\mathcal{S}$ is a totally ordered set, $\mathcal{S}'$ is a totally ordered set. QED.

Posting $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ is semantically equivalent to posting $\vec{X} \leq_{lex} \vec{Y}$, $\sum_i X_i = Sx$, and $\sum_i Y_i = Sy$. Operationally, a filtering algorithm which removes from the vectors those values that cannot be a part of any solution to $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ can lead to more pruning than the total pruning obtained by the filtering algorithms of the lexicographic ordering constraint and the sum constraint.

Theorem 33 *GAC(*`LexLeqAndSum`*$(\vec{X}, \vec{Y}, Sx, Sy)$) is strictly stronger than GAC($\vec{X} \leq_{lex} \vec{Y}$), GAC($\sum_i X_i = Sx$), and GAC($\sum_i Y_i = Sy$).*

Proof: GAC(`LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$) is as strong as GAC($\vec{X} \leq_{lex} \vec{Y}$), GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), because the former implies the latter. To show strictness, consider:

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

with $Sx = Sy = 1$. We have GAC($\vec{X} \leq_{lex} \vec{Y}$), GAC($\sum_i X_i = 1$), and GAC($\sum_i Y_i = 1$). The assignment $Y_2 \leftarrow 1$ forces $\vec{Y}$ to be $\langle 0,0,1\rangle$ which is lexicographically less than $min\{\vec{x} \mid \ \sum_i x_i = 1 \ \wedge \ \vec{x} \in \vec{X}\} = \langle 0,1,0\rangle$. Hence, GAC(`LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$) does not hold. QED.

A similar result holds also for the strict lexicographic ordering with sum constraints.

Theorem 34 *GAC(*`LexLessAndSum`*$(\vec{X}, \vec{Y}, Sx, Sy)$) is strictly stronger than GAC($\vec{X} <_{lex} \vec{Y}$), GAC($\sum_i X_i = Sx$), and GAC($\sum_i Y_i = Sy$).*

Proof: The example in Theorem 33 shows strictness. QED.

6.3 A Filtering Algorithm for Lexicographic Ordering with Sums

In this section, we present a filtering algorithm for the lexicographic ordering constraint together with two sum constraints for 0/1 variables. This algorithm either detects the disentailment of `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ or prunes inconsistent values so as to achieve GAC on `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$.

Assume Sx and Sy are ground. GAC on `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ requires us to maintain GAC on $\vec{X} \leq_{lex} \vec{Y}$ in the presence of Sx 1s and $n - Sx$ 0s placed on $\vec{X}$, as well as Sy 1s and $n - Sy$ 0s placed on $\vec{Y}$. Therefore, we need to decide:

1. with which variables of $\vec{X}$, the value 1 can be extended with $Sx - 1$ more 1s from the remaining variables of $\vec{X}$ to obtain a vector $\vec{x}$, and with Sy 1s from the variables of $\vec{Y}$ to obtain a vector $\vec{y}$, such that $\vec{x} \leq_{lex} \vec{y}$;

2. with which variables of $\vec{X}$, the value 0 can be extended with $n - Sx - 1$ more 0s (or equivalently with Sx 1s) from the remaining variables of $\vec{X}$ to obtain a vector $\vec{x}$, and with Sy 1s from the variables of $\vec{Y}$ to obtain a vector $\vec{y}$, such that $\vec{x} \leq_{lex} \vec{y}$;

3. with which variables of $\vec{Y}$, the value 0 can be extended with $n - Sy - 1$ more 0s (or equivalently with Sy 1s) from the remaining variables of $\vec{Y}$ to obtain a vector $\vec{y}$, and with Sx 1s from the variables of $\vec{X}$ to obtain a vector $\vec{x}$, such that $\vec{x} \leq_{lex} \vec{y}$;

4. with which variables of $\vec{Y}$, the value 1 can be extended with $Sy - 1$ more 1s from the remaining variables of $\vec{Y}$ to obtain a vector $\vec{y}$, and with Sx 1s from the variables of $\vec{X}$ to obtain a vector $\vec{x}$, such that $\vec{x} \leq_{lex} \vec{y}$.

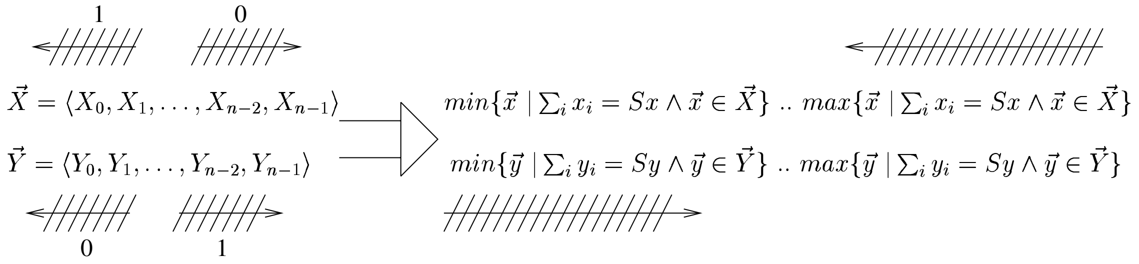
1
0
$\vec{X} = \langle X_0, X_1, \ldots, X_{n-2}, X_{n-1}\rangle$
$\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-2}, Y_{n-1}\rangle$
0
1
$min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ .. $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$
$min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ .. $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$

Figure 6.1: Prunings of the filtering algorithm of `LexLeqAndSum`.

The algorithm performs these decisions step by step, and in each step prunes the inconsistent values which cannot be extended in the way desired.

The prunings of the algorithm are depicted in Figure 6.1. Steps 1 and 2 remove 1s and 0s respectively from $\vec{X}$ to tighten the upper bound of $\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ until:

$$max\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$$

The support for the upper bound is also the support for all the other values in $\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$. Similarly, steps 3 and 4 remove 0s and 1s respectively from $\vec{Y}$ to tighten the lower bound of $min\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$ until:

$$min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$$

The support for the lower bound is also the support for all the other values in $\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.

Before giving the details of the algorithm in Section 6.3.3, we first sketch its main features in Section 6.3.1 on a running example and then in Section 6.3.2 provide the theoretical background from which the algorithm is derived.

6.3.1 A Worked Example

In each step, we maintain a pair of lexicographically minimal and maximal ground vectors $\vec{sx} = \langle sx_0, \ldots, sx_{n-1}\rangle$ and $\vec{sy} = \langle sy_0, \ldots, sy_{n-1}\rangle$, and a flag α where for all $0 \leq i < \alpha$ we have $sx_i = sy_i$ and $sx_\alpha \neq sy_\alpha$. That is, α is the most significant index where $\vec{sx}$ and $\vec{sy}$ differ. Additionally, we may need to know whether $\vec{sx}_{\alpha+1 \to n-1}$ and $\vec{sy}_{\alpha+1 \to n-1}$ are lexicographically ordered. Therefore, we introduce a boolean flag γ whose value is *true* iff $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$.

Consider the vectors:

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

and the constraint `LexLeqAndSum`$(\vec{X}, \vec{Y}, 3, 2)$. Our algorithm starts with step 1 in which we have:

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

where $\vec{sx} = min\{\vec{x} \mid \sum_i x_i = Sx - 1 \ \wedge \ \vec{x} \in \vec{X}\}$, $\vec{sy} = max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$, and $\vec{sx} \leq_{lex} \vec{sy}$. We check where we can place one more 1 in $\vec{sx}$ to make the sum $Sx = 3$ as required without disturbing $\vec{sx} \leq_{lex} \vec{sy}$. We have $\alpha = 0$ and $\gamma = true$. We can safely place 1 to the right of α as this does not affect $\vec{sx} \leq_{lex} \vec{sy}$. Since γ is $true$, placing 1 at α also does not affect the order of the vectors. Therefore, all the 1s in $\vec{X}$ can be extended with $Sx - 1$ more 1s from the remaining variables of $\vec{X}$ to obtain a vector $\vec{x}$, and with Sy 1s from the variables of $\vec{Y}$ to obtain a vector $\vec{y}$, such that $\vec{x} \leq_{lex} \vec{y}$.

In step 2 we have:

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

where $\vec{sx} = min\{\vec{x} \mid \sum_i x_i = Sx + 1 \ \wedge \ \vec{x} \in \vec{X}\}$, $\vec{sy}$ is as before, and $\vec{sx} >_{lex} \vec{sy}$. Note that $\sum_i sx_i = Sx + 1$ means that there are $n - Sx - 1$ 0s in $\vec{sx}$. We check where we can place one more 0 in $\vec{sx}$ to make the sum $Sx = 3$ as required and to obtain $\vec{sx} \leq_{lex} \vec{sy}$. We have $\alpha = 1$ and $\gamma = true$. Placing 0 to the left of α makes $\vec{sx}$ less than $\vec{sy}$. Since γ is $true$, placing 0 at α also makes $\vec{sx}$ less than $\vec{sy}$. Therefore, any 0 on the left side of α can be extended with $n - Sx - 1$ more 0s (or equivalently with Sx 1s) from the remaining variables of $\vec{X}$ to obtain a vector $\vec{x}$, and with Sy 1s from the variables of $\vec{Y}$ to obtain a vector $\vec{y}$, such that $\vec{x} \leq_{lex} \vec{y}$. This is not true, however, for the other 0s in the vector. Placing 0 to the right of α orders the vectors lexicographically the wrong way. Hence, we remove 0 from the domains of the variables of $\vec{X}$ to the right of α. The vector $\vec{X}$ is now $\langle\{0, 1\}, \{0, 1\}, \{0\}, \{0\}, \{1\}, \{1\}, \{0\}, \{0\}\rangle$.

In step 3 we have:

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

where $\vec{sx} = min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$, $\vec{sy} = max\{\vec{Y} \mid \sum_i y_i = Sy + 1 \ \wedge \ \vec{y} \in \vec{Y}\}$, and $\vec{sx} \leq_{lex} \vec{sy}$. Note that $\sum_i sy_i = Sy + 1$ means that there are $n - Sy - 1$ 0s in $\vec{sy}$. We check where we can place one more 0 in $\vec{sy}$ to make the sum $Sy = 2$ as required without disturbing $\vec{sx} \leq_{lex} \vec{sy}$. We have $\alpha = 0$ and $\gamma = true$. We can safely place 0 to the right of α as this does not affect $\vec{sx} \leq_{lex} \vec{sy}$. Since γ is $true$, placing 0 at α also does not affect the order of the vectors. Therefore, all the 0s in $\vec{Y}$ can be extended with $n - Sy - 1$ more 0s (or equivalently with Sy 1s) from the remaining variables of $\vec{Y}$ to obtain a vector $\vec{y}$, and with Sx 1s from the variables of $\vec{X}$ to obtain a vector $\vec{x}$, such that $\vec{x} \leq_{lex} \vec{y}$.

Finally, in step 4 we have:

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

where $\vec{sx}$ is as before, $\vec{sy} = max\{\vec{Y} \mid \sum_i y_i = Sy - 1 \ \wedge \ \vec{y} \in \vec{Y}\}$, and $\vec{sx} > \vec{sy}$. We check where we can place one more 1 in $\vec{sy}$ to make the sum $Sy = 2$ as required and to obtain $\vec{sx} \leq_{lex} \vec{sy}$. We have $\alpha = 1$ and $\gamma = true$. Placing 1 to the left of α makes $\vec{sx}$ less than $\vec{sy}$. Since γ is $true$, we can also safely place 1 at α. Therefore, any 1 on the left side of α can be extended with $Sy - 1$ more 1s from the remaining variables of $\vec{Y}$ to obtain a vector $\vec{y}$, and with Sx 1s from the variables of $\vec{X}$ to obtain a vector $\vec{x}$, such that $\vec{x} \leq_{lex} \vec{y}$. This is not true, however, for the other 1s in the vector. Placing 1 to the right of α makes $\vec{sx} >_{lex} \vec{sy}$. Hence, we remove 1 from the domains of the variables of $\vec{Y}$ to the right of α:

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

The constraint $\texttt{LexLeqAndSum}(\vec{X},\vec{Y},3,2)$ is now GAC.

6.3.2 Theoretical Background

Two theoretical results, which show when $\texttt{LexLeqAndSum}(\vec{X},\vec{Y},Sx,Sy)$ is disentailed and what conditions ensure GAC on $\texttt{LexLeqAndSum}(\vec{X},\vec{Y},Sx,Sy)$, form the foundations of the filtering algorithm.

A constraint is said to be *disentailed* when the constraint is $false$. We have shown in Theorem 8 that $\vec{X} \leq_{lex} \vec{Y}$ is disentailed when $\beta = \alpha$. By the definitions of these pointers, $\beta = \alpha$ iff $\texttt{floor}(\vec{X}) >_{lex} \texttt{ceiling}(\vec{Y})$. The lexicographic ordering constraint together with two sum constraints is disentailed in a condition similar to that of $\leq_{lex}$ with the difference that we now need to compare the smallest $\vec{x} \in \vec{X}$ and the largest $\vec{y} \in \vec{Y}$ where $\sum_i x_i = Sx$ and $\sum_i y_i = Sy$.

Theorem 35 *Given GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$),* $\texttt{LexLeqAndSum}$*($\vec{X}$, $\vec{Y}$, Sx, Sy) is disentailed iff $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.*

Proof: ($\Rightarrow$) Since $\texttt{LexLeqAndSum}(\vec{X},\vec{Y},Sx,Sy)$ is disentailed, any combination of assignments, including $\vec{X} \leftarrow min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $\vec{Y} \leftarrow max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$, does not satisfy $\vec{X} \leq_{lex} \vec{Y}$. Hence, $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.

($\Leftarrow$) Any $\vec{x} \in \{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ is lexicographically greater than any $\vec{y} \in \{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$. Hence, $\texttt{LexLeqAndSum}(\vec{X},\vec{Y},Sx,Sy)$ is disentailed. QED.

We now state the necessary conditions for $\texttt{LexLeqAndSum}(\vec{X},\vec{Y},Sx,Sy)$ to be GAC.

Theorem 36 *Given GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), GAC(*$\texttt{LexLeqAndSum}$*($\vec{X}$, $\vec{Y}$, Sx, Sy)) iff for all $0 \leq j < n$ and for all $k \in \mathcal{D}(X_j)$:*

$$min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.1)$$

and for all $k \in \mathcal{D}(Y_j)$:

$$min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.2)$$

Proof: ($\Rightarrow$) As the constraint is GAC, all values have support. In particular, $X_j \leftarrow k$ has a support $\vec{x_1} \in \{\vec{x} \mid \sum_i x_i = Sx \wedge x_j = k \wedge \vec{x} \in \vec{X}\}$ and $\vec{y_1} \in \{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$, where $\vec{x_1} \leq_{lex} \vec{y_1}$. Any $\vec{x_2} \in \{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{x_1}$ and $\vec{y_2} \in \{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\} \geq_{lex} \vec{y_1}$ support $X_j \leftarrow k$. In particular, $min\{\vec{x} \mid \sum_i x_i =$

$Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$ support $X_j \leftarrow k$. Hence, $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.

A dual argument holds for the variables of $\vec{Y}$. As the constraint is GAC, $Y_j \leftarrow k$ has a support $\vec{x_1} \in \{\vec{x} \mid \ \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ and $\vec{y_1} \in \{\vec{y} \mid \ \sum_i y_i = Sy \wedge y_j = k \wedge \vec{y} \in \vec{Y}\}$, where $\vec{x_1} \leq_{lex} \vec{y_1}$. Any $\vec{x_2} \in \{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{x_1}$ and $\vec{y_2} \in \{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\} \geq_{lex} \vec{y_1}$, in particular $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\}$ support $Y_j \leftarrow k$. Hence, $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\}$.

($\Leftarrow$) 6.1 and 6.2 ensure that for all $0 \leq j < n$, all the values in $\mathcal{D}(X_j)$ and $\mathcal{D}(Y_j)$ are supported, respectively. Hence, the constraint is GAC. QED.

On one hand, we can decide which values in the vectors are consistent by checking naively whether they satisfy 6.1 or 6.2. On the other hand, this would computationally be very expensive. Even though the variables have domain size 2, exploring every variable, constructing the necessary vectors and then comparing them lexicographically would take $O(n^2)$ time. We can, however, exploit our assumption that the domains of the vectors are $\{0, 1\}$ and decide for instance at which positions of $\vec{X}$, 1 is consistent without having to explore all the variables of $\vec{X}$ individually.

The following 8 theorems hold for a pair of vectors $\vec{X}$ and $\vec{Y}$ of 0/1 variables, where we show at which positions of $\vec{X}$ and $\vec{Y}$ we have (in-)consistent values. We will frequently need to call upon the following assumptions in what follows:

$$\vec{sx} = min\{\vec{x} \mid \ \sum_i x_i = Sx - 1 \wedge \vec{x} \in \vec{X}\} \ \wedge \ \vec{sy} = max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.3)$$

$$\vec{sx} = min\{\vec{x} \mid \ \sum_i x_i = Sx + 1 \wedge \vec{x} \in \vec{X}\} \ \wedge \ \vec{sy} = max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.4)$$

$$\vec{sx} = min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \wedge \vec{sy} = max\{\vec{y} \mid \ \sum_i y_i = Sy + 1 \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.5)$$

$$\vec{sx} = min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \wedge \vec{sy} = max\{\vec{y} \mid \ \sum_i y_i = Sy - 1 \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.6)$$

We start with 1s in $\vec{X}$.

Theorem 37 *Under assumption 6.3 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), if there is some index α in $[0, n)$ such that for all $0 \leq i < \alpha$ we have $sx_i = sy_i$ and $sx_\alpha < sy_\alpha$ then:*

1. *for all $0 \leq j < \alpha$ where $\mathcal{D}(X_j) = \{0, 1\}$, $1 \in \mathcal{D}(X_j)$ is inconsistent;*
2. *if $1 \in \mathcal{D}(X_\alpha)$ then 1 is consistent iff $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$;*
3. *for all $\alpha < j < n$ where $\mathcal{D}(X_j) = \{0, 1\}$, $1 \in \mathcal{D}(X_j)$ is consistent.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0, 1\}$ and $sx_j = 0$, we can get $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ x_j = 1 \ \wedge \ \vec{x} \in \vec{X}\}$ by replacing $sx_j = 0$ with $sx_j = 1$. Note that replacing $sx_j = 0$ with $sx_j = 1$ for the largest such j would also give $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ x_j = 1 \ \wedge \ \vec{x} \in \vec{X}\}$ for all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0, 1\}$ and $sx_j = 1$. If $j < \alpha$ then $min\{\vec{x} \mid \ \sum_i x_i = Sx \wedge x_j = 1 \wedge \vec{x} \in \vec{X}\} >_{lex} \vec{sy}$. In this case, $1 \in \mathcal{D}(X_j)$ is inconsistent by Theorem 36. We have $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ x_\alpha = 1 \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{sy}$ provided that $\vec{sx}_{\alpha+1\to n-1} \leq_{lex} \vec{sy}_{\alpha+1\to n-1}$. In this case, $1 \in \mathcal{D}(X_\alpha)$ is consistent by Theorem 36. Otherwise, we have $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_\alpha = 1 \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} \vec{sy}$, and $1 \in \mathcal{D}(X_\alpha)$ is inconsistent by Theorem 36. Finally, if $j > \alpha$ then $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 1 \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} \vec{sy}$, so $1 \in \mathcal{D}(X_j)$ is consistent by Theorem 36. QED.

Theorem 38 *Under assumption 6.3 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$),* `LexLeqAndSum`*($\vec{X}, \vec{Y}, Sx, Sy$) is disentailed if $\vec{sx} \geq_{lex} \vec{sy}$.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$ and $sx_j = 0$, we can get $min\{\vec{x} \mid \sum_i x_i = Sx \wedge x_j = 1 \wedge \vec{x} \in \vec{X}\}$ by replacing $sx_j = 0$ with $sx_j = 1$. Note that replacing $sx_j = 0$ with $sx_j = 1$ for the largest such j would also give $min\{\vec{x} \mid \sum_i x_i = Sx \wedge x_j = 1 \wedge \vec{x} \in \vec{X}\}$ for all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$ and $sx_j = 1$. For all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$, we have $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 1 \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} \vec{sx} \geq_{lex} \vec{sy}$. This means that $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} \vec{sy}$. By Theorem 35, the constraint is disentailed. QED.

Dual theorems hold for 0s in $\vec{X}$.

Theorem 39 *Under assumption 6.4 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), if there is some index α in $[0, n)$ such that for all $0 \leq i < \alpha$ we have $sx_i = sy_i$ and $sx_\alpha > sy_\alpha$ then:*

1. *for all $0 \leq j < \alpha$ where $\mathcal{D}(X_j) = \{0,1\}$, $0 \in \mathcal{D}(X_j)$ is consistent;*
2. *if $0 \in \mathcal{D}(X_\alpha)$ then 0 is consistent iff $\vec{sx}_{\alpha+1\to n-1} \leq_{lex} \vec{sy}_{\alpha+1\to n-1}$;*
3. *for all $\alpha < j < n$ where $\mathcal{D}(X_j) = \{0,1\}$, $0 \in \mathcal{D}(X_j)$ is inconsistent.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$ and $sx_j = 1$, we can get $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 0 \ \wedge \ \vec{x} \in \vec{X}\}$ by replacing $sx_j = 1$ with $sx_j = 0$. Note that replacing $sx_j = 1$ with $sx_j = 0$ for the smallest such j would also give $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 0 \ \wedge \ \vec{x} \in \vec{X}\}$ for all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$ and $sx_j = 0$. If $j < \alpha$ then $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 0 \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} \vec{sy}$. In this case, $0 \in \mathcal{D}(X_j)$ is consistent by Theorem 36. We have $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_\alpha = 0 \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{sy}$ provided that $\vec{sx}_{\alpha+1\to n-1} \leq_{lex} \vec{sy}_{\alpha+1\to n-1}$. In this case, $0 \in \mathcal{D}(X_\alpha)$ is consistent by Theorem 36. Otherwise, we have $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_\alpha = 0 \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} \vec{sy}$, and $0 \in \mathcal{D}(X_\alpha)$ is inconsistent by Theorem 36. Finally, if $j > \alpha$ then $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 0 \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} \vec{sy}$, so $0 \in \mathcal{D}(X_j)$ is inconsistent by Theorem 36. QED.

Theorem 40 *Under assumption 6.4 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), if $\vec{sx} \leq_{lex} \vec{sy}$ then for all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$, $0 \in \mathcal{D}(X_j)$ is consistent.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$ and $sx_j = 1$, we can get $min\{\vec{x} \mid \sum_i x_i = Sx \wedge x_j = 0 \wedge \vec{x} \in \vec{X}\}$ by replacing $sx_j = 1$ with $sx_j = 0$. Note that replacing $sx_j = 1$ with $sx_j = 0$ for the smallest such j would also give $min\{\vec{x} \mid \sum_i x_i = Sx \wedge x_j = 0 \wedge \vec{x} \in \vec{X}\}$ for all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$ and $sx_j = 0$. For all $0 \leq j < n$ where $\mathcal{D}(X_j) = \{0,1\}$, we have $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = 0 \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} \vec{sx} \leq_{lex} \vec{sy}$, so $0 \in \mathcal{D}(X_j)$ is consistent by Theorem 36. QED.

Very similar but also dual theorems hold for the (in-)consistent values in $\vec{Y}$.

Theorem 41 *Under assumption 6.5 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), if there is some index α in $[0, n)$ such that for all $0 \leq i < \alpha$ we have $sx_i = sy_i$ and $sx_\alpha < sy_\alpha$ then:*

1. *for all $0 \leq j < \alpha$ where $\mathcal{D}(Y_j) = \{0, 1\}$, $0 \in \mathcal{D}(Y_j)$ is inconsistent;*
2. *if $0 \in \mathcal{D}(Y_\alpha)$ then 0 is consistent iff $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$;*
3. *for all $\alpha < j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$, $0 \in \mathcal{D}(Y_j)$ is consistent.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 1$, we can get $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$ by replacing $sy_j = 1$ with $sy_j = 0$. Note that replacing $sy_j = 1$ with $sy_j = 0$ for the largest such j would also give $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$ for all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 0$. If $j < \alpha$ then $\vec{sx} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$. In this case, $0 \in \mathcal{D}(Y_j)$ is inconsistent by Theorem 36. We have $\vec{sx} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_\alpha = 0 \wedge \vec{y} \in \vec{Y}\}$ provided that $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$. In this case, $0 \in \mathcal{D}(Y_\alpha)$ is consistent by Theorem 36. Otherwise, we have $\vec{sx} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_\alpha = 0 \wedge \vec{y} \in \vec{Y}\}$, and $0 \in \mathcal{D}(Y_\alpha)$ is inconsistent by Theorem 36. Finally, if $j > \alpha$ then $\vec{sx} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$, so $0 \in \mathcal{D}(Y_j)$ is consistent by Theorem 36.

Theorem 42 *Under assumption 6.5 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) is disentailed if $\vec{sx} \geq_{lex} \vec{sy}$.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 1$, we can get $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$ by replacing $sy_j = 1$ with $sy_j = 0$. Note that replacing $sy_j = 1$ with $sy_j = 0$ for the largest such j would also give $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$ for all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 0$. For all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$, we have $\vec{sx} \geq_{lex} \vec{sy} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 0 \wedge \vec{y} \in \vec{Y}\}$. This means that $\vec{sx} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$. By Theorem 35, the constraint is disentailed. QED.

Theorem 43 *Under assumption 6.6 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), if there is some index α in $[0, n)$ such that for all $0 \leq i < \alpha$ we have $sx_i = sy_i$ and $sx_\alpha > sy_\alpha$ then:*

1. *for all $0 \leq j < \alpha$ where $\mathcal{D}(Y_j) = \{0, 1\}$, $1 \in \mathcal{D}(Y_j)$ is consistent;*
2. *if $1 \in \mathcal{D}(Y_\alpha)$ then 1 is consistent iff $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$;*
3. *for all $\alpha < j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$, $1 \in \mathcal{D}(Y_j)$ is inconsistent.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 0$, we can get $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$ by replacing $sy_j = 0$ with $sy_j = 1$. Note that replacing $sy_j = 0$ with $sy_j = 1$ for the smallest such j would also give $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$ for all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 1$. If $j < \alpha$ then $\vec{sx} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$. In this case, $1 \in \mathcal{D}(Y_j)$ is consistent by Theorem 36. We have $\vec{sx} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_\alpha = 1 \wedge \vec{y} \in \vec{Y}\}$ provided that $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$. In this case, $1 \in \mathcal{D}(Y_\alpha)$ is consistent by Theorem 36. Otherwise, we have $\vec{sx} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_\alpha = 1 \wedge \vec{y} \in \vec{Y}\}$, and $1 \in \mathcal{D}(Y_\alpha)$ is inconsistent by Theorem 36. Finally, if $j > \alpha$ then $\vec{sx} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$, so $1 \in \mathcal{D}(Y_j)$ is inconsistent by Theorem 36. QED.

Procedure GAC($\vec{X}, Sx$)

```
$min := \sum_i min(X_i)$;
$max := \sum_i max(X_i)$;
setMin($Sx, min$);
setMax($Sx, max$);
foreach $i \in [0, n)$ do
    setMin($X_i, min(Sx) - max + max(X_i)$);
    setMax($X_i, max(Sx) - min + min(X_i)$);
end
```

Theorem 44 *Under assumption 6.6 and GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), if $\vec{sx} \leq_{lex} \vec{sy}$ then for all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$, $1 \in \mathcal{D}(Y_j)$ is consistent.*

Proof: For all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 0$, we can get $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$ by replacing $sy_j = 0$ with $sy_j = 1$. Note that replacing $sy_j = 0$ with $sy_j = 1$ for the smallest such j would also give $max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$ for all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$ and $sy_j = 1$. For all $0 \leq j < n$ where $\mathcal{D}(Y_j) = \{0, 1\}$, we have $\vec{sx} \leq_{lex} \vec{sy} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge y_j = 1 \wedge \vec{y} \in \vec{Y}\}$, so $1 \in \mathcal{D}(Y_j)$ is consistent by Theorem 36. QED.

6.3.3 Algorithm Details

Based on the theoretical results given in the previous section, we have designed an efficient linear time algorithm, `LexLeqAndSum`, which either detects the disentailment of `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) or prunes inconsistent values so as to achieve GAC on `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$).

The algorithm is not incremental and therefore no data structure has to be initialised nor maintained incrementally. Every time the algorithm is called, all the data structures are recomputed and the generalised arc-consistent state is established. Therefore, when the constraint is first posted, `LexLeqAndSum` (and no initialisation algorithm) is called.

When `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) is GAC, the values in $\vec{X}$ and $\vec{Y}$ are supported by $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ and $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$, respectively. Any modification to the variables affecting $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ or $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ triggers the filtering algorithm. We cannot, however, decide what modifications can alter these vectors, because this all depends on the sums and on the current partial assignments of the vectors. For instance, removing 0 from the domain of some X_i changes $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ if $x_i = 0$ in the current $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$. Hence, the algorithm is called by the event handler every time any variable in the vectors is modified.

In Algorithm 23, we show the steps of `LexLeqAndSum`. The algorithm first establishes GAC on the sum constraints in lines 1 and 2 via the call `GAC`, which is based on Proposition 1 in [RR00]. Note that for 0/1 variables, BC is equivalent to GAC.

Proposition 1 **([RR00])** *Given $\vec{X}$ and an integer variable Sx, and the constraint $\sum_i X_i = Sx$ between these variables, BC($\sum_i X_i = Sx$) iff the following four conditions hold:*

1. *$min(Sx) \geq \sum_i min(X_i)$*

Algorithm 23: `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($\vec{X} \leq_{lex} \vec{Y} \wedge \sum_i X_i = Sx \wedge \sum_i Y_i = Sy$)
1      GAC($\vec{X}, Sx$);
2      GAC($\vec{Y}, Sy$);
3      $\vec{sx} := min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$;
4      $\vec{sy} := min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$;
5      if $\vec{sx} >_{lex} \vec{sy}$ then fail;
6      if $\exists i \in \{0, \ldots, n-1\} \,.\, |\mathcal{D}(X_i)| > 1$ then
6.1        $\vec{sx} := min\{\vec{x} \mid \sum_i x_i = Sx - 1 \wedge \vec{x} \in \vec{X}\}$;
6.2        PruneLeft($\vec{sx}, \vec{sy}, \vec{X}$);
6.3        GAC($\vec{X}, Sx$);
6.4        if $\exists i \in \{0, \ldots, n-1\} \,.\, |\mathcal{D}(X_i)| > 1$ then
6.4.1          $\vec{sx} := min\{\vec{x} \mid \sum_i x_i = Sx + 1 \wedge \vec{x} \in \vec{X}\}$;
6.4.2          PruneRight($\vec{sx}, \vec{sy}, \vec{X}$);
6.4.3          GAC($\vec{X}, Sx$);
           end
       end
7      if $\exists i \in \{0, \ldots, n-1\} \,.\, |\mathcal{D}(Y_i)| > 1$ then
7.1        $\vec{sx} := min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$;
7.2        $\vec{sy} := max\{\vec{y} \mid \sum_i y_i = Sy + 1 \wedge \vec{y} \in \vec{Y}\}$;
7.3        PruneLeft($\vec{sx}, \vec{sy}, \vec{Y}$);
7.4        GAC($\vec{Y}, Sy$);
7.5        if $\exists i \in \{0, \ldots, n-1\} \,.\, |\mathcal{D}(Y_i)| > 1$ then
7.5.1          $\vec{sy} := max\{\vec{y} \mid \sum_i y_i = Sy - 1 \wedge \vec{y} \in \vec{Y}\}$;
7.5.2          PruneRight($\vec{sx}, \vec{sy}, \vec{Y}$);
7.5.3          GAC($\vec{Y}, Sy$);
           end
       end
```

2. $max(Sx) \leq \sum_i max(X_i)$

3. $\forall i \in [0, n) \,.\, min(X_i) \geq min(Sx) - \sum_{j \neq i} max(X_j)$

4. $\forall i \in [0, n) \,.\, max(X_i) \leq max(Sx) - \sum_{j \neq i} min(X_j)$

Lines 3-5.2 of `GAC(`$\vec{X}, Sx$`)` implement the conditions 1-4 of Proposition 1 by calling `SetMin` and `SetMax` to tighten the lower and upper bounds, respectively. If no failure is encountered inside `GAC`, we check in line 5 of `LexLeqAndSum` whether $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$. If so, disentailment is detected and `LexLeqAndSum` terminates with failure. Otherwise ($min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$), we continue with 4 pruning steps.

In step 1 (lines 4, 6-6.3), we assume $\vec{X}$ is not ground and we are concerned with support for 1s in $\vec{X}$. We first construct $\vec{sx}$ as the minimum $\vec{X}$ with $\sum_i X_i = Sx - 1$ (line 6.1), and $\vec{sy}$ as the maximum $\vec{Y}$ with $\sum_i Y_i = Sy$ (line 4). We then determine

```
Procedure PruneLeft($\vec{sx}, \vec{sy}, \vec{V}$)
1       $\alpha := 0$;
2       while $sx_\alpha = sy_\alpha$ do $\alpha := \alpha + 1$;
3       $\gamma := false$;
4       if $\vec{sx}_{\alpha \to n-1} \leq_{lex} \vec{sy}_{\alpha \to n-1}$ then $\gamma := true$;
5       $i := 0$;
6       while $i < \alpha$ do
6.1         if $|\mathcal{D}(V_i)| > 1$ then
6.1.1           if $\vec{V} = \vec{X}$ then setMax($V_i, 0$);
6.1.2           else if $\vec{V} = \vec{Y}$ then setMin($V_i, 1$);
            end
6.2         $i := i + 1$;
        end
7       if $\neg\gamma \ \wedge \ |\mathcal{D}(V_\alpha)| > 1$ then
7.1         if $\vec{V} = \vec{X}$ then setMax($V_\alpha, 0$);
7.2         else if $\vec{V} = \vec{Y}$ then setMin($V_\alpha, 1$);
        end
```

where we can place one more 1 on $\vec{sx}$ and have $\vec{sx} \leq_{lex} \vec{sy}$ (via the call `PruneLeft` in line 6.2). Since $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{sy}$, we already have $\vec{sx} <_{lex} \vec{sy}$ in `PruneLeft`. Therefore, in lines 1 and 2 of `PruneLeft`, we first find the most significant index α where $sx_\alpha < sy_\alpha$. Placing 1 in $\vec{sx}$ to the left of α makes $\vec{sx} >_{lex} \vec{sy}$. This is true also for α provided that the subvectors after α are not lexicographically ordered. Anywhere after α has support since $\vec{sx} <_{lex} \vec{sy}$ remains valid. After deciding whether $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$ (lines 3-4), we prune all 1s in $\vec{X}$ to the left of α (lines 5-6.2) and at α if $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$ (lines 7-7.2). We then return back to `LexLeqAndSum`, maintain GAC($\sum_i X_i = Sx$) (via the call `GAC` in line 6.3), and continue with step 2.

In step 2 (lines 4, 6.4-6.4.3), we assume $\vec{X}$ is not ground and we are concerned with support for 0s in $\vec{X}$. We first construct $\vec{sx}$ as the minimum $\vec{X}$ with $\sum_i X_i = Sx + 1$ (line 6.4.1), and $\vec{sy}$ as the maximum $\vec{Y}$ with $\sum_i Y_i = Sy$ (line 4). We then determine where we can place one more 0 in $\vec{sx}$ and have $\vec{sx} \leq_{lex} \vec{sy}$ (via the call `PruneRight` in line 6.4.2). If $\vec{sx} \leq_{lex} \vec{sy}$ then placing another 0 on any location of $\vec{sx}$ makes $\vec{sx} < \vec{sy}$, so we do no pruning. If, however, we have $\vec{sx} >_{lex} \vec{sy}$ then placing 0 on $\vec{sx}$ to the right of α (the most significant index where $sx_\alpha > sy_\alpha$) does not change that $\vec{sx} > \vec{sy}$. 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 $\vec{sx} <_{lex} \vec{sy}$. We therefore first decide in `PruneRight` whether $\vec{sx} \leq_{lex} \vec{sy}$ (lines 1-2). If so, `PruneRight` returns. After deciding whether $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$ (lines 3-4), we prune all 0s in $\vec{X}$ to the right of α (lines 5-6.2) and at α if $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$ (lines 7-7.2). We then return back to `LexLeqAndSum`, and maintain GAC($\sum_i x_i = Sx$) (via the call `GAC` in line 6.4.3).

Step 3 (lines 7-7.4) is very similar to step 1, except we assume $\vec{Y}$ is not ground and identify support for the 0s in $\vec{Y}$. We first construct $\vec{sx}$ as the minimum $\vec{X}$ with $\sum_i X_i = Sx$ (line 7.1), and $\vec{sy}$ as the maximum $\vec{Y}$ with $\sum_i Y_i = Sy + 1$ (line 7.2). We then determine where we can place one more 0 in $\vec{sy}$ and have $\vec{sx} \leq_{lex} \vec{sy}$, in a way similar to that of step 1. Instead of pruning 1s, we now prune from $\vec{Y}$ those 0s which lack support. Due to this similarity, we can perform the prunings of the first and the third steps of the algorithm

```
Procedure PruneRight($\vec{sx}, \vec{sy}, \vec{V}$)
1     while $\alpha < n \ \wedge \ sx_\alpha = sy_\alpha$ do $\alpha := \alpha + 1$;
2     if $\alpha = n \ \vee \ sx_\alpha < sy_\alpha$ then return;
3     $\gamma := false$;
4     if $\vec{sx}_{\alpha \to n-1} \leq_{lex} \vec{sy}_{\alpha \to n-1}$ then $\gamma := true$;
5     $i := n - 1$;
6     while $i > \alpha$ do
6.1       if $|\mathcal{D}(V_i)| > 1$ then
6.1.1         if $\vec{V} = \vec{X}$ then setMin($V_i, 1$);
6.1.1         else if $\vec{V} = \vec{Y}$ then setMax($V_i, 0$);
          end
6.2       $i := i - 1$;
      end
7     if $\neg\gamma \ \wedge \ |\mathcal{D}(V_\alpha)| > 1$ then
7.1       if $\vec{V} = \vec{X}$ then setMin($V_\alpha, 1$);
7.2       else if $\vec{V} = \vec{Y}$ then setMax($V_\alpha, 0$);
      end
```

with the same procedure, `PruneLeft`. The input to the procedure is the vectors $\vec{sx}$ and $\vec{sy}$ and either of $\vec{X}$ and $\vec{Y}$. `PruneLeft` prunes either 1s from $\vec{X}$ (via the call `setMax`) or 0s from $\vec{Y}$ (via the call `setMin`) between the beginning of the vector and index α.

Step 4 (lines 7.1, 7.5-7.5.3) is very similar to step 2, except we assume $\vec{Y}$ is not ground and identify support for the 1s in $\vec{Y}$. We first construct $\vec{sx}$ as the minimum $\vec{X}$ with $\sum_i X_i = Sx$ (line 7.1), and $\vec{sy}$ as the maximum $\vec{Y}$ with $\sum_i Y_i = Sy - 1$ (line 7.5.1). We then determine where we can place one more 1 in $\vec{sy}$ and have $\vec{sx} \leq_{lex} \vec{sy}$, in a way similar to that of step 2. Instead of pruning 0s, we now prune from $\vec{Y}$ those 1s which lack support. Due to this similarity, we can perform the prunings of the second and the fourth steps of the algorithm with the same procedure, `PruneRight`. The input to the procedure is the vectors $\vec{sx}$ and $\vec{sy}$ and either of $\vec{X}$ and $\vec{Y}$. `PruneRight` prunes either 0s from $\vec{X}$ (via the call `setMin`) or 1s from $\vec{Y}$ (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 $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$. If yes then we fail; otherwise we have $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$. 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.

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 $\vec{sy}$ constructed in step 1. Similarly, in step 4 we can work on $\vec{sx}$ constructed in step 3. Second, from step 1 to step 2, we increase the number of 1s in $\vec{sx}$ by two and do not change $\vec{sy}$. In step 1, all 1s in $\vec{X}$ before X_α and perhaps the 1 at $\mathcal{D}(X_\alpha)$ are pruned. In step 2, α is either unchanged (i.e., 1 in $\mathcal{D}(X_\alpha)$ 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. A 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 $\vec{sx}$ and increase in $\vec{sy}$ by one. So α may move to the left and thus we must recompute α in step 3. A 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 $min\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$ and $max\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ without touching $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ and $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ which provide support for the values in the vectors (recall Figure 6.1). The exception is when a domain wipe-out occurs. As we have $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$, there is at least one value in the domain of each variable which is supported. This means that the prunings of the algorithm cannot cause any domain wipe-out.

6.4 Theoretical Properties

The filtering algorithm `LexLeqAndSum` runs in linear time in the length of the vectors.

Theorem 45 `LexLeqAndSum` *runs in time $O(n)$.*

Proof: `GAC` computes $\sum_i min(X_i)$ and $\sum_i max(X_i)$ in time $O(n)$. The computation of $\sum_{j \neq i} max(X_j)$ is constant time as it is equal to $\sum_j max(X_j) - max(X_i)$, and $\sum_j max(X_j)$ is already computed. Similarly, $\sum_{j \neq i} min(X_j)$ is equal to $\sum_j min(X_j) - min(X_i)$ and therefore computed in constant time. As each of $\sum_{j \neq i} max(X_j)$ and $\sum_{j \neq i} min(X_j)$ are computed for all $0 \leq i < n$, GAC on $\sum_i X_i = Sx$ can be achieved in $O(n)$. Each of `PruneLeft` and `PruneRight` runs in $O(n)$ as in the worst case the whole vectors need to traversed. Checking whether a vector is ground can be done in constant time by comparing $\sum_i min(X_i)$ and $\sum_i max(X_i)$ that are recomputed at every call to `GAC`. Hence, the algorithm runs in $O(n)$. QED.

The algorithm `LexLeqAndSum` is correct and complete.

Theorem 46 `LexLeqAndSum` *either establishes failure if* `LexLeqAndSum`*$(\vec{X}, \vec{Y}, Sx, Sy)$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC(*`LexLeqAndSum` *$(\vec{X}, \vec{Y}, Sx, Sy)$).*

Proof: `LexLeqAndSum` calls `PruneLeft` and `PruneRight`. We first give the pre- and post-conditions of these procedures, and show that they establish their post-conditions given the necessary pre-conditions.

`PruneLeft(`$\vec{sx}, \vec{sy}, \vec{X}$`)` must be invoked only when $\sum_i X_i = Sx$ is GAC, $\vec{X}$ is not ground, $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i = Sx{-}1 \wedge \vec{x} \in \vec{X}\}$, $\vec{sy}$ is $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$, and $\vec{sx} < \vec{sy}$. Similarly, `PruneLeft(`$\vec{sx}, \vec{sy}, \vec{Y}$`)` must be invoked only when $\sum_i Y_i = Sy$ is GAC, $\vec{Y}$ is not ground, $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$, $\vec{sy}$ is $max\{\vec{y} \mid \sum_i y_i = Sy{+}1 \wedge \vec{y} \in \vec{Y}\}$, and $\vec{sx} < \vec{sy}$. After execution of `PruneLeft(`$\vec{sx}, \vec{sy}, \vec{X}$`)` and `PruneLeft(`$\vec{sx}, \vec{sy}, \vec{Y}$`)`, all the inconsistent 1s and 0s from $\vec{X}$ and $\vec{Y}$ must be removed, respectively. Assuming that the pre-condition is satisfied, in lines 1 and 2 of `PruneLeft`, $\vec{sx}$ and $\vec{sy}$ are traversed until the most significant index α where $sx_\alpha < sy_\alpha$ is found. In lines 3 and 4, it is determined whether $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$. Lines 5 to 6.2 examine every variable more significant than α. If the variable is not yet ground, 1 is removed from its domain in the case of PruneLeft($\vec{sx}, \vec{sy}, \vec{X}$) but instead 0 in the case of PruneLeft($\vec{sx}, \vec{sy}, \vec{Y}$). A similar pruning happens at α (lines 7-7.2) provided that $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$. Consequently, all the inconsistent 1s and 0s from $\vec{X}$ and $\vec{Y}$ are removed respectively by Theorems 37 and 41. Hence, PruneLeft establishes its post-condition.

PruneRight($\vec{sx}, \vec{sy}, \vec{X}$) must be invoked only when $\sum_i X_i = Sx$ is GAC, $\vec{X}$ is not ground, $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i = Sx{+}1 \wedge \vec{x} \in \vec{X}\}$, and $\vec{sy}$ is $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$. Similarly, PruneRight($\vec{sx}, \vec{sy}, \vec{Y}$) must be invoked only when $\sum_i Y_i = Sy$ is GAC, $\vec{Y}$ is not ground, $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$, and $\vec{sy}$ is $max\{\vec{y} \mid \sum_i y_i = Sy - 1 \wedge \vec{y} \in \vec{Y}\}$. After execution of PruneRight($\vec{sx}, \vec{sy}, \vec{X}$) and PruneRight($\vec{sx}, \vec{sy}, \vec{Y}$), all the inconsistent 0s and 1s from $\vec{X}$ and $\vec{Y}$ must be removed, respectively. Assuming that the pre-condition is satisfied, line 1 of PruneRight traverses $\vec{sx}$ and $\vec{sy}$ until either it reaches the end of the vectors (because the vectors are equal), or it finds an index α where $sx_\alpha < sy_\alpha$ or $sx_\alpha > sy_\alpha$. If $\vec{sx} \leq_{lex} \vec{sy}$, all the 0s in $\vec{X}$ are consistent in the case of PruneRight($\vec{sx}, \vec{sy}, \vec{X}$) by Theorem 40. Similarly, all the 1s in $\vec{Y}$ are consistent in the case of PruneRight($\vec{sx}, \vec{sy}, \vec{Y}$) by Theorem 44. Hence, the algorithm returns in line 2. If, however, $\vec{sx} >_{lex} \vec{sy}$ then it is determined in lines 3 and 4 whether $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$. Lines 5 to 6.2 examine every variable less significant than α. If the variable is not yet ground, 0 is removed from its domain in the case of PruneRight($\vec{sx}, \vec{sy}, \vec{X}$) but instead 1 in the case of PruneRight($\vec{sx}, \vec{sy}, \vec{Y}$). A similar pruning happens at α (lines 7-7.2) provided that $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$. Consequently, all the inconsistent 0s and 1s from $\vec{X}$ and $\vec{Y}$ are removed respectively by Theorems 39 and 43. Hence, PruneRight establishes its post-condition.

We now analyse LexLeqAndSum. In lines 1 and 2, procedure GAC establishes GAC on the sum constraints based on Proposition 1. If $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ then LexLeqAndSum($\vec{X}, \vec{Y}, Sx, Sy$) is disentailed by Theorem 35 and thus LexLeqAndSum terminates with failure (line 5). Otherwise, we have $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$. This means that there is at least one value in the domain of each variable which is consistent. The algorithm therefore continues with 4 pruning steps and seeks support for a variable only if its domain is not a singleton.

Step 1 (lines 4, 6-6.3) PruneLeft($\vec{sx}, \vec{sy}, \vec{X}$) is called in line 6.2 satisfying its precondition: $\sum_i X_i = Sx$ is GAC (line 1), $\vec{X}$ is not ground (line 6), $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i = Sx - 1 \wedge \vec{x} \in \vec{X}\}$ (line 6.1), and $\vec{sy}$ is $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ (line 4). Moreover, $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ implies $\vec{sx} < \vec{sy}$ by Theorems 35 and 38. PruneLeft($\vec{sx}, \vec{sy}, \vec{X}$) removes all inconsistent 1s from $\vec{X}$. Finally, GAC($\sum_i X_i = Sx$) is restored in line 6.3.

Step 2 (lines 4, 6.4-6.4.3) PruneRight($\vec{sx}, \vec{sy}, \vec{X}$) is called in line 6.4.2 satisfying its precondition: $\sum_i X_i = Sx$ is GAC (line 6.3), $\vec{X}$ is not ground (line 6.4), $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i = Sx + 1 \wedge \vec{x} \in \vec{X}\}$ (line 6.4.1), and $\vec{sy}$ is $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ (line 4). PruneRight($\vec{sx}, \vec{sy}, \vec{X}$) removes all inconsistent 0s from $\vec{X}$. Finally, GAC($\sum_i X_i = Sx$) is restored in line 6.4.3.

Step 3 (lines 7-7.4) PruneLeft($\vec{sx}, \vec{sy}, \vec{Y}$) is called in line 7.3 satisfying its precondition: $\sum_i Y_i = Sy$ is GAC (line 2), $\vec{Y}$ is not ground (line 7), $\vec{sx}$ is $min\{\vec{x} \mid \sum_i x_i =$

$Sx \ \wedge \ \vec{x} \in \vec{X}\}$ (line 7.1), and $\vec{sy}$ is $max\{\vec{y} \mid \ \sum_i y_i = Sy + 1 \ \wedge \ \vec{y} \in \vec{Y}\}$ (line 7.2). Moreover, $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} >_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$ implies $\vec{sx} < \vec{sy}$ by Theorems 35 and 42. `PruneLeft(`$\vec{sx}, \vec{sy}, \vec{Y}$`)` removes all inconsistent 0s from $\vec{Y}$. Finally, GAC($\sum_i Y_i = Sy$) is restored in line 7.4.

Step 4 (lines 7.1, 7.5-7.5.3) `PruneRight(`$\vec{sx}, \vec{sy}, \vec{Y}$`)` is called in line 7.5.2 satisfying its precondition: $\sum_i Y_i = Sy$ is GAC (line 7.4), $\vec{Y}$ is not ground (line 7.5), $\vec{sx}$ is $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ (line 7.1), and $\vec{sy}$ is $max\{\vec{y} \mid \ \sum_i y_i = Sy - 1 \ \wedge \ \vec{y} \in \vec{Y}\}$ (line 7.5.1). `PruneRight(`$\vec{sx}, \vec{sy}, \vec{Y}$`)` removes all inconsistent 1s from $\vec{Y}$. Finally, GAC($\sum_i Y_i = Sy$) is restored in line 7.5.3.

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

6.5 Extensions

In this section, we extend the filtering algorithm `LexLeqAndSum` to cover some interesting cases. We first show in Section 6.5.1 how we can obtain a filtering algorithm for `LexLessAndSum`. Then, in Section 6.5.2, we explain how we catch entailment. Finally, in Section 6.5.3, we discuss how we deal with sums that are not ground but bounded.

6.5.1 Strict Lexicographic Ordering with Sum Constraints

`LexLeqAndSum` can easily be modified to obtain a filtering algorithm, `LexLessAndSum`, which either detects the disentailment of `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) or prunes inconsistent values so as to achieve GAC on `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$). To do so, we need to disallow equality between the vectors.

Before showing the modifications to `LexLeqAndSum`, we study the theoretical background of `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$). We start with the disentailment condition which is very similar to that of `LexLeqAndSum` except that equality between $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$ now disentails the constraint.

Theorem 47 *Given GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$),* `LexLessAndSum`*($\vec{X}$, $\vec{Y}$, Sx, Sy) is disentailed iff $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \geq_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.*

Proof: ($\Rightarrow$) Since `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) is disentailed, any combination of assignments, including $\vec{X} \leftarrow min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $\vec{Y} \leftarrow max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$, does not satisfy $\vec{X} <_{lex} \vec{Y}$. Hence, $min\{\vec{x} \mid \ \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \geq_{lex} max\{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.

($\Leftarrow$) Any $\vec{x} \in \{\vec{x} \mid \ \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ is lexicographically greater than or equal to any $\vec{y} \in \{\vec{y} \mid \ \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$. Hence, `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) is disentailed. QED.

The necessary conditions to ensure GAC on `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) are very similar to those of `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) except that the vectors supporting the values must now be strict lexicographically ordered.

Theorem 48 *Given GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$), GAC(*`LexLessAndSum`*($\vec{X}$, $\vec{Y}$, Sx, Sy)) iff for all $0 \leq j < n$ and for all $k \in \mathcal{D}(X_j)$:*

$$min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.7)$$

and for all $k \in \mathcal{D}(Y_j)$:

$$min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\} \quad (6.8)$$

Proof: ($\Rightarrow$) As the constraint is GAC, all values have support. In particular, $X_j \leftarrow k$ has a support $\vec{x_1} \in \{\vec{x} \mid \sum_i x_i = Sx \wedge x_j = k \wedge \vec{x} \in \vec{X}\}$ and $\vec{y_1} \in \{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$, where $\vec{x_1} <_{lex} \vec{y_1}$. Any $\vec{x_2} \in \{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{x_1}$ and $\vec{y_2} \in \{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\} \geq_{lex} \vec{y_1}$ support $X_j \leftarrow k$. In particular, $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$ support $X_j \leftarrow k$. Hence, $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ x_j = k \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.

A dual argument holds for the variables of $\vec{Y}$. As the constraint is GAC, $Y_j \leftarrow k$ has a support $\vec{x_1} \in \{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $\vec{y_1} \in \{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\}$, where $\vec{x_1} <_{lex} \vec{y_1}$. Any $\vec{x_2} \in \{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} \vec{x_1}$ and $\vec{y_2} \in \{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\} \geq_{lex} \vec{y_1}$, in particular $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\}$ support $Y_j \leftarrow k$. Hence, $min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ y_j = k \ \wedge \ \vec{y} \in \vec{Y}\}$.

($\Leftarrow$) 6.7 and 6.8 ensure that for all $0 \leq j < n$, all the values in $\mathcal{D}(X_j)$ and $\mathcal{D}(Y_j)$ are supported, respectively. Hence, the constraint is GAC. QED.

By exploiting the similarity between Theorems 36 and 48, we can easily decide at which positions of $\vec{X}$ and $\vec{Y}$ we have (in-)consistent values. In Theorems 37 to 44, we construct a pair of vectors $\vec{sx}$ and $\vec{sy}$ and decide where we can place one more 0/1 in $\vec{sx}/\vec{sy}$ and have $\vec{sx} \leq_{lex} \vec{sy}$. In Theorems 38 and 42, by placing one more 1 and 0 on $\vec{sx}$ and $\vec{sy}$ respectively, we obtain $\vec{sx} >_{lex} \vec{sy}$. This proves `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) is $false$, which in fact also establishes that `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) is disentailed. Similarly, in Theorems 40 and 44, by placing one more 0 and 1 on $\vec{sx}$ and $\vec{sy}$ respectively, we obtain $\vec{sx} <_{lex} \vec{sy}$. Since the ground vectors supporting 0 and 1 anywhere in $\vec{X}$ and $\vec{Y}$ are strict lexicographically ordered, 0 and 1 are consistent at every position of $\vec{X}$ and $\vec{Y}$.

In Theorems 37, 39, 41, and 43, by placing one more 0/1 in $\vec{sx}/\vec{sy}$, we obtain either of $\vec{sx} > \vec{sy}$, $\vec{sx} = \vec{sy}$, and $\vec{sx} < \vec{sy}$. In the first case, the support vectors are ordered lexicographically the wrong way so the value is inconsistent. In the third case, the support vectors are strict lexicographically ordered so the value is consistent. Since we do not want equality between $\vec{X}$ and $\vec{Y}$, we now declare that the value is inconsistent if we obtain $\vec{sx} = \vec{sy}$. Hence, we modify the second condition of Theorems 37, 39, 41, and 43 by stating that:

if $v \in \mathcal{D}(V_\alpha)$ then v is consistent iff $\vec{sx}_{\alpha+1 \rightarrow n-1} <_{lex} \vec{sy}_{\alpha+1 \rightarrow n-1}$

where $v \in \{0, 1\}$ and $\vec{V} \in \{\vec{X}, \vec{Y}\}$.

In summary, Theorems 38, 40, 42, and 44 are valid also for `LexLessAndSum`($\vec{X}, \vec{Y}, Sx, Sy$). Theorems 37, 39, 41, and 43 need only a small modification so that we ensure disequality between $\vec{X}$ and $\vec{Y}$. Putting this together with Theorem 47, we need only two modifications to the algorithm. First, we change the definition of γ in `PruneLeft` and `PruneRight`. The flag γ is *true* iff $\vec{sx}_{\alpha+1 \rightarrow n-1} <_{lex} \vec{sy}_{\alpha+1 \rightarrow n-1}$.

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

```
  ⋮
  $\gamma := false$;
  if $\vec{sx}_{\alpha \to n-1} <_{lex} \vec{sy}_{\alpha \to n-1}$ then $\gamma := true$;
  ⋮
```

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

```
  ⋮
  $\gamma := false$;
  if $\vec{sx}_{\alpha \to n-1} <_{lex} \vec{sy}_{\alpha \to n-1}$ then $\gamma := true$;
  ⋮
```

Algorithm 28: `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($\vec{X} \leq_{lex} \vec{Y} \wedge \sum_i X_i = Sx \wedge \sum_i Y_i = Sy$)

  ⋮
  $\vec{sx} := min\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$;
  $\vec{sy} := min\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$;
  if $\vec{sx} \geq_{lex} \vec{sy}$ then fail;
  ⋮
```

Second, we fail under the new disentailment condition. Algorithm 28 shows how we modify Algorithm 23.

6.5.2 Entailment

The importance of detecting entailment was discussed in Chapter 5.4.2. We thus introduce a Boolean flag called *entailed* which indicates whether `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is entailed. More formally:

Definition 34 *Given $\vec{X}$, $\vec{Y}$, and integers Sx and Sy, the flag* entailed *is set to true iff* `LexLeqAndSum`*$(\vec{X}, \vec{Y}, Sx, Sy)$ is true.*

We have shown in Theorem 17 that $\vec{X} \leq_{lex} \vec{Y}$ is *true* when $\gamma = \alpha$. By the definitions of these pointers, $\gamma = \alpha$ iff `ceiling`$(\vec{X}) \leq_{lex}$ `floor`$(\vec{X})$. The lexicographic ordering constraint together with two sum constraints is entailed in a condition similar to that of $\leq_{lex}$ with the difference that we now need to compare the largest $\vec{x} \in \vec{X}$ and the smallest $\vec{y} \in \vec{Y}$ where $\sum_i x_i = Sx$ and $\sum_i y_i = Sy$.

Theorem 49 *Given GAC($\sum_i X_i = Sx$) and GAC($\sum_i Y_i = Sy$),* `LexLeqAndSum`*($\vec{X}$, $\vec{Y}$, Sx, Sy) is entailed iff $max\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \ \wedge \ \vec{y} \in \vec{Y}\}$.*

Proof: ($\Rightarrow$) Since `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is entailed, any combination of assignments, including $\vec{X} \leftarrow max\{\vec{x} \mid \sum_i x_i = Sx \ \wedge \ \vec{x} \in \vec{X}\}$ and $\vec{Y} \leftarrow min\{\vec{y} \mid \sum_i y_i =$

```
Algorithm 29: Initialise
    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 : entailed is initialised, GAC($\vec{X} \leq_{lex} \vec{Y} \wedge \sum_i X_i = Sx \wedge \sum_i Y_i = Sy$)
    $entailed := false$;
    LexLeqAndSum;
```

$Sy \;\wedge\; \vec{y} \in \vec{Y}\}$, satisfies $\vec{X} \leq_{lex} \vec{Y}$. Hence, $max\{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\}$.

($\Leftarrow$) Any $\vec{x} \in \{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\}$ is lexicographically less than or equal to any $\vec{y} \in \{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\}$. Hence, `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is entailed. QED.

When `LexLeqAndSum` is called, `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ might have already been entailed due to the previous modifications or might be entailed due to the latest modifications occurred. Alternatively, `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ could be entailed after a pruning step of the algorithm. As depicted in Figure 6.1, `LexLeqAndSum` modifies $max\{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\}$ and $min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\}$. Even if we have $max\{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\} > min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\}$ when the algorithm is called, subsequent prunings might lead to $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\}$. For instance, assume that $\vec{X}$ and $\vec{Y}$ are constrained as `LexLeqAndSum`$(\vec{X}, \vec{Y}, 2, 2)$ and that `LexLeqAndSum` is called with the following state of the vectors:

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

We have $min\{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\} = \langle 0,1,0,0,0,1\rangle \leq_{lex} max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\} = \langle 1,0,1,0,0,0\rangle$ and $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} = \langle 1,1,0,0,0,0\rangle >_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\} = \langle 1,0,0,1,0,0\rangle$, so `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is neither disentailed nor entailed. The algorithm therefore proceeds with four pruning steps. In step 1, we have $\vec{sx} = \langle 0,0,0,0,0,1\rangle$, $\vec{sy} = \langle 1,0,1,0,0,0\rangle$, and $\alpha = 0$. No pruning is possible as $\vec{sx}_{\alpha+1\to n-1} \leq_{lex} \vec{sy}_{\alpha+1\to n-1}$. In step 2, we have $\vec{sx} = \langle 1,1,0,0,0,1\rangle$, $\vec{sy} = \langle 1,0,1,0,0,0\rangle$, and $\alpha = 1$. This means that 0 in $\mathcal{D}(X_1)$ is consistent as we have $\vec{sx}_{\alpha+1\to n-1} \leq_{lex} \vec{sy}_{\alpha+1\to n-1}$, but 0 in $\mathcal{D}(X_5)$ is inconsistent. The removal of 0 from $\mathcal{D}(X_5)$ gives $max\{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\} = \langle 1,0,0,0,0,1\rangle \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\} = \langle 1,0,0,1,0,0\rangle$. Now `LexLeqAndSum`$(\vec{X}, \vec{Y}, 2, 2)$ is entailed.

In Algorithm 30, we modify the filtering algorithm given in Algorithm 23 to detect entailment. We add line 0, where we return if `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ has already been entailed. Moreover, just before proceeding with the pruning steps in line 6, we check whether the latest modifications that triggered the algorithm resulted in entailment. Furthermore, after every pruning step and before proceeding ahead, we check entailment by comparing the current $max\{\vec{x} \mid \sum_i x_i = Sx \;\wedge\; \vec{x} \in \vec{X}\}$ and $min\{\vec{y} \mid \sum_i y_i = Sy \;\wedge\; \vec{y} \in \vec{Y}\}$. Whenever entailment is detected, we set *entailed* to *true* and return from the algorithm.

Finally, we need to initialise the flag *entailed* when `LexLeqAndSum` is first called. To distinguish between the first and the future calls to the algorithm, we introduce `Initialise` which is called when `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is first posted. `Initialise`

Algorithm 30: `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($\vec{X} \leq_{lex} \vec{Y} \wedge \sum_i X_i = Sx \wedge \sum_i Y_i = Sy$)
0    if entailed then return;
     ⋮
⇒    if $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ then
         entailed := true; return;
     end
6    if $\exists i \in \{0, \ldots, n-1\} . |\mathcal{D}(X_i)| > 1$ then
         ⋮
⇒        if $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ then
             entailed := true; return;
         end
6.4      if $\exists i \in \{0, \ldots, n-1\} . |\mathcal{D}(X_i)| > 1$ then
             ⋮
         end
     end
⇒    if $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ then
         entailed := true; return;
     end
7    if $\exists i \in \{0, \ldots, n-1\} . |\mathcal{D}(Y_i)| > 1$ then
         ⋮
⇒        if $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ then
             entailed := true; return;
         end
7.5      if $\exists i \in \{0, \ldots, n-1\} . |\mathcal{D}(Y_i)| > 1$ then
             ⋮
         end
     end
⇒    if $max\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ then
         entailed := true; return;
     end
```

initialises *entailed* to $false$ and then calls `LexLeqAndSum`.

Detecting entailment requires constructing a pair of additional vectors of length n and then checking whether they are lexicographically ordered. In the worst case, the vectors need to be examined from the beginning to the end. Hence, the runtime of the algorithm remains $O(n)$.

6.5.3 Bounded Sums

So far we have assumed that the sums on the vectors (i.e. Sx and Sy) are ground, and designed `LexLeqAndSum` based on this assumption. An exception is `GAC(`$\vec{X}, Sx$`)`, which maintains GAC on $\sum_i X_i = Sx$ given $\vec{X}$ and an integer variable Sx. How do we cope with sums that are not ground but bounded in the filtering algorithm of $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$?

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, second for the values in the range of $lx..ux$, and third for the values in the range of $ly..uy$. In the first part, since we seek support by minimising $\vec{X}$ and maximising $\vec{Y}$, we can run our algorithm `LexLeqAndSum` with $\sum_i X_i = lx$ and $\sum_i Y_i = uy$. This will provide the best support for the values in the vectors. In the second part, we tighten the upper bound of Sx until:

$$max\{\vec{x} \mid \sum_i x_i = ux \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = uy \ \wedge \ \vec{y} \in \vec{Y}\}$$

The support for the upper bound of Sx is also the support for all the other values in the domain of Sx. In the third part, we tighten the lower bound of Sy until:

$$min\{\vec{x} \mid \sum_i x_i = lx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \sum_i y_i = ly \ \wedge \ \vec{y} \in \vec{Y}\}$$

The support for the lower bound of Sy is also the support for all the other values in the domain of Sy.

The prunings of the second and third part do not require any calls back to the first part because we tighten only ux and ly without touching lx and uy. The exception is when there is a domain wipe-out. We reach the second and the third part given that the first part does not fail and we have $min\{\vec{x} \mid \sum_i x_i = lx \wedge \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = uy \ \wedge \ \vec{y} \in \vec{Y}\}$. Hence, no domain wipe-out can occur.

Consider tightening the upper bound of Sx. We first construct $\vec{sx}$ as $max\{\vec{x} \mid \sum_i x_i = ux \ \wedge \ \vec{x} \in \vec{X}\}$, and $\vec{sy}$ as $max\{\vec{y} \mid \sum_i y_i = uy \ \wedge \ \vec{y} \in \vec{Y}\}$. If $\vec{sx} \leq_{lex} \vec{sy}$ then ux is supported. If, however, $\vec{sx} >_{lex} \vec{sy}$ then we need to compute the largest ux with which we have $\vec{sx} \leq_{lex} \vec{sy}$. We can do this by first recording the most significant index α where $sx_\alpha > sy_\alpha$. This is where $sx_\alpha = 1$ and $sy_\alpha = 0$. Second, we replace the 1s at sx_α and below sx_α by 0s if the corresponding variables in $\vec{X}$ contain 0 in their domains. Finally, if we still have $\vec{sx} >_{lex} \vec{sy}$ after the replacements, we replace one more 1 above sx_α by a 0. This will definitely give us $\vec{sx} \leq_{lex} \vec{sy}$. Note that in order to keep $\vec{sx}$ maximal, we start replacing 1s by 0s below α. Let us illustrate this on an example. Consider $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ where:

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

and $2 \leq Sx \leq 6$ and $Sy = 2$. The corresponding maximal vectors are:

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

We have $\vec{sx} >_{lex} \vec{sy}$ and $\alpha = 3$. We need to decrease the number of 1s in $\vec{sx}$ while keeping $\vec{sx}$ as the maximum vector in a way that we obtain $\vec{sx} \leq_{lex} \vec{sy}$. To have $\vec{sx} \leq_{lex} \vec{sy}$, 1 at sx_α needs to replaced with 0:

$$\begin{array}{rcl} \vec{sx} & = & \langle 1,\ 0,\ 1,\ \overset{0}{\cancel{1}},\ 0,\ 1,\ 1,\ 1\rangle \\ \vec{sy} & = & \langle 1,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0\rangle \\ & & \qquad\qquad\quad \alpha \uparrow \end{array}$$

To keep $\vec{sx}$ maximal, we need to replace all the 1s beneath α by 0 if the domain of the corresponding variable contains 0. So we start at $\alpha + 1$, move towards the end of the vectors, and replace $sx_i = 1$ with $sx_i = 0$ if $\mathcal{D}(X_i) = \{0, 1\}$. We might have $\vec{sx}_{\alpha \to n-1} >_{lex} \vec{sy}_{\alpha \to n-1}$ after all the replacements. During the traversal, we record whether $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$:

$$\begin{array}{rcllllllll} & & & & & 0 & & & & 0 \\ \vec{sx} & = & \langle 1, & 0, & 1, & \cancel{1}, & 0, & 1, & 1, & \cancel{1}\rangle \\ \vec{sy} & = & \langle 1, & 0, & 1, & 0, & 0, & 0, & 0, & 0\rangle \\ & & & & & \alpha \uparrow & & & & \end{array}$$

We now know that $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$. To obtain $\vec{sx} \leq_{lex} \vec{sy}$, we eliminate another 1 above α. To keep $\vec{sx}$ maximal, we replace 1 at $sx_{\alpha-1}$ by 0:

$$\begin{array}{rcllllllll} & & & & 0 & 0 & & & & 0 \\ \vec{sx} & = & \langle 1, & 0, & \cancel{1}, & \cancel{1}, & 0, & 1, & 1, & \cancel{1}\rangle \\ \vec{sy} & = & \langle 1, & 0, & 1, & 0, & 0, & 0, & 0, & 0\rangle \\ & & & & & \alpha \uparrow & & & & \end{array}$$

Now we have $\vec{sx} \leq_{lex} \vec{sy}$. Since we decreased the number of 1s in $\vec{sx}$ by 3, the new upper bound of Sx is $6 - 3 = 3$.

Replacing a 1 above α by a 0 makes $\vec{sx} < \vec{sy}$. If $\mathcal{D}(X_\alpha)$ does not contain 0 or eliminating the 1s beneath α gave $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$, then it suffices to replace only one more 1 above α by 0 to obtain $\vec{sx} < \vec{sy}$. The location of such a replacement is the least significant index above α where 0 is in the domain of the corresponding variable. If, however, we obtain $\vec{sx}_{\alpha \to n-1} \leq_{lex} \vec{sy}_{\alpha \to n-1}$ after the replacements then we have the maximum number of 1s placed in $\vec{sx}$ and $\vec{sx} \leq_{lex} \vec{sy}$.

The new upper bound is guaranteed to be at least lx because running `LexLeqAndSum` with $\sum_i X_i = lx$ and $\sum_i Y_i = uy$ in the first part ensures $max\{\vec{x} \mid \sum_i x_i = lx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} max\{\vec{y} \mid \sum_i y_i = uy \ \wedge \ \vec{y} \in \vec{Y}\}$ (recall Figure 6.1).

In Procedure `TightenSx`, we show how we compute the new upper bound $newux$ of Sx, given the existing upper bound ux, $\vec{sx} = max\{\vec{x} \mid \sum_i x_i = ux \ \wedge \ \vec{x} \in \vec{X}\}$, $\vec{sy} = max\{\vec{y} \mid \sum_i y_i = uy \ \wedge \ \vec{y} \in \vec{Y}\}$, and $\vec{sx} >_{lex} \vec{sy}$. Lines 1 and 2 record the position α where $sx_\alpha = 1$ and $sy_\alpha = 0$. At line 3, we start counting the number of times we replace a 1 with a 0. If $|\mathcal{D}(X_\alpha)| > 1$ then 1 at sx_α can be replaced by 0, so we first increment the counter (line 5.1) and then assign -1 to γ so that we check later whether the subvectors beneath α are lexicographically ordered (line 5.2). In lines 6-7.3, we examine the values of $\vec{sx}$ between $\alpha + 1$ and $n - 1$. If we can replace a 1 with a 0 then we increment the counter (lines 7.1-7.1.2). We check in lines 7.2-7.2.2 whether the subvectors between $\alpha + 1$ and the current location are lexicographically ordered. The flag γ here saves us from traversing the subvectors. If $\mathcal{D}(X_\alpha)$ does not contain 0 or eliminating the 1s beneath α gave $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$, then it suffices to replace only one more 1 above α by 0 to obtain $\vec{sx} < \vec{sy}$. Line 8 increments the counter by 1 in a such a case. Finally, since $count$ is the number of 1s eliminated, the new upper bound of Sx is computed as $ux - count$ in line 9.

In a similar way, we can tighten the lower bound of Sy (see Procedure `TightenSy`). We first construct $\vec{sx}$ as $min\{\vec{x} \mid \sum_i x_i = lx \ \wedge \ \vec{x} \in \vec{X}\}$, and $\vec{sy}$ as $min\{\vec{y} \mid \sum_i y_i = ly \ \wedge \ \vec{y} \in \vec{Y}\}$. If $\vec{sx} \leq_{lex} \vec{sy}$ then ly is supported. Otherwise α is the most significant index where $sx_\alpha > sy_\alpha$. This is where $sx_\alpha = 1$ and $sy_\alpha = 0$. We now need to increase

```
Procedure Integer TightenSx($\vec{sx}, \vec{sy}, ux, newux$)
$\alpha := 0$;
while $sx_\alpha = sy_\alpha$ do $\alpha := \alpha + 1$;
$count := 0$;
$\gamma := -2$;
if $|\mathcal{D}(X_\alpha)| > 1$ then
    $count := count + 1$;
    $\gamma := -1$;
end
$i := \alpha + 1$;
while $i < n$ do
    if $sx_i = 1 \ \wedge \ |\mathcal{D}(X_i)| > 1$ then
        $sx_i = 0$;
        $count := count + 1$;
    end
    if $\gamma = -1$ then
        if $sx_i > sy_i$ then $\gamma := 1$;
        if $sx_i < sy_i$ then $\gamma := 0$;
    end
    $i := i + 1$;
end
if $\gamma = 1 \ \vee \ |\mathcal{D}(X_\alpha)| = 1$ then $count := count + 1$;
$newux := ux - count$;
```

the number of 1s in $\vec{sy}$ while keeping $\vec{sy}$ as the minimum vector so that $\vec{sx} \leq_{lex} \vec{sy}$. To have $\vec{sx} \leq_{lex} \vec{sy}$, 0 at sy_α needs to be replaced by 1 if $\mathcal{D}(Y_\alpha)$ contains 1. To keep $\vec{sy}$ minimal, we need to replace all the 0s beneath α by 1 if the domain of the corresponding variable contains 1. So we start at $\alpha+1$, move towards the end of the vectors, and replace $sy_i = 0$ with $sy_i = 1$ if $\mathcal{D}(Y_i) = \{0, 1\}$. We might have $\vec{sx}_{\alpha \to n-1} >_{lex} \vec{sy}_{\alpha \to n-1}$ after all the replacements. During the traversal, we record whether $\vec{sx}_{\alpha+1 \to n-1} \leq_{lex} \vec{sy}_{\alpha+1 \to n-1}$. Replacing a 0 above α by a 1 makes $\vec{sx} < \vec{sy}$. If $\mathcal{D}(Y_\alpha)$ does not contain 1 or eliminating the 0s beneath α gave $\vec{sx}_{\alpha+1 \to n-1} >_{lex} \vec{sy}_{\alpha+1 \to n-1}$, then it suffices to replace only one more 0 above α by 1 to obtain $\vec{sx} < \vec{sy}$. If, however, we obtain $\vec{sx}_{\alpha \to n-1} \leq_{lex} \vec{sy}_{\alpha \to n-1}$ after the replacements then we have the minimum number of 1s placed in $\vec{sy}$ and $\vec{sx} \leq_{lex} \vec{sy}$.

The new lower bound is guaranteed to be at most uy because running LexLeqAndSum with $\sum_i X_i = lx$ and $\sum_i Y_i = uy$ in the first part ensures $min\{\vec{x} \mid \ \sum_i x_i = lx \ \wedge \ \vec{x} \in \vec{X}\} \leq_{lex} min\{\vec{y} \mid \ \sum_i y_i = uy \ \wedge \ \vec{y} \in \vec{Y}\}$ (recall Figure 6.1).

Dealing with bounded sums does not change the complexity of the filtering algorithm. The first part deploys LexLeqAndSum which runs in time $O(n)$. In the second and third parts, we construct a pair of vectors of length n and in the worst case we traverse the whole vectors to decide the new bounds of the sums. Using γ, we do not need to scan the subvectors between $\alpha + 1$ and the current location of the traversal. The runtime of the filtering algorithm therefore remains $O(n)$.

Procedure `Integer TightenSy`($\vec{sx}, \vec{sy}, ly, newly$)

```
α := 0;
while $sx_\alpha = sy_\alpha$ do $\alpha := \alpha + 1$;
$count := 0$;
$\gamma := -2$;
if $|\mathcal{D}(Y_\alpha)| > 1$ then
    $count := count + 1$;
    $\gamma := -1$;
end
$i := \alpha + 1$;
while $i < n$ do
    if $sy_i = 0 \ \wedge \ |\mathcal{D}(Y_i)| > 1$ then
        $sy_i = 1$;
        $count := count + 1$;
    end
    if $\gamma = -1$ then
        if $sx_i > sy_i$ then $\gamma := 1$;
        if $sx_i < sy_i$ then $\gamma := 0$;
    end
    $i := i + 1$;
end
if $\gamma = 1 \ \vee \ |\mathcal{D}(Y_\alpha)| = 1$ then $count := count + 1$;
$newly := ly + count$;
```

6.6 Multiple Vectors

We often have multiple rows of a matrix that are lexicographically ordered and are constrained by sum constraints (see the examples in Section 6.2). We can treat such a problem as a single global constraint over the whole matrix. Alternatively, we can decompose it into lexicographic ordering with sum constraints on all pairs of rows, or (further still) onto ordering constraints just on neighbouring pairs of rows. The following results show that such decompositions hinder constraint propagation. However, we identify a special case which occurs in a number of applications where it does not. This case is when the sums are just 1, which occurs in many assignment problems.

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

Theorem 50 *GAC(*`LexLeqAndSum`*($\vec{X}_i, \vec{X}_j, S_i, S_j$)) for all $0 \leq i < j \leq n-1$ is strictly stronger than GAC(*`LexLeqAndSum`*($\vec{X}_i, \vec{X}_{i+1}, S_i, S_{i+1}$)) for all $0 \leq i < n-1$.*

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

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

with $S_0 = S_1 = S_2 = 2$. We have GAC(`LexLeqAndSum`($\vec{X}_i, \vec{X}_{i+1}, S_i, S_{i+1}$)) for all $0 \leq i < 2$. The assignment $X_{0,0} \leftarrow 1$ forces $\vec{X}_0$ to be $\langle 1, 1, 0, 0\rangle$, which is lexicographically greater than $max\{\vec{x} \mid \sum_i x_i = 2 \wedge \vec{x} \in \vec{X}_2\} = \langle 1, 0, 1, 0\rangle$. Therefore, GAC(`LexLeqAndSum`$(\vec{X}_0, \vec{X}_2, S_0, S_2)$) does not hold. QED.

Theorem 51 *GAC(*`LexLessAndSum`*$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) for all $0 \leq i < j \leq n-1$ is strictly stronger than GAC(*`LexLessAndSum`*$(\vec{X}_i, \vec{X}_{i+1}, S_i, S_{i+1})$) for all $0 \leq i < n-1$.*

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

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

with $S_0 = S_1 = S_2 = 2$. We have GAC(`LexLessAndSum`$(\vec{X}_i, \vec{X}_{i+1}, S_i, S_{i+1})$) for all $0 \leq i < 2$. The assignment $X_{0,0} \leftarrow 1$ forces $\vec{X}_0$ to be $\langle 1, 0, 1, 0, 0\rangle$, which is lexicographically greater than $max\{\vec{x} \mid \sum_i x_i = 2 \ \wedge \ \vec{x} \in \vec{X}_2\} = \langle 1, 0, 0, 1, 0\rangle$. Therefore, GAC(`LexLessAndSum`$(\vec{X}_0, \vec{X}_2, S_0, S_2)$) does not hold. QED.

Theorem 52 *GAC($\forall ij \ 0 \leq i < j \leq n-1$.* `LexLeqAndSum`*$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) is strictly stronger than GAC(*`LexLeqAndSum`*$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) for all $0 \leq i < j \leq n-1$.*

Proof: GAC($\forall ij \ 0 \leq i < j \leq n-1$. `LexLeqAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) is as strong as GAC(`LexLeqAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) for all $0 \leq i < j \leq n-1$. To show strictness, consider the following 3 vectors:

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

with $S_0 = 2$, $S_1 = 3$, and $S_2 = 3$. We have GAC(`LexLeqAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) for all $0 \leq i < j \leq 2$. The assignment $X_{0,0} \leftarrow 1$ forces $\vec{X}_0$ to be $\langle 1, 0, 1, 0, 0, 0\rangle$, which is supported by $\vec{X}_1$ only with the assignment $\langle 1, 1, 0, 1, 0, 0\rangle$. The latter assignment is, however, lexicographically greater than $max\{\vec{x} \mid \sum_i x_i = 3 \ \wedge \ \vec{x} \in \vec{X}_2\} = \langle 1, 1, 0, 0, 1, 0\rangle$. Therefore, GAC($\forall ij \ 0 \leq i < j \leq 2$. `LexLeqAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) does not hold. QED.

Theorem 53 *GAC($\forall ij \ 0 \leq i < j \leq n-1$.* `LexLessAndSum`*$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) is strictly stronger than GAC(*`LexLessAndSum`*$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) for all $0 \leq i < j \leq n-1$.*

Proof: GAC($\forall ij \ 0 \leq i < j \leq n-1$. `LexLessAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) is as strong as GAC(`LexLessAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) for all $0 \leq i < j \leq n-1$, because the former implies the latter. To show strictness, consider 7 vectors $\vec{X}_0$ to $\vec{X}_6$ with $\vec{X}_i = \langle\{0,1\}, \{0,1\}, \{0,1\}, \{0,1\}\rangle$ and $S_i = 2$ for all $i \in [0, 6]$. For all $0 \leq i < j \leq 6$, we have GAC(`LexLessAndSum`$(\vec{X}_i, \vec{X}_j, S_i, S_j)$) but there is no globally consistent solution as there are only $\binom{4}{2}= 6$ possible distinct vectors. QED.

Theorem 54 *GAC($\forall ij \ 0 \leq i < j \leq n-1$.* `LexLessAndSum`*$(\vec{X}_i, \vec{X}_j, 1, 1)$) is equivalent to GAC(*`LexLessAndSum`*$(\vec{X}_i, \vec{X}_j, 1, 1)$) for all $0 \leq i < j \leq n-1$.*

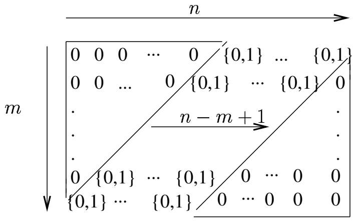

Figure 6.2: GAC($\texttt{LexLessAndSum}(\vec{X}_i, \vec{X}_j, 1, 1)$) for all $0 \leq i < j \leq n-1$.

Proof: ($\Leftarrow$) The domains of the variables when we have GAC($\texttt{LexLessAndSum}(\vec{X}_i, \vec{X}_j, 1, 1)$) for all $0 \leq i < j \leq n-1$ on an $n \times m$ matrix is shown in Figure 6.2. If $m > n$ then no solution is allowed. If $m = n$ then only one solution, namely the top-right to bottom-left diagonal matrix, is allowed. If $m < n$ then each 1 in every row is supported by the other 1s along its top-right to bottom-left diagonal, as well as the other 0s along its row. This also gives support for each 0 in every row. QED.

6.7 Related Work

A number of other constraints have been combined together to give new global constraints. For example, Régin and Rueher have introduced a global constraint which combines together sum and difference constraints [RR00]. More specifically, they have proposed an efficient filtering algorithm for constraint optimisation problems where a vector $\vec{X}$ of variables are subject to the difference constraints of the form $X_j - X_i \leq c$ and the objective function is defined by a sum $\sum_i X_i = Y$. Such constraints and the objective function occur together in deterministic scheduling problems where mean flow time or tardiness is the optimality criteria. The filtering algorithm can yield significant gains in the amount of pruning compared to propagating the difference constraints after each reduction of the bound of Y. Unfortunately, no experimental results are given, so it is difficult to judge how useful the global constraint is in practice. As a second example, following [FHK$^+$02], Carlsson and Beldiceanu have combined together a chain of lexicographic ordering constraints [CB02a] so that we post a single constraint on n vectors as opposed to $n-1$ of them. Even though in theory more constraint propagation can occur with this new global constraint, in practice there may be no gain. For a detailed discussion, see Section 6.9.

More generally, Bessière and Régin have defined a schema for enforcing GAC on an arbitrary conjunction of constraints [BR98]. This form of local consistency is called *conjunctive consistency* and the schema is derived by extending GAC-schema [BR97] which is a general framework for AC algorithms for constraints of arbitrary arity. The proposed filtering algorithm to achieve conjunctive consistency is based on the instantiation of GAC-schema for constraints given as a predicate. Since the semantics of the predicate is not known in such a general setting, finding a support for a value a of a variable V is done by searching for a solution for the problem derived when $V \leftarrow a$ by means of a search algorithm. Hence, the worst case complexity of the algorithm is bounded by the complexity of the search procedure used to seek supports. However, we here exploit the semantics of the global constraint, and therefore enforce GAC more efficiently. To be precise, by taking into account that the vectors should satisfy $\vec{X} \leq_{lex} \vec{Y} \wedge \sum_i X_i = Sx \wedge \sum_i Y_i = Sy$ and considering that the domains of the vectors are $\{0, 1\}$, we decide at which positions of $\vec{X}$ and $\vec{Y}$ the values 0 and 1 are inconsistent without having to explore all the variables of

$\vec{X}$ and $\vec{Y}$ individually and to seek support for every variable.

6.8 Experimental Results

We implemented our global constraints `LexLeqAndSum` and `LexLessAndSum` in C++ using ILOG Solver 5.3 [ILO02]. The global constraints encapsulate the corresponding filtering algorithm that either maintains GAC on (strict) lexicographic ordering with two sum constraints or establishes failure at each node of the search tree.

We performed a wide range of experiments to test whether our algorithm(s) reduces search significantly and is useful in practice. In the experiments, we have a matrix of 0/1 variables where the rows and/or columns are subject to sum constraints and also symmetric. To break the symmetry, we can pose lexicographic ordering constraints on the corresponding rows and/or columns. Due to the presence of the sum constraints, we can alternatively post lexicographic ordering with sum constraints. In the next two paragraphs, we show results comparing our new global constraints with the associated lexicographic ordering constraints developed in Chapter 5 when searching for a solution. Note that whenever we post lexicographic ordering with sum constraints, we no longer post sum constraints.

We tested our global constraints on two problem domains: the ternary Steiner problem, which originates from the computation of hypergraphs in combinatorial mathematics, and the balanced incomplete block design problem, which is a standard combinatorial problem from design theory with applications in cryptography and experimental design.

The results of the experiments are shown in tables where a "-" means no result is obtained in 1 hour (3600 secs). Whilst the number of choice points gives the number of alternatives explored in the search tree, the number of fails gives the number of incorrect decisions at choice points. The best result of each entry in a table is typeset in bold. If posing lexicographic ordering on the rows is done via a technique called $Tech$ then we write $Tech$ R. Similarly, posing lexicographic ordering on the columns using $Tech$ is specified as $Tech$ C, and on the rows and columns as $Tech$ RC. In theory posing lexicographic ordering constraints between every pair of rows (similarly for columns) leads to more pruning than posing between adjacent rows (see Chapter 5.6 and Section 6.6). We could not see any evidence of this in practice, therefore lexicographic ordering constraints are enforced just between the adjacent rows.

We used ILOG Solver 5.3 on a 1GHz pentium III processor with 256 Mb RAM under Windows XP.

Ternary Steiner Problem This was introduced in Chapter 3.2.1. In Figure 3.2, one way of modelling the problem is given. Since the subsets are indistinguishable, we can freely permute the subsets of X 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.

To break the row symmetry, we enforce that the rows $\vec{R_0}, \vec{R_1}, \ldots, \vec{R}_{n-1}$ corresponding to the n elements are anti-lexicographically ordered: $\vec{R_0} \geq_{lex} \vec{R_1} \ldots \geq_{lex} \vec{R}_{n-1}$. Due to the intersection constraint between any two subsets, no pair of columns can be equal. We therefore break the column symmetry by insisting that the columns corresponding to the $b = n(n-1)/6$ subsets of $\mathcal{V}$ are strict anti-lexicographically ordered: $\vec{C_0} >_{lex} \vec{C_1} \ldots >_{lex}$

n	No symmetry breaking			`LexLess` C			`LexLessAndSum` C		
				`LexLeq` R			`LexLeq` R		
	Fails	Choice	Time	Fails	Choice	Time	Fails	Choice	Time
		points	(secs.)		points	(secs.)		points	(secs.)
6	6,195	6,194	0.3	14	13	0	**11**	**10**	**0**
7	6	20	0	2	16	0	**1**	**15**	**0**
8	-	-	-	741	740	0.1	**390**	**389**	**0.1**
9	4,521	4,545	0.4	336	360	0.1	**250**	**274**	**0.1**
10	-	-	-	723,210	723,209	128.8	**433,388**	**433,387**	**105.3**

Table 6.1: Ternary Steiner problem: `LexLess` vs `LexLessAndSum` with column-wise labelling.

$\vec{C}_{b-1}$. Since the columns are also subject to sum constraints, we pose the lexicographic ordering constraints on the columns using either `LexLessAndSum` or `LexLess`.

We tried many different labelling heuristics, and obtained the best results by instantiating the matrix along its columns from left to right, and exploring the domain of each variable in *decreasing* order (i.e. 1 first and then 0). This labelling heuristic together with the anti-lexicographic ordering constraints on the rows and columns works very well for solving tSps. Table 6.1 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 [LR80]. Therefore, only $n = 7$ and $n = 9$ have solutions in the table.

We observe in Table 6.1 that the symmetry breaking ordering 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 satisfiable instances ($n = 7$ and $n = 9$). The difference becomes apparent with the unsatisfiable instances when we have a larger search space to explore. For $n = 8$ and $n = 10$, `LexLessAndSum` reduces the size of the search tree by almost a half compared to `LexLess`.

If we change the labelling heuristic 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 6.2 and 6.3 show the search tree and the run-times obtained when labelling along the columns and rows, and along the rows of the matrix, respectively. Reasoning with lexicographic ordering constraints in the presence of sum constraints now become very useful, and the instances are solved much more quicker than the lexicographic ordering constraints alone, with notable differences in the size of the search tree. Note for instance that $n = 10$ in Table 6.3 could be proved to have no solution only with `LexLessAndSum`, given the time limit 1 hour.

In summary, with a good labelling heuristic for solving tSps, propagating the lexicographic ordering constraints on the columns via `LexLessAndSum` is beneficial over `LexLess` only when we have a large space to explore. Otherwise the benefits are very modest. On the other hand, `LexLessAndSum` significantly reduces the size of the search tree compared to `LexLess` when our labelling heuristic is very poor for solving the problem. Also, the additional constraint propagation helps to solve the problem quicker as the instances get larger.

Balanced Incomplete Block Design Problem This was introduced in Chapter 3.2.1. In Figure 3.1, one way of modelling the problem is given. Since the elements as well as

n	No symmetry breaking			`LexLess` C			`LexLessAndSum` C		
				`LexLeq` R			`LexLeq` R		
	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
6	12,248	12,247	0.5	22	21	0	**11**	**10**	**0**
7	115	129	0.3	21	35	0	**14**	**28**	**0**
8	-	-	-	1,259	1,258	0.2	**410**	**409**	**0.1**
9	4,289,520	4,289,546	697.6	2,106	2,132	0.4	**619**	**645**	**0.2**
10	-	-	-	4,153,162	4,153,161	940.5	**643,152**	**643,151**	**217.2**

Table 6.2: Ternary Steiner problem: `LexLess` vs `LexLessAndSum` with column-and-row-wise labelling.

n	No symmetry breaking			`LexLess` C			`LexLessAndSum` C		
				`LexLeq` R			`LexLeq` R		
	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
6	26,352	26,351	1.2	47	46	0	**27**	**26**	**0**
7	585,469	585,485	40.4	146	162	0	**52**	**68**	**0**
8	-	-	-	6,826	6,825	0.7	**1,962**	**1,961**	**0.4**
9	-	-	-	89,760	89,789	14.1	**8,971**	**9,000**	**2.0**
10	-	-	-	-	-	-	**3,701,480**	**3,701,479**	**1323.7**

Table 6.3: Ternary Steiner problem: `LexLess` vs `LexLessAndSum` with row-wise labelling.

the subsets containing the elements are indistinguishable, the matrix X modelling the problem has row and column symmetry.

Due to the constraints on the rows, no pair of rows can be equal unless $r = \lambda$. To break the row symmetry, we enforce that the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{v-1}$ corresponding to the v elements are strict anti-lexicographically ordered: $\vec{R}_0 >_{lex} \vec{R}_1 \ldots >_{lex} \vec{R}_{v-1}$. As for the column symmetry, we enforce that the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{b-1}$ corresponding to the b subsets of $\mathcal{V}$ are anti-lexicographically ordered: $\vec{C}_0 \geq_{lex} \vec{C}_1 \ldots \geq_{lex} \vec{C}_{b-1}$. Even though this model is similar to that of tSp, there are two main differences: 1) the matrix of BIBD has sum constraints on its rows, but the matrix of tSp does not; 2) the scalar product constraint is posted on the rows of the matrix of BIBD, but on the columns of the matrix of tSp. Moreover, the scalar product is to be at most 1 in tSp but exactly λ in BIBD. Since both the rows and columns are also constrained by sum constraints, we pose the lexicographic ordering constraints using either `LexLeqAndSum` and `LexLessAndSum`, or `LexLeq` and `LexLess`.

Instantiating the matrix along its rows from top to bottom and exploring the domain of each variable in *increasing* order works extremely well with anti-lexicographic ordering constraints on the rows and columns[1]. All the instances of [MT01] 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 labelling heuristic, we observe no difference between our global constraints and the associated lexicographic ordering constraints. Examples can be found in Table 6.4.

[1] This was also pointed out by Jean-François Puget in [Pug02a].

v,b,r,k,λ	No symmetry breaking			LexLessR			LexLessAndSum R		
				LexLeqC			LexLeqAndSum C		
	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
6,20,10,3,4	8,944	8,980	0.7	**43**	**49**	**0**	**43**	**49**	**0**
7,21,9,3,3	7,438	7,483	0.7	**42**	**52**	**0**	**42**	**52**	**0**
6,30,15,3,6	1,893,458	1,893,512	192.3	**68**	**74**	**0.1**	**68**	**74**	**0.1**
7,28,12,3,4	229,241	229,301	26.1	**64**	**75**	**0.1**	**64**	**75**	**0.1**
9,24,8,3,2	6,841	6,922	1.1	**48**	**68**	**0.1**	**48**	**68**	**0.1**
6,40,20,3,8	-	-	-	**108**	**114**	**0.1**	**108**	**114**	**0.1**
7,35,15,3,5	7,814,878	7,814,953	1444.4	**88**	**100**	**0.1**	**88**	**100**	**0.1**
7,42,18,3,6	-	-	-	**115**	**127**	**0.2**	**115**	**127**	**0.2**

Table 6.4: BIBD: LexLess/LexLeq vs LexLessAndSum/LexLeqAndSum with row-wise labelling from top to bottom of the matrix.

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 some 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-end 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 anti-lexicographic ordering constraints. Even small instances become hard to solve within an hour. We can make the problem a bit more difficult to solve by choosing one row from the top and then one row from the bottom, and so on. In this way, we conflict the symmetry breaking constraints and the labelling heuristic. Comparing Table 6.4 and Table 6.5 shows how the search tree is affected with this conflict, though the size of the search tree remains the same if no ordering constraints are imposed.

Table 6.5 contains some interesting results. First, imposing anti-lexicographic 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 labelling heuristic clash, resulting in larger 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 labelling heuristic, 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 by imposing our new global constraint. See entry 8.

In summary, if the labelling heuristic we use for solving BIBDs does not conflict with the lexicographic ordering constraints, LexLeqAndSum and LexLessAndSum reduce neither the size of the search tree nor the time to solve the problem compared to LexLeq and LexLess. If, however, the labelling heuristic pushes search in a different direction to the ordering constraints, then the additional inference of our global constraints can compensate for this conflict. By this way, we can still reduce the size of the search tree

	v, b, r, k, λ	No symmetry			LexLess R			LexLessAndSum R		
		breaking			LexLeq C			LexLeqAndSum C		
		Fails	Choice	Time	Fails	Choice	Time	Fails	Choice	Time
			points	(secs.)		points	(secs.)		points	(secs.)
1	6,20,10,3,4	8,944	8,980	0.7	916	937	0.3	**327**	**345**	**0.1**
2	7,21,9,3,3	7,438	7,483	0.7	20,182	20,203	5.4	**5,289**	**5,309**	**2.1**
3	6,30,15,3,6	1,893,458	1,893,512	192.3	10,618	10,649	3.7	**1,493**	**1,520**	**1.0**
4	7,28,12,3,4	229,241	229,301	26.1	801,290	801,318	330.7	**52,927**	**52,954**	**27.0**
5	9,24,8,3,2	**6,841**	**6,922**	**1.1**	2,338,067	2,338,107	1115.9	617,707	617,741	524.3
6	6,40,20,3,8	-	-	-	117,126	117,167	67.5	**4,734**	**4,770**	**4.4**
7	7,35,15,3,5	7,814,878	7,814,953	1444.4	-	-	-	**382,173**	**382,207**	**311.2**
8	7,42,18,3,6	-	-	-	-	-	-	**2,176,006**	**2,176,047**	**2,603.7**

Table 6.5: BIBD: LexLess/LexLeq vs LexLessAndSum/LexLeqAndSum with row-wise labelling alternating between top and bottom of the matrix.

and time to solve the problem compared to no symmetry breaking with significant gains over LexLeq and LexLess.

6.9 Lexicographic Ordering with Other Constraints

In Section 6.2, we have shown that lexicographic ordering and sum constraints on 0/1 variables frequently occur together in problems involving demand, partitioning or capacity that are modelled using matrices with row and/or column symmetry. In Theorem 33, we proved that global reasoning on the conjunction of a lexicographic ordering constraint and two sum constraints help detect more inconsistent values than reasoning on each of them individually. This motivated us to design the filtering algorithms LexLeqAndSum and LexLessAndSum for propagating the (strict) lexicographic ordering with sum constraints.

In Section 6.8, we have tested how beneficial our new global constraints are in practise on tSp and BIBD. In the experiments we compared LexLeqAndSum and LexLessAndSum with LexLeq and LexLess respectively to observe how much more constraint propagation we achieve. Our results show that the global constraints are useful (1) when there is a very large space to explore, such as when our problem is not satisfiable; (2) when our labelling heuristic is very poor for solving the problem; (3) when our labelling heuristic conflicts with the symmetry breaking constraints. Similar but unreported results are obtained when solving rack configuration problem, steel mill slab design problem, and social golfers problem. Even though any of the situations above is likely to encounter when modelling and solving a CSP, it is disappointing not to have any noteworthy advantage with our new global constraints when the problem is satisfiable or when the labelling heuristic is neither poor nor clashes with the symmetry breaking constraints. This raises the question of whether combining lexicographic ordering with any other constraint is worthwhile?

Katsirelos and Bacchus have proposed a simple heuristic for combining constraints together [KB01]. The heuristic suggests grouping constraints together if they share many variables in common. Following this, LexLeqAndSum$(\vec{X}, \vec{Y}, Sx, Sy)$ is a useful combination of constraints since the variables of $\vec{X} \leq_{lex} \vec{Y}$ are a super set of the variables of $\sum_i X_i = Sx$ and $\sum_i Y_i = Sy$. However, this heuristic takes into account only the number of shared variables. How each of the constraints are propagated is ignored. As discussed in Chapter 5.2, LexLeq prunes only at position α. If the vectors are already ordered at this position then the rest of the future assignments are all irrelevant. Otherwise, α can move to the right but on the average it moves one position for each assignment. At

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	-	1,472	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**

Table 6.6: 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.

each node of the search tree, the lexicographic ordering constraint interacts with each of the sum constraints at one variable on the average. This interaction is of little value because the constraints are already communicating with each other via the domain of that variable. Consequently, we have limited benefits of `LexLeqAndSum`($\vec{X}, \vec{Y}, Sx, Sy$) over `LexLeq`. This argument holds also for combining lexicographic ordering constraints with other constraints. Hence, it may not be worthwhile at all to reason more globally than the lexicographic ordering constraint itself.

Following [FHK$^+$02], Carlsson and Beldiceanu have introduced a new global constraint, called **lex_chain**, which combines together a chain of lexicographic ordering constraints [CB02a]. When we have a matrix say with row symmetry, we can now post a single lexicographic ordering constraint on all the n vectors corresponding to the rows as opposed to posting $n-1$ of them. As discussed in Chapter 5.6, more constraint propagation is expected by this new global constraint. Our experiments on BIBDs , however, indicate no gain over posting lexicographic ordering constraints between the adjacent vectors. In Table 6.6, we report the results of solving BIBDs using SICStus Prolog constraint solver 3.10.1 [SIC03]. 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 each adjacent symmetric rows/columns. In all the cases, we observe 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 only one constraint on the rows/columns. In fact, 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.

Consequently, we expect very limited gain by combining lexicographic ordering with other constraints. Our observations also indicate that combining constraints is useful when the combination is likely to prune a significant number of shared variables.

6.10 Implementation

We implemented our global constraints in C++ using ILOG Solver 5.3 [ILO02]. Implementing a global constraint involves deciding at which propagation events to wake up the constraint and when to propagate the constraint. In this section, we explain the decisions we have made when implementing $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$.

As discussed in Chapter 5.10, three propagation events are available in Solver for an integer variable: **whenValue**, **whenRange**, and **whenDomain**. For 0/1 variables, these three events overlap. That is, if we remove a value from a variable's domain $\{0, 1\}$ then not only the domain of the variable is modified, but also the minimum or maximum in the domain is changed, moreover the variable is assigned a value. Therefore, we can attach any of the propagation events above to the variables of the vectors.

When $\texttt{LexLeqAndSum}(\vec{X}, \vec{Y}, Sx, Sy)$ is GAC, the values in $\vec{X}$ and $\vec{Y}$ are supported by $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ and $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$, respectively. Any modification to the variables affecting $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ or $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$ must trigger the filtering algorithm. It is, however, difficult to know in advance which variable's of $\vec{X}$ (resp. $\vec{Y}$) alter $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ (resp. $max\{\vec{y} \mid \sum_i y_i = Sy \wedge \vec{y} \in \vec{Y}\}$), because this all depends on Sx (resp. Sy) and on the current partial assignments of the variables of $\vec{X}$ (resp. $\vec{Y}$). As an example, consider $\vec{X} = \langle\{0, 1\}, \{0, 1\}, \{0, 1\}, \{0, 1\}, \{0, 1\}, \{0, 1\}, \{0, 1\}\rangle$ with $Sx = 4$. Removing 0 from X_3, X_4, X_5 or X_6 does not change that $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\} = \langle 0, 0, 0, 1, 1, 1, 1\rangle$. Now assume that we reach a state during search where we have $\vec{X} = \langle\{1\}, \{1\}, \{1\}, \{0, 1\}, \{0, 1\}, \{0, 1\}, \{0, 1\}\rangle$. The current $min\{\vec{x} \mid \sum_i x_i = Sx \wedge \vec{x} \in \vec{X}\}$ is $\langle 1, 1, 1, 0, 0, 0, 1\rangle$. Removing 0 from X_6 does not change this, but removing 0 from any of X_3, X_4, and X_5 now assigns a vector to $\vec{X}$ which is lexicographically greater than $\langle 1, 1, 1, 0, 0, 0, 1\rangle$. Therefore, we attach a propagation event to all the variables of the vectors.

When do we propagate our constraint? As discussed in Chapter 5.10, two ways of propagating a constraint are available in Solver: (1) respond to each propagation event individually; (2) wait until all propagation events accumulate. The first method of propagation suits perfectly to an algorithm in which the data structures can be maintained incrementally, and this can be done easily and efficiently at every propagation event. `LexLeqAndSum` constructs 7 vectors (in lines 3, 4, 6.1, 6.4.1, 7.1, 7.2, and 7.5.1) during one execution. Every time `LexLeqAndSum` is called, all the data structures are recomputed from scratch as opposed to restored using their previous value. This means that responding to each event would be very costly. Due to the non-incremental nature of the algorithm, we use the second method of propagation in which the data structures are recomputed after all the events accumulate.

We start by implementing **post** which is called by Solver the first time the constraint is posted. The role of this procedure is to initialise the data structures, define on which events to propagate the constraint, and then to propagate the constraint. The propagation algorithm is implemented using **propagate**. Since we want to accumulate all the events and respond only once, we post a demon on the variables of the vectors so that the propagation does not take place right after a variable is modified. Note that we do not need to distinguish between the demons attached to the variables of $\vec{X}$ and $\vec{Y}$.

Since there are no data structures to be maintained incrementally, we only need to attach an event demon to a **whenRange** event for every variable in the vectors. If we are concerned with entailment then we initialise *entailed* to $false$.

```
Procedure post
  $entailed := false$;
  foreach $i \in [0, n)$ do
      $X_i$.whenRange(EventDemon);
      $Y_i$.whenRange(EventDemon);
  end
```

We now use Solver's **push** to delay our response to each propagation event.

```
Procedure EventDemon
  push();
```

Solver automatically aggregates all such calls to **push** into a single invocation of the **propagate** procedure which propagates the constraint. Note that **propagate** is also automatically called once the constraint is posted.

Procedure **propagate** calls the filtering algorithm.

```
Procedure propagate
  LexLeqAndSum;
```

6.11 Summary

In this chapter, we have introduced two new global constraints `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ and `LexLessAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ for 0/1 variables. Each of the constraints is an ordering constraint and combines together a lexicographic ordering constraint with two sum constraints.

Lexicographic ordering and sum constraints on 0/1 variables frequently occur together in problems involving demand, capacity or partitioning that are modelled using matrices with row and/or column symmetry. Given two vectors of variables $\vec{X}$ and $\vec{Y}$, posting `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is semantically equivalent to posting $\vec{X} \leq_{lex} \vec{Y}$, $\sum_i X_i = Sx$, and $\sum_i Y_i = Sy$. Operationally, a filtering algorithm which removes from the vectors those values that cannot be a part of any solution to `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ can lead to more pruning than the total pruning obtained by the filtering algorithms of the lexicographic ordering constraint and the sum constraint. We have therefore developed an efficient filtering algorithm which either proves that `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$ is disentailed, or ensures GAC on `LexLeqAndSum`$(\vec{X}, \vec{Y}, Sx, Sy)$. The algorithm runs in time $O(n)$. The complexity of the algorithm is optimal as there are $O(n)$ variables to consider.

The filtering algorithm assumes that the sums are ground and exploits the restriction of 0/1 variables. By this way, the algorithm decides at which locations of the vectors the values are inconsistent without having to explore each variable and its domain individually. This gives us a linear time complexity as opposed to quadratic when each variable is naively examined in turn. An alternative way of propagating our new global constraint is to use the algorithm of Bessière and Régin for enforcing GAC on an arbitrary conjunction of constraints. By taking advantage of the semantics of the constraint, we here enforce

GAC more efficiently using our filtering algorithm. The algorithm can easily be modified to obtain a filtering algorithm for LexLessAndSum($\vec{X}, \vec{Y}, Sx, Sy$). Moreover, the non-incremental nature of the algorithm helps detect entailment in a simple and dual manner to detecting disentailment. Furthermore, we can deal with sums that are not ground but bounded by making use of our algorithm.

We have shown that decomposing a chain of these new ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. We have also compared with related work.

Our experiments on tSps and BIBDs show that combining a lexicographic ordering constraint with two sum constraints is useful in a number of ways. First, it is of benefit when there is a very large space to explore, such as when our problem is not satisfiable. Second, it is useful when we lack a good labelling heuristic. Third, it can compensate for the labelling heuristic trying to push the search in a different direction to the symmetry breaking constraints. This is a very important finding considering one of the major disadvantages of breaking symmetry by ordering constraints: a solution satisfying the ordering constraints might be found later in the search process than the solutions not satisfying the ordering constraints. Unfortunately, this combination of the constraints is not useful when the problem is satisfiable or when the labelling heuristic is neither poor nor clashes with the symmetry breaking constraints.

Since the lexicographic ordering constraint is concerned only with the variables at position α, we observe that there is often only a limited interaction between a lexicographic ordering constraint and sum constraints, or between a chain of lexicographic ordering constraints, which can result in limited propagation. We therefore expect a similar behaviour when we combine lexicographic ordering with other constraints. Our observations suggest that combining constraints is useful when the combination is likely to prune a significant number of shared variables.

Chapter 7

Multiset Ordering Constraint

7.1 Introduction

Given two vectors $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a multiset ordering constraint as $\vec{X} \leq_m \vec{Y}$ and a strict multiset ordering constraint as $\vec{X} <_m \vec{Y}$. The multiset ordering constraint $\vec{X} \leq_m \vec{Y}$ ensures that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ respectively, when viewed as multisets, are multiset ordered according to Definition 20. That is, either $\{\{\vec{x}\}\} = \{\{\vec{y}\}\}$, or $max\{\{\vec{x}\}\} < max\{\{\vec{y}\}\}$, or $max\{\{\vec{x}\}\} = max\{\{\vec{y}\}\}$ and $\{\{\vec{x}\}\} - \{\{max\{\{\vec{x}\}\}\}\} <_m \{\{\vec{y}\}\} - \{\{max\{\{\vec{y}\}\}\}\}$. The strict multiset ordering constraint disallows $\{\{\vec{x}\}\} = \{\{\vec{y}\}\}$.

Multiset ordering constraints are useful for breaking row and column symmetries. We can either insist that the rows and columns are both multiset ordered, or enforce multiset ordering in one dimension and lexicographic ordering in the other. How can we post and propagate these constraints effectively and efficiently? In this chapter, we design global constraints for multiset orderings, each of which encapsulates its own filtering algorithm.

This chapter is organised as follows. In Section 7.2, we present a filtering algorithm for the multiset ordering constraint $\vec{X} \leq_m \vec{Y}$. Then in Section 7.3, we discuss the complexity of the algorithm, and prove that the algorithm is correct and complete. An alternative filtering algorithm is introduced in Section 7.4. In Section 7.5, we extend our algorithm to obtain a filtering algorithm for $\vec{X} <_m \vec{Y}$ and to detect entailment, and then we study multiset equality and disequality constraints. Alternative approaches to propagating the multiset ordering constraint are discussed in Section 7.6. We demonstrate in Section 7.7 that decomposing a chain of multiset ordering constraints into multiset ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. Computational results are presented in Section 7.8. Finally, before summarising in Section 7.10, we give in Section 7.9 the details of the implementation.

7.2 A Filtering Algorithm for Multiset Ordering Constraint

In this section, we present a filtering algorithm for the multiset ordering constraint which either detects that $\vec{X} \leq_m \vec{Y}$ is disentailed or prunes inconsistent values so as to achieve GAC on $\vec{X} \leq_m \vec{Y}$. After sketching the main features of the algorithm on a running example in Section 7.2.1, we first present the theoretical results that the algorithm exploits in Section 7.2.2 and then give the details of the algorithm in Section 7.2.3.

7.2.1 A Worked Example

The key idea behind the algorithm is to build and work on a pair of occurrence vectors (see Chapter 2.6) associated with $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$. The algorithm goes through every variable of $\vec{X}$ and $\vec{Y}$ checking for support for values in the domains. It suffices to have $occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})) \leq_{lex} occ(\texttt{ceiling}(\vec{Y}))$ to ensure that all values of $\mathcal{D}(X_i)$ are consistent. Similarly, we only need $occ(\texttt{floor}(\vec{X})) \leq_{lex} occ(\texttt{ceiling}(\vec{Y}_{Y_j \leftarrow min(Y_j)}))$ to hold for the values of $\mathcal{D}(Y_j)$ to be consistent. We can avoid the repeated construction and traversal of these vectors by building, once and for all, the vectors $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$, and defining some pointers and flags on them. For instance, assume we have $occ(\texttt{floor}(\vec{X})) \leq_{lex} occ(\texttt{ceiling}(\vec{Y}))$. The vector $occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)}))$ can be obtained from $occ(\texttt{floor}(\vec{X}))$ by decreasing the number of occurrences of $min(X_i)$ by 1, and increasing the number of occurrences of $max(X_i)$ by 1. The pointers and flags tell us whether this disturbs the lexicographic ordering, and if so they help us to find quickly the largest $max(X_i)$ which does not.

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

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

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

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

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

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

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

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

We have now the following generalised arc-consistent vectors:

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

7.2.2 Theoretical Background

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

Theorem 55 *GAC($\vec{X} \leq_m \vec{Y}$) iff for all $0 \leq i < n$, $max(X_i)$ and $min(Y_i)$ are consistent.*

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

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

Theorem 56 *$\vec{X} \leq_m \vec{Y}$ is disentailed iff $\{\!\{\texttt{floor}(\vec{X})\}\!\} >_m \{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$.*

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

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

Theorem 57 *GAC($\vec{X} \leq_m \vec{Y}$) iff for all i in $[0, n)$:*

$$\{\!\{\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})\}\!\} \leq_m \{\!\{\texttt{ceiling}(\vec{Y})\}\!\} \tag{7.1}$$

$$\{\!\{\texttt{floor}(\vec{X})\}\!\} \leq_m \{\!\{\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})\}\!\} \tag{7.2}$$

Proof: ($\Rightarrow$) As the constraint is GAC, all values have support. In particular, $X_i \leftarrow max(X_i)$ has a support $\vec{x_1} \in \{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ and $\vec{y_1} \in \vec{Y}$ where $\{\!\{\vec{x_1}\}\!\} \leq_m \{\!\{\vec{y_1}\}\!\}$. Any $\vec{x_2} \in \{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ less than or equal to $\vec{x_1}$, and any $\vec{y_2} \in \vec{Y}$ greater than or equal to $\vec{y_1}$, under multiset ordering, support $X_i \leftarrow max(X_i)$. In particular, $min\{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \vec{y} \in \vec{Y}\}$ support $X_i \leftarrow max(X_i)$. We get $min\{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ if all the other variables in $\vec{X}$ take their minimums, and we get $max\{\vec{y} \mid \vec{y} \in \vec{Y}\}$ if all the variables in $\vec{Y}$ take their maximums. Hence, $\{\!\{\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})\}\!\} \leq_m \{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$.

A dual argument holds for the variables of $\vec{Y}$. As the constraint is GAC, $Y_i \leftarrow min(Y_i)$ has a support $\vec{x_1} \in \vec{X}$ and $\vec{y_1} \in \{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ where $\{\!\{\vec{x_1}\}\!\} \leq_m \{\!\{\vec{y_1}\}\!\}$. Any $\vec{x_2} \in \vec{X}$ less than or equal to $\vec{x_1}$, and any $\vec{y_2} \in \{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ greater than or equal to $\vec{y_1}$, in particular $min\{\vec{x} \mid \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ support $Y_i \leftarrow min(Y_i)$. We get $min\{\vec{x} \mid \vec{x} \in \vec{X}\}$ if all the variables in $\vec{X}$ take their minimums, and we get $max\{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ if all the other variables in $\vec{Y}$ take their maximums. Hence, $\{\!\{\texttt{floor}(\vec{X})\}\!\} \leq_m \{\!\{\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})\}\!\}$.

($\Leftarrow$) 7.1 ensures that for all $0 \leq i < n$, $max(X_i)$ is supported, and 7.2 ensures that for all $0 \leq i < n$, $min(Y_i)$ is supported. By theorem 55, the constraint is GAC. QED.

In Theorems 56 and 57, we need to check whether two ground vectors are multiset ordered. The following theorem shows that we can do this by lexicographically comparing the occurrence vectors associated with these vectors.

Theorem 58 *$\{\!\{\vec{x}\}\!\} \leq_m \{\!\{\vec{y}\}\!\}$ iff $occ(\vec{x}) \leq_{lex} occ(\vec{y})$.*

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

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

Theorems 56 and 57 together with Theorem 58 yield to the following two propositions:

Proposition 2 *$\vec{X} \leq_m \vec{Y}$ is disentailed iff $occ(\texttt{floor}(\vec{X})) >_{lex} occ(\texttt{ceiling}(\vec{Y}))$.*

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

$$occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})) \leq_{lex} occ(\texttt{ceiling}(\vec{Y})) \tag{7.3}$$

$$occ(\texttt{floor}(\vec{X})) \leq_{lex} occ(\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})) \tag{7.4}$$

A naive way of enforcing GAC on $\vec{X} \leq_m \vec{Y}$ is going through every variable in the vectors, constructing the appropriate occurrence vectors, and checking if their bounds satisfy 7.3 or 7.4. If yes then the bound is consistent. Otherwise, we try the nearest bound until we obtain a consistent bound. We can, however, do better than this by building only the vectors $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$, and then defining some pointers and Boolean flags on them. This saves us from the repeated construction and traversal of the appropriate occurrence vectors. Another advantage is that we can find consistent bounds without having to explore the values in the domains.

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

Definition 35 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, the pointer α is set either to the index in $[u, l]$ such that:*

$$ox_\alpha < oy_\alpha \ \wedge$$

$$\forall i \ u \geq i > \alpha \, . \, ox_i = oy_i$$

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

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

Definition 36 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, the pointer β is set either to the index in $(\alpha, l]$ such that:*

$$ox_\beta > oy_\beta \ \wedge$$

$$\forall i \ \alpha > i > \beta \, . \, ox_i \leq oy_i$$

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

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

Definition 37 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, the flag γ is true iff:*

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

Informally, γ is *true* if $\beta \neq -\infty$, and either $\beta = \alpha - 1$ or $\vec{ox}_{\alpha-1\to\beta+1} = \vec{oy}_{\alpha-1\to\beta+1}$. If, however, $\beta = -\infty$, or $\beta < \alpha - 1$ and $\vec{ox}_{\alpha-1\to\beta+1} <_{lex} \vec{oy}_{\alpha-1\to\beta+1}$, then γ is *false*.

Definition 38 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leq_{lex} \vec{oy}$, the flag σ is true iff:*

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

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

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

Theorem 59 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leq_{lex} \vec{oy}$, if $max(X_i) \geq \alpha$ and $min(X_i) < \alpha$ then for all $v \in \mathcal{D}(X_i)$:*

1. *if $v > \alpha$ then v is inconsistent;*

2. *if $v < \alpha$ then v is consistent;*

3. *if $v = \alpha$ then v is inconsistent iff:*

$$\begin{aligned}
(ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1) \quad &\vee \\
(ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma) \quad &\vee \\
(ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) < \beta \ \wedge \ \gamma) \quad &
\end{aligned}$$

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

or $ox_{\alpha+1}+1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma$. Now consider the second case, where we have $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ min(X_i) < \beta \ \wedge \ \gamma$. Decreasing $ox_{min(X_i)}$ by 1 gives $\vec{ox} >_{lex} \vec{oy}$. Hence, if α is inconsistent then we have either of the three conditions. QED.

Theorem 60 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, if $max(X_i) < \alpha$ then $max(X_i)$ is the tight upper bound.*

Proof: If $max(X_i) < \alpha$ then we have $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Increasing $ox_{max(X_i)}$ by 1 does not change this. By Proposition 3, $max(X_i)$ is consistent. QED.

Theorem 61 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, if $min(X_i) \geq \alpha$ then $min(X_i)$ is the tight upper bound.*

Proof: Any $v > min(X_i)$ in $\mathcal{D}(X_i)$ is greater than α. Increasing ox_v by 1 gives $\vec{ox} >_{lex} \vec{oy}$. By Proposition 3, any $v > min(X_i)$ in $\mathcal{D}(X_i)$ is inconsistent. QED.

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

Theorem 62 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, if $max(Y_i) = \alpha$ and $min(Y_i) \leq \beta$ then for all $v \in \mathcal{D}(Y_i)$*

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

$$\begin{array}{c}(ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1) \quad \vee \\ (ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma)\end{array}$$

Proof: If $max(Y_i) = \alpha$ and $min(Y_i) \leq \beta$ then $\alpha \neq -\infty$, $\beta \neq -\infty$, and $\vec{ox} <_{lex} \vec{oy}$. Let v be a value in $\mathcal{D}(Y_i)$ greater than β. Increasing oy_v by 1 and decreasing oy_α by 1 does not change $\vec{ox} <_{lex} \vec{oy}$. This is because for all $\alpha > i > \beta$, we have $ox_i \leq oy_i$. Even if now $\vec{ox}_{\alpha \to v+1} = \vec{oy}_{\alpha \to v+1}$, at v we have $ox_v < oy_v$. By Proposition 3, v is consistent. Now let v be less than β. If the condition in item 2 is *true* then we obtain $\vec{ox} >_{lex} \vec{oy}$ by decreasing oy_α by 1 and increasing oy_v by 1. By Proposition 3, v is inconsistent. We now need to show that this condition is exhaustive. If v is inconsistent then by Proposition 3, we obtain $\vec{ox} >_{lex} \vec{oy}$ after decreasing oy_α by 1 and increasing oy_v by 1. This is in fact the same as obtaining $\vec{ox} >_{lex} \vec{oy}$ after increasing ox_α by 1 and decreasing ox_v by 1. We have already captured this case in the last condition of item 3 in Theorem 59. Hence, it is necessary to have $ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma$ for v to be inconsistent. What about β then? If any of the conditions in item 3 is *true* then we obtain $\vec{ox} >_{lex} \vec{oy}$ by decreasing oy_α by 1 and increasing oy_β by 1. By Proposition 3, β is inconsistent. In this case, the values less than β are also inconsistent. Therefore, the smallest value which is greater than β is the tight lower bound. We now need to show that the conditions of item 3 are exhaustive. If β is inconsistent then by Proposition 3, we obtain $\vec{ox} >_{lex} \vec{oy}$ after decreasing oy_α by 1 and increasing oy_β by 1. This is the same as obtaining $\vec{ox} >_{lex} \vec{oy}$

```
Algorithm 36: Initialise
    Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
    Result : $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$ are initialised, GAC($\vec{X} \leq_m \vec{Y}$)
    $l := min(\{\!\{\texttt{floor}(\vec{X})\}\!\} \cup \{\!\{\texttt{floor}(\vec{Y})\}\!\})$;
    $u := max(\{\!\{\texttt{ceiling}(\vec{X})\}\!\} \cup \{\!\{\texttt{ceiling}(\vec{Y})\}\!\})$;
    $\vec{ox} := occ(\texttt{floor}(\vec{X}))$;
    $\vec{oy} := occ(\texttt{ceiling}(\vec{Y}))$;
    MsetLeq;
```

after increasing ox_α by 1 and decreasing ox_β by 1. We have captured this case in the first two conditions of item 3 in Theorem 59. Hence, if β is inconsistent then we have either $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1$ or $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma$. QED.

Theorem 63 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, if $max(Y_i) = \alpha$ and $min(Y_i) > \beta$ then $min(Y_i)$ is the tight lower bound.*

Proof: If $max(Y_i) = \alpha$ then $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Increasing $oy_{min(Y_i)}$ by 1 and decreasing oy_α by 1 does not change $\vec{ox} <_{lex} \vec{oy}$. This is because for all $\alpha > i > max\{l-1, \beta\}$, we have $ox_i \leq oy_i$. Even if now $\vec{ox}_{\alpha \to min(Y_i)+1} = \vec{oy}_{\alpha \to min(Y_i)+1}$, at $min(Y_i)$ we have $ox_{min(Y_i)} < oy_{min(Y_i)}$. By Proposition 3, $min(Y_i)$ is consistent. QED.

Theorem 64 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, if $max(Y_i) < \alpha$ then $min(Y_i)$ is the tight lower bound.*

Proof: If $max(Y_i) < \alpha$ then we have $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Decreasing $oy_{max(Y_i)}$ by 1 does not change this. By Proposition 3, $min(Y_i)$ is consistent. QED.

Theorem 65 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as u..l where $\vec{ox} \leq_{lex} \vec{oy}$, if $max(Y_i) > \alpha$ then $max(Y_i)$ is the tight lower bound.*

Proof: Decreasing $oy_{max(Y_i)}$ by 1 gives $\vec{ox} >_{lex} \vec{oy}$. By Proposition 3, any $v < max(Y_i)$ in $\mathcal{D}(Y_i)$ is inconsistent. QED.

7.2.3 Algorithm Details

Based on Theorems 59-65, we have designed an efficient linear time algorithm, `MsetLeq`, which either detects the disentailment of $\vec{X} \leq_m \vec{Y}$ or prunes inconsistent values so as to achieve GAC on $\vec{X} \leq_m \vec{Y}$.

The algorithm uses two pointers α and β, and two flags γ and σ, all defined on the occurrence vectors $\vec{ox}$ and $\vec{oy}$ to avoid traversing these vectors each time we look for support. The pointers and flags are recomputed every time the algorithm is called, as maintaining them incrementally in an easy way is not obvious. Fortunately, incremental maintenance of the occurrence vectors is trivial. When the minimum value in some $\mathcal{D}(X_i)$ changes, we update $\vec{ox}$ by incrementing the entry corresponding to new $min(X_i)$ by 1, and decrementing the entry corresponding to old $min(X_i)$ by 1. Similarly, when the maximum value in some $\mathcal{D}(Y_i)$ changes, we update $\vec{oy}$ by incrementing the entry corresponding to new $max(Y_i)$ by 1, and decrementing the entry corresponding to old $max(Y_i)$ by 1.

When the constraint is first posted, we need to initialise the occurrence vectors, and call the filtering algorithm `MsetLeq` to establish the generalised arc-consistent state with the initial values of the occurrence vectors. In Algorithm 36, we show the steps of this initialisation. In lines 1 and 2, we compute indices for our occurrence vectors. The indexing is from left to right with the most significant index being u and the least significant index being l such that $u > l$. The values in $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$ are not enough to index our occurrence vectors because we will later use them to check support for $max(X_i)$ and $min(Y_i)$ for all $0 \leq i < n$. Hence, u is the maximum among the maximums, and l is the minimum among the minimums of the variables in $\vec{X}$ and $\vec{Y}$. In lines 3 and 4, we construct the occurrence vectors as follows. We create a pair of vectors $\vec{ox}$ and $\vec{oy}$ of length $u - l + 1$, where each ox_i and oy_i are initially set to 0. Then, for each value v in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$, we increment ox_v by 1. Similarly, for each value v in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$, we increment oy_v by 1. Finally, we call the filtering algorithm `MsetLeq` in line 5.

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

In Algorithm 37, we show the steps of `MsetLeq`. Since $\vec{ox}$ and $\vec{oy}$ are maintained incrementally, the algorithm first sets the pointers and flags in line **A1** using the current state of these vectors.

In line 2 of `SetPointersAndFlags`, we traverse $\vec{ox}$ and $\vec{oy}$, starting at index u, until either we reach the end of the vectors (because the vectors are equal), or we find an index i where $ox_i \neq oy_i$. In the first case, we set α to $-\infty$ (line 4). In the second case, we set α to i only if $ox_i < oy_i$ (line 5). This is the most significant index where the vectors are strictly ordered. If, however, $ox_i > oy_i$, then disentailment is detected and `SetPointersAndFlags` terminates with failure (line 3). This also triggers the filtering algorithm to fail.

In lines 6-13, we seek a value for β. If $\alpha \leq l$ then we set β to $-\infty$ in line 6. Otherwise, we traverse the vectors in lines 9-11, starting at index $\alpha-1$, until either we reach the end of the vectors (because $\vec{ox}_{\alpha-1 \to l} \leq_{lex} \vec{oy}_{\alpha-1 \to l}$), or we find an index j where $ox_j > oy_j$. We set β to $-\infty$ in the first case (line 12), but instead to j in the second case (line 13). During this traversal, we record in an intermediate Boolean flag *temp* whether the subvectors between α and β are equal. More precisely, *temp* is *true* iff $\vec{ox}_{\alpha-1 \to max\{l, \beta+1\}} = \vec{oy}_{\alpha-1 \to max\{l, \beta+1\}}$. Using α, β, and *temp*, we can now decide the value of γ. In line 14, we initialise γ to $false$. We set γ to *true* in line 15 only if $\beta \neq -\infty$, and either $\beta = \alpha - 1$ (there are no subvectors between α and β) or *temp* is *true* (the subvectors between α and β are equal).

Finally, after initialising σ to $false$ in line 14, we check in line 16 whether there are any subvectors below β. If there are then in line 18 we traverse $\vec{ox}$ and $\vec{oy}$, starting at index $\beta - 1$, until either we reach the end of the vectors (because the subvectors below β are equal), or we find an index k where $ox_k \neq oy_k$. In the first case, σ remains $false$. In the second case, we set σ to *true* only if $ox_k > oy_k$ (line 19). This is the most significant index where the subvectors below β are ordered lexicographically the wrong way. Otherwise ($ox_k < oy_k$), σ remains $false$.

In lines **B1-11** and **C1-9**, we check support for $max(X_i)$ and $min(Y_i)$ for all i in $[0, n)$ as follows. We obtain $occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)}))$ by increasing $ox_{max(X_i)}$ by 1, and decreasing $ox_{min(X_i)}$ by 1. If now we have $\vec{ox} >_{lex} \vec{oy}$ then we find the tight upper bound for X_i. Likewise, we obtain $occ(\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)}))$ by increasing $oy_{min(Y_i)}$ by 1, and decreasing $oy_{max(Y_i)}$ by 1. If now we have $\vec{ox} >_{lex} \vec{oy}$ then we find the tight lower bound for

Algorithm 37: `MsetLeq`

```
    Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
    Result : GAC($\vec{X} \leq_m \vec{Y}$)
A1  SetPointersAndFlags;
B1  foreach $i \in [0, n)$ do
B2      if $min(X_i) \neq max(X_i)$ then
B3          if $min(X_i) \geq \alpha$ then setMax($X_i, min(X_i)$);
B4          if $max(X_i) \geq \alpha \ \wedge \ min(X_i) < \alpha$ then
B5              setMax($X_i, \alpha$);
B6              if $ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma$ then
B7                  if $ox_\beta = oy_\beta + 1$ then
B8                      if $\sigma$ then setMax($X_i, \alpha - 1$);
                    else
B9                      setMax($X_i, \alpha - 1$);
                    end
                end
B10             if $ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) < \beta \ \wedge \ \gamma$ then
B11                 setMax($X_i, \alpha - 1$);
                end
            end
        end
    end
C1  foreach $i \in [0, n)$ do
C2      if $min(Y_i) \neq max(Y_i)$ then
C3          if $max(Y_i) > \alpha$ then setMin($Y_i, max(Y_i)$);
C4          if $max(Y_i) = \alpha \ \wedge \ min(Y_i) \leq \beta$ then
C5              if $ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma$ then
C6                  setMin($Y_i, \beta$);
C7                  if $ox_\beta = oy_\beta + 1$ then
C8                      if $\sigma$ then setMin($Y_i, \beta + 1$);
                    else
C9                      setMin($Y_i, \beta + 1$);
                    end
                end
            end
        end
    end
```

Y_i. For each variable, we only check for support if its domain is not a singleton (lines **B2** and **C2**). The reason is as follows. In `SetPointersAndFlags`, we check whether we have $\vec{ox} >_{lex} \vec{oy}$. If so then we fail; otherwise we have $\vec{ox} \leq_{lex} \vec{oy}$. This means that $min(X_i)$ and $max(Y_i)$ for all $0 \leq i < n$ are consistent. We therefore seek support for a variable only if its domain is not a singleton.

There are two cases where we prune the domain of some X_i: (1) when $min(X_i) \geq \alpha$ (line **B3**); (2) when $max(X_i) \geq \alpha$ and $min(X_i) < \alpha$ (line **B4**). Otherwise, we have $max(X_i) < \alpha$ and increasing $ox_{max(X_i)}$ by 1 does not disturb the lexicographic ordering.

Procedure SetPointersAndFlags

```
$i := u$;
**while** $i \geq l \ \wedge \ ox_i = oy_i$ **do** $i := i - 1$;
**if** $i \geq l \ \wedge \ ox_i > oy_i$ **then** fail;
**else if** $i = l - 1$ **then** $\alpha := -\infty$;
**else** $\alpha := i$;
**if** $\alpha \leq l$ **then** $\beta := -\infty$;
**else if** $\alpha > l$ **then**
    $j := \alpha - 1$, $temp := true$;
    **while** $j \geq l \ \wedge \ ox_j \leq oy_j$ **do**
        **if** $ox_j < oy_j$ **then** $temp := false$;
        $j := j - 1$;
    **end**
    **if** $j = l - 1$ **then** $\beta := -\infty$;
    **else** $\beta := j$;
**end**
$\gamma := false$, $\sigma := false$;
**if** $\beta \neq -\infty \ \wedge \ (\beta = \alpha - 1 \ \vee \ temp)$ **then** $\gamma := true$;
**if** $\beta > l$ **then**
    $k := \beta - 1$;
    **while** $k \geq l \ \wedge \ ox_k = oy_k$ **do** $k := k - 1$;
    **if** $k \geq l \ \wedge \ ox_k > oy_k$ **then** $\sigma := true$;
**end**
```

In the first case, since any $v > min(X_i)$ is greater than α, increasing ox_v by 1 gives $\vec{ox} >_{lex} \vec{oy}$. We therefore prune any value greater than $min(X_i)$ (via the call `SetMax`). In the second case, increasing ox_v by 1 gives $\vec{ox} >_{lex} \vec{oy}$ if $v > \alpha$. Any value greater than α therefore lacks support and is pruned (line **B5**). Now $max(X_i)$ is α, but does it have any support? Increasing ox_α and decreasing $ox_{min(X_i)}$ by 1 can disturb the lexicographic ordering only when $ox_\alpha + 1 = oy_\alpha$ and $min(X_i) \leq \beta$, and γ is $true$. In lines **B6-B9**, we consider the subcase $min(X_i) = \beta$, whereas in lines **B10-B11** we consider the subcase $min(X_i) < \beta$. In the first subcase, we have $\vec{ox}_{u \to \beta-1} = \vec{oy}_{u \to \beta-1}$. If decreasing ox_β by 1 does not change the fact that the vectors are ordered lexicographically the wrong way as of β, then we prune α (line **B9**). If, however, decreasing ox_β by 1 gives us $ox_\beta = oy_\beta$ (line **B7**) then we prune α only if the subvectors below β are ordered lexicographically the wrong way (line **B8**). Now, consider the second subcase. We again have $\vec{ox}_{u \to \beta-1} = \vec{oy}_{u \to \beta-1}$. Since the vectors are ordered lexicographically the wrong way as of β, this property persists by decreasing $min(X_i)$ by 1. We therefore prune α (**B11**).

There are two cases where we prune the domain of some Y_i: (1) when $max(Y_i) > \alpha$ (line **C3**); (2) when $max(Y_i) = \alpha$ and $min(Y_i) \leq \beta$ (line **C4**). Otherwise, we have either $max(Y_i) = \alpha$ and $min(Y_i) > \beta$, or $max(Y_i) < \alpha$, and in any case increasing $oy_{min(Y_i)}$ by 1, and decreasing $oy_{max(Y_i)}$ do not disturb the lexicographic ordering.

In the first case, decreasing $oy_{max(Y_i)}$ by 1 gives $\vec{ox} >_{lex} \vec{oy}$. We therefore prune any value less than $max(Y_i)$ (via the call `SetMin`). In the second case, decreasing oy_α and increasing $oy_{min(Y_i)}$ by 1 can disturb the lexicographic ordering only when $ox_\alpha + 1 = oy_\alpha$ and γ is $true$ (**C5**). In this case, we have $\vec{ox}_{u \to \beta-1} = \vec{oy}_{u \to \beta-1}$. Increasing an occurrence of $\vec{oy}$ below β does not change that $\vec{ox} >_{lex} \vec{oy}$. Any value less than β lacks support and is therefore pruned (line **C6**). Now $min(Y_i)$ is β, but does it have any support? If increasing oy_β by 1 does not change the fact that the vectors are ordered lexicographically the wrong way as of β, then we prune β (line **C9**). If, however, increasing oy_β by 1 gives us $ox_\beta = oy_\beta$ (line **C7**) then we prune β only if the subvectors below β are ordered lexicographically the wrong way (line **C8**).

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

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

The algorithm corrects a mistake that appears in [FHK$^+$03]. We have noticed that in [FHK$^+$03] we do not always prune the values greater than α when we have $max(X_i) \geq \alpha$ and $min(X_i) < \alpha$. As shown next, this algorithm is correct and complete.

7.3 Theoretical Properties

`Initialise` runs in time $O(n + d)$ and `MsetLeq` runs in time $O(nb + d)$, where b is the cost of adjusting the bounds of a variable, and d is the length of the occurrence vectors.

Theorem 66 `Initialise` *runs in time $O(n+d)$, where d is the length of the occurrence vectors.*

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

Theorem 67 `MsetLeq` *runs in time $O(nb+d)$, where b is the cost of adjusting the bounds of a variable, and d is the length of the occurrence vectors.*

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

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

Both `Initialise` and `MsetLeq` are correct and complete.

Theorem 68 `Initialise` *initialises $\vec{ox}$ and $\vec{oy}$ correctly. Then it either establishes failure if $\vec{X} \leq_m \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC($\vec{X} \leq_m \vec{Y}$).*

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

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

Theorem 69 `MsetLeq` *either establishes failure if $\vec{X} \leq_m \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC($\vec{X} \leq_m \vec{Y}$).*

Proof: `MsetLeq` calls `SetPointersAndFlags`. We first show that this procedure either sets α, β, γ, and σ as per their definitions, or establishes failure as $\vec{X} \leq_m \vec{Y}$ is disentailed.

Line 2 of `SetPointersAndFlags` traverses $\vec{ox}$ and $\vec{oy}$, starting at index u, until either it reaches the end of the vectors (because $\vec{ox} = \vec{oy}$), or it finds an index i where $ox_i \neq oy_i$. In the first case, α is set to $-\infty$ (line 4) as per Definition 35. In the second case, α is set to i only if $ox_i < oy_i$ (line 5). This is correct by Definition 35 and means that $\vec{ox} <_{lex} \vec{oy}$. If, however, $ox_i > oy_i$ then we have $\vec{ox} >_{lex} \vec{oy}$. By Proposition 2, $\vec{X} \leq_m \vec{Y}$ is disentailed and thus `SetPointersAndFlags` terminates with failure (line 3). This also triggers the filtering algorithm to fail.

If $\alpha \leq l$ then β is set to $-\infty$ (line 6) as per Definition 36. Otherwise, the vectors are traversed in lines 9-11, starting at index $\alpha - 1$, until either the end of the vectors are reached (because $\vec{ox}_{\alpha-1 \to l} \leq_{lex} \vec{oy}_{\alpha-1 \to l}$), or an index j where $ox_j > oy_j$ is found. In the first case, β is set to $-\infty$ (line 12), and in the second case, β is set j (line 13) as per Definition 36. During this traversal, the Boolean flag *temp* is set to *true* iff $\vec{ox}_{\alpha-1 \to max\{l,\beta+1\}} = \vec{oy}_{\alpha-1 \to max\{l,\beta+1\}}$. In lines 14 and 15, γ is set to *true* iff $\beta \neq -\infty$, and either $\beta = \alpha - 1$ or *temp* is *true* (because $\vec{ox}_{\alpha-1 \to \beta+1} = \vec{oy}_{\alpha-1 \to \beta+1}$). This is correct by Definition 37.

In line 14, σ is initialised to *false*. If $\beta \leq l$ then σ remains *false* (line 16) as per Definition 38. Otherwise, the vectors are traversed in line 18, starting at index $\beta - 1$, until either the end of the vectors are reached (because $\vec{ox}_{\beta-1 \to l} = \vec{oy}_{\beta-1 \to l}$), or an index k where $ox_k \neq oy_k$ is found. In the first case, σ remains *false* as per Definition 38. In the second case, σ is set to *true* only if $ox_k > oy_k$ (line 19). This is correct by Definition 38 and means that $\vec{ox}_{\beta-1 \to l} >_{lex} \vec{oy}_{\beta-1 \to l}$. If, however, $ox_k < oy_k$ then σ remains *false* as per Definition 38.

We now analyse the rest of `MsetLeq`, where the tight upper bound for X_i and the tight lower bound for Y_i, for all $0 \leq i < n$, are sought. If $\vec{X} \leq_m \vec{Y}$ is not disentailed then we have $\vec{ox} \leq_{lex} \vec{oy}$ by Proposition 2. This means that $min(X_i)$ and $max(Y_i)$ for all $0 \leq i < n$ are consistent by Proposition 3. The algorithm therefore seeks the tight upper bound for

X_i only if $max(X_i) > min(X_i)$ (lines **B2-11**), and similarly the tight lower bound for Y_i only if $min(Y_i) < max(Y_i)$ (lines **C2-9**).

For each $\mathcal{D}(X_i)$: (1) If $min(X_i) \geq \alpha$ then all values greater than $min(X_i)$ are pruned, giving $min(X_i)$ as the tight upper bound (line **B3**). This is correct by Theorem 61. (2) If $max(X_i) \geq \alpha \ \wedge \ min(X_i) < \alpha$ then:

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

All the values less than α remain in the domain. By Theorem 59, all the inconsistent values are removed. (3) If, however, $max(X_i) < \alpha$ then $max(X_i)$ is the tight upper bound by Theorem 60, and thus no pruning is necessary.

For each $\mathcal{D}(Y_i)$: (1) If $max(Y_i) > \alpha$ then all values less than $max(Y_i)$ are pruned, giving $max(Y_i)$ as the tight lower bound (line **C3**). This is correct by Theorem 65. (2) If $max(Y_i) = \alpha \ \wedge \ min(Y_i) \leq \beta$ then:

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

All the values greater than β remain in the domain. By Theorem 62, all the inconsistent values are removed. (3) If, however, $max(Y_i) = \alpha \ \wedge \ min(Y_i) > \beta$ or $max(Y_i) < \alpha$ then $min(Y_i)$ is the tight lower bound by Theorems 63 and 64, and thus no pruning is needed.

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

7.4 Multiset Ordering with Large Domains

`MsetLeq` is a linear time algorithm in the length of the vectors given that $u - l \ll n$, where u is the maximum among the maximums, and l is the minimum among the minimums of the variables in $\vec{X}$ and $\vec{Y}$. If we instead have $n \ll u - l$ then the complexity of the algorithm is $O(u - l)$, dominated by the cost of the construction of the occurrence vectors and the initialisation of the pointers and flags. This can happen, for instance, when the vectors being multiset ordered are variables in the occurrence representation of a multiset [KW02]. Is there then an alternative way of propagating the multiset ordering constraint whose complexity is independent of the domains?

7.4.1 Remedy

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

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

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

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

This means that the number occurrences of any value greater than sy_i are equal in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$, but there are more occurrence of sy_i in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ than in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. That is, $ox_5 = oy_5$ and $ox_4 < oy_4$. By Definition 35, α points to 4. We now move only along $\vec{sy}$ until we find an index j such that $sy_j \neq sy_{j-1}$, so that we reason about the number of occurrences of the smaller values. In our example, j is 4:

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

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

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

We now have $sx_i > sy_j$, which suggests that sx_i occurs at least once in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ but does not occur in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. That is, $ox_2 > 0$ and $oy_2 = 0$. By Definition 36, β points to 2. This does not change that γ is $true$. We now move only along $\vec{sx}$ by incrementing i until we find $sx_i \neq sx_{i-1}$, so that we reason about the number of occurrences of the smaller values:

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

With the new value of i, we have $sx_i = sy_i = 1$. This increases both ox_1 and oy_1 by one. Reaching the end of only $\vec{sx}$ hints the following: either 1 occurs more than once in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$, or it occurs once but there are values in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ less than 1 and they do not occur in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. By Definition 38, γ is $false$.

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

Algorithm 39: `Initialise`

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

~~$l := min(\{min(X_i) \mid i \in [0,n)\} \cup \{min(Y_i) \mid i \in [0,n)\})$;~~
~~$u := max(\{max(X_i) \mid i \in [0,n)\} \cup \{max(Y_i) \mid i \in [0,n)\})$;~~
~~$\vec{ox} := occ(\texttt{floor}(\vec{X}))$~~   $\vec{sx} := sort(\texttt{floor}(\vec{X}))$;
~~$\vec{oy} := occ(\texttt{ceiling}(\vec{Y}))$~~   $\vec{sy} := sort(\texttt{ceiling}(\vec{Y}))$;
MsetLeq;
```

7.4.2 An Alternative Filtering Algorithm

As witnessed in the previous section, it suffices to sort `floor`$(\vec{X})$ and `ceiling`$(\vec{Y})$, and scan the sorted vectors to compute α, β, γ, σ, ox_α, oy_α, ox_β, and oy_β. We can then directly reuse lines **B1-11** and **C1-9** of `MsetLeq` to obtain a new filtering algorithm. As a result, we need to change only `Initialise` and `SetPointersAndFlags`.

In Algorithm 39, we show how we modify Algorithm 36. Instead of constructing a pair of occurrence vectors associated with `floor`$(\vec{X})$ and `ceiling`$(\vec{Y})$, we now sort `floor`$(\vec{X})$ and `ceiling`$(\vec{Y})$.

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

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

Theorem 70 *$occ(\vec{x}) \leq_{lex} occ(\vec{y})$ iff $sort(\vec{x}) \leq_{lex} sort(\vec{y})$.*

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

($\Leftarrow$) Suppose $\vec{ox} = occ(\vec{x})$, $\vec{oy} = occ(\vec{y})$, $\vec{sx} = sort(\vec{x})$, $\vec{sy} = sort(\vec{y})$, and we have $\vec{sx} = \vec{sy}$. Then $\{\!\{\vec{x}\}\!\}$ and $\{\!\{\vec{y}\}\!\}$ contain the same elements with equal occurrences. Hence, $\vec{ox} = \vec{oy}$. Suppose $\vec{sx} <_{lex} \vec{sy}$. If $sx_0 < sy_0$ then the leftmost index of $\vec{ox}$ and $\vec{oy}$ is sy_0,

```
Procedure SetPointersAndFlags
1   $i := 0$;
2   while $i < n \ \wedge \ sx_i = sy_i$ do $i := i + 1$;
3   if $i < n \ \wedge \ sx_i > sy_i$ then fail;
4   else if $i = n$ then $\alpha := -\infty$, $\beta := -\infty$, $\gamma := false$, $\sigma := false$, return;
5   else $\alpha := sy_i$;
6   $\gamma := true$;
7   $j := i + 1$;
8   while $j < n \ \wedge \ sy_j = sy_{j-1}$ do $j := j + 1$;
9   if $j = n$ then $\beta := sx_i$;
10  else if $j < n$ then
11      while $i < n \ \wedge \ j < n$ do
12          if $sx_i > sy_j$ then $\beta := sx_i$, exit;
13          if $sx_i < sy_j$ then $\gamma := false$, $j := j + 1$;
14          if $sx_i = sy_j$ then $i := i + 1$, $j := j + 1$;
        end
15      if $j = n$ then $\beta := sx_i$;
    end
16  $k := i + 1$;
17  while $k < n \ \wedge \ sx_k = sx_{k-1}$ do $k := k + 1$;
18  if $k = n$ then $\sigma := false$;
19  else if $k < n$ then
20      while $k < n \ \wedge \ j < n$ do
21          if $sx_k > sy_j$ then $\sigma := true$, exit;
22          if $sx_k < sy_j$ then $\sigma := false$, exit;
23          if $sx_k = sy_j$ then $k := k + 1$, $j := j + 1$;
        end
24      if $k = n$ then $\sigma := false$, exit;
25      if $j = n$ then $\sigma := true$;
    end
26  $i := 0$, $ox_\alpha = 0$, $oy_\alpha = 0$, $ox_\beta = 0$, $oy_\beta = 0$;
27  foreach $i \in [0, n)$ do
28      if $sx_i = \alpha$ then $ox_\alpha := ox_\alpha + 1$;
29      if $sx_i = \beta$ then $ox_\beta := ox_\beta + 1$;
30      if $sy_i = \alpha$ then $oy_\alpha := oy_\alpha + 1$;
31      if $sy_i = \beta$ then $oy_\beta := oy_\beta + 1$;
    end
```

and we have $ox_{sy_0} = 0$ and $oy_{sy_0} > 0$. This gives $\vec{ox} <_{lex} \vec{oy}$. If $sx_0 = sy_0 = a$ then we eliminate one occurrence of a from $\{\{\vec{x}\}\}$ and $\{\{\vec{y}\}\}$, and compare the resulting multisets. QED.

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

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

The process of finding the value of σ (lines 16-25) is very similar to that of β. In line 17, we traverse $\vec{sx}$, starting at index $i+1$, until either we reach the end of the vector (because all the remaining values in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ are β), or we find an index k such that $sx_k \neq sx_{k-1}$. In the first case, we set σ to $false$ (line 18) because either sy_j occurs at least once in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ but does not occur in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ (due to line 12), or there are no values less than β both in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ (due to line 15). In the second case, sx_k gives us the next largest value in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. In lines 20-23, we traverse $\vec{sx}$ starting from k, and $\vec{sy}$ starting from j. The reasoning now is very similar to that of the traversal for β. Instead of setting a value for β, we set σ to $true$, and instead of setting γ to $false$, we set σ to $false$, for the same reasons. If k reaches n before j, then we set σ to $false$ (line 24) due to the same reason as in line 22. If k and j reach n together, then again we set σ to $false$, because we have the same number of occurrences of any value less than β in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. If, however, j reaches n before k, then we set σ to $true$ (line 25) due to the same reason as in line 21.

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

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

7.5 Extensions

In this section, we tackle three important questions. First, how can we enforce strict multiset ordering? Second, how can we catch entailment? And third, can we propagate multiset disequality and equality constraints using (strict) multiset ordering constraints without any loss in the amount of constraint propagation? The answers are given in Sections 7.5.1, 7.5.2, and 7.5.3, respectively.

7.5.1 Strict Multiset Ordering Constraint

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

Before showing how we modify `MsetLeq`, we first study $\vec{X} <_m \vec{Y}$ from a theoretical point of view. In the following three theorems, we show that $\vec{X} <_m \vec{Y}$ is disentailed or is GAC under the conditions similar to those of $\vec{X} \leq_m \vec{Y}$ except that equality between the multiset view of the vectors is now not allowed.

Theorem 71 *$\vec{X} <_m \vec{Y}$ is disentailed iff $\{\!\{\texttt{floor}(\vec{X})\}\!\} \geq_m \{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$.*

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

($\Leftarrow$) Any $\vec{x} \in \vec{X}$ is greater than or equal to any $\vec{y} \in \vec{Y}$ under multiset ordering. Hence, $\vec{X} <_m \vec{Y}$ is disentailed. QED.

Theorem 72 *GAC($\vec{X} <_m \vec{Y}$) iff for all $0 \leq i < n$, $max(X_i)$ and $min(Y_i)$ are consistent.*

Proof: Similar to the proof of Theorem 55. QED.

Theorem 73 *GAC($\vec{X} <_m \vec{Y}$) iff for all i in $[0, n)$:*

$$\{\!\{\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})\}\!\} \quad <_m \quad \{\!\{\texttt{ceiling}(\vec{Y})\}\!\} \tag{7.5}$$

$$\{\!\{\texttt{floor}(\vec{X})\}\!\} \quad <_m \quad \{\!\{\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})\}\!\} \tag{7.6}$$

Proof: ($\Rightarrow$) As the constraint is GAC, all values have support. In particular, $X_i \leftarrow max(X_i)$ has a support $\vec{x_1} \in \{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ and $\vec{y_1} \in \vec{Y}$ where $\{\!\{\vec{x_1}\}\!\} <_m \{\!\{\vec{y_1}\}\!\}$. Any $\vec{x_2} \in \{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ less than or equal to $\vec{x_1}$, and any $\vec{y_2} \in \vec{Y}$ greater than or equal to $\vec{y_1}$, under multiset ordering, support $X_i \leftarrow max(X_i)$. In particular, $min\{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \vec{y} \in \vec{Y}\}$ support $X_i \leftarrow max(X_i)$. We get $min\{\vec{x} \mid x_i = max(X_i) \wedge \vec{x} \in \vec{X}\}$ if all the other variables in $\vec{X}$ take their minimums, and we get $max\{\vec{y} \mid \vec{y} \in \vec{Y}\}$ if all the variables in $\vec{Y}$ take their maximums. Hence, $\{\!\{\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})\}\!\} <_m \{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$.

A dual argument holds for the variables of $\vec{Y}$. As the constraint is GAC, $Y_i \leftarrow min(Y_i)$ has a support $\vec{x_1} \in \vec{X}$ and $\vec{y_1} \in \{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ where $\{\!\{\vec{x_1}\}\!\} <_m \{\!\{\vec{y_1}\}\!\}$. Any $\vec{x_2} \in \vec{X}$ less than or equal to $\vec{x_1}$, and any $\vec{y_2} \in \{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ greater than or equal to $\vec{y_1}$, in particular $min\{\vec{x} \mid \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ support $Y_i \leftarrow min(Y_i)$. We get $min\{\vec{x} \mid \vec{x} \in \vec{X}\}$ if all the variables in $\vec{X}$ take their minimums, and we get $max\{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ if all the other variables in $\vec{Y}$ take their maximums. Hence, $\{\!\{\texttt{floor}(\vec{X})\}\!\} <_m \{\!\{\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})\}\!\}$.

($\Leftarrow$) 7.5 ensures that for all $0 \leq i < n$, $max(X_i)$ is supported, and 7.6 ensures that for all $0 \leq i < n$, $min(Y_i)$ is supported. By theorem 72, the constraint is GAC. QED.

Theorems 71 and 73 together with Theorem 58 yield to the following two propositions:

Proposition 4 *$\vec{X} <_m \vec{Y}$ is disentailed iff $occ(\texttt{floor}(\vec{X})) \geq_{lex} occ(\texttt{ceiling}(\vec{Y}))$.*

Proposition 5 *GAC($\vec{X} <_m \vec{Y}$) iff for all i in $[0,n)$:*

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

We can exploit the similarity between Proposition 3 and 5, and find the tight consistent bounds by making use of the occurrence vectors $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$, the pointers, and the flags. In Theorems 59 to 65, we have $\vec{ox} \leq_{lex} \vec{oy}$. We decide whether a value v in some domain D is consistent or not by first increasing ox_v/oy_v by 1, and then decreasing $min(D)/max(D)$ by 1. The value is consistent for $\vec{X} \leq_m \vec{Y}$ iff the change gives $\vec{ox} \leq_{lex} \vec{oy}$. In Theorems 61 and 65, changing the occurrences gives $\vec{ox} >_{lex} \vec{oy}$. This means that v is inconsistent not only for $\vec{X} \leq_m \vec{Y}$ but also for $\vec{X} <_m \vec{Y}$. In Theorems 60, 63, and 64, however, we initially have $\vec{ox} <_{lex} \vec{oy}$ and changing the occurrences does not disturb the strict lexicographic ordering. This suggests v is consistent also for $\vec{X} <_m \vec{Y}$.

In Theorems 59 and 62, we initially have $\vec{ox} <_{lex} \vec{oy}$, and after the change we obtain either of $\vec{ox} >_{lex} \vec{oy}$, $\vec{ox} = \vec{oy}$, and $\vec{ox} <_{lex} \vec{oy}$. In the first case v is inconsistent, whereas in the third case v is consistent, for both constraints. In the second case, however, v is consistent for $\vec{X} \leq_m \vec{Y}$ but not for $\vec{X} <_m \vec{Y}$. This case arises if we get $\vec{ox}_{u \to \beta} = \vec{oy}_{u \to \beta}$ by the change to the occurrence vectors, and we have either $\beta > l$ and $\vec{ox}_{\beta-1 \to l} = \vec{oy}_{\beta-1 \to l}$, or $\beta = l$. We therefore need to record whether there are any subvectors below β, and if this is the case we need to know whether they are equal. This can easily be done by extending the definition of σ which already tells us whether we have $\beta > l$ and $\vec{ox}_{\beta-1 \to l} >_{lex} \vec{oy}_{\beta-1 \to l}$.

Definition 39 *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leq_{lex} \vec{oy}$, the flag σ is true iff:*

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

Theorems 59 and 62 now declare a value inconsistent if we get $\vec{ox}_{u \to \beta} = \vec{oy}_{u \to \beta}$ when the occurrence vectors change, and we have either $\beta > l$ and $\vec{ox}_{\beta-1 \to l} = \vec{oy}_{\beta-1 \to l}$, or $\beta = l$.

How do we now modify `MsetLeq` to obtain the filtering algorithm `MsetLess`? Theorems 60, 61, 63, 64, and 65 are valid also for $\vec{X} <_m \vec{Y}$. Moreover, Theorems 59 and 62 can easily be adapted for $\vec{X} <_m \vec{Y}$ by changing the definition of σ. Hence, the pruning part of the algorithm need not to be modified, provided that σ is set correctly. Also, by Proposition 4, we need to fail under the new disentailment condition. These suggest we only need to revise `SetPointersAndFlags`, so that we fail whenever we have $\vec{ox} \geq_{lex} \vec{oy}$, and set σ to *true* also when we have $\beta = l$, or $\beta > l$ and $\vec{ox}_{\beta-1 \to l} = \vec{ox}_{\beta-1 \to l}$. This corrects a mistake in [FHK$^+$03] which claims that failing whenever we have $\vec{ox} \geq_{lex} \vec{oy}$ and setting β to $l-1$ as opposed to $-\infty$ are enough to achieve strict multiset ordering.

7.5.2 Entailment

The importance of detecting entailment was discussed in Chapter 5.4.2. We thus introduce another Boolean flag, called *entailed*, which indicates whether $\vec{X} \leq_m \vec{Y}$ is entailed. More formally:

Definition 40 *Given $\vec{X}$ and $\vec{Y}$, the flag* entailed *is set to true iff $\vec{X} \leq_m \vec{Y}$ is true.*

Procedure `SetPointersAndFlags`

```
1   $i := u$;
2   while $i \geq l \ \wedge \ ox_i = oy_i$ do $i := i - 1$;
3   if $i \geq l \ \wedge \ ox_i > oy_i$ then fail;
4   else if $i = l - 1$ then fail;
5   else $\alpha := i$;
    ⋮
14  $\sigma := false$;
    ⋮
16  if $\beta > l$ then
17      $k := \beta - 1$;
18      while $k \geq l \ \wedge \ ox_k = oy_k$ do $k := k - 1$;
19      if $k \geq l \ \wedge \ ox_k > oy_k$ then $\sigma := true$;
20      else if $k = l - 1$ then $\sigma := true$;
    end
21  else if $\beta = l$ then $\sigma := true$;
```

We have seen in Theorems 17 and 49 that both lexicographic ordering constraint and lexicographic ordering with sum constraints are entailed whenever the largest value that $\vec{X}$ can take is less than or equal to the smallest value that $\vec{Y}$ can take under the ordering in concern. This is valid also for the multiset ordering constraint.

Theorem 74 *$\vec{X} \leq_m \vec{Y}$ is entailed iff $\{\!\{\texttt{ceiling}(\vec{X})\}\!\} \leq_m \{\!\{\texttt{floor}(\vec{Y})\}\!\}$.*

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

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

By Theorems 58 and 74, we can detect entailment by lexicographically comparing the occurrence vectors associated with $\texttt{ceiling}(\vec{X})$ and $\texttt{floor}(\vec{Y})$.

Proposition 6 *$\vec{X} \leq_m \vec{Y}$ is entailed iff $occ(\texttt{ceiling}(\vec{X})) \leq_{lex} occ(\texttt{floor}(\vec{Y}))$.*

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

To deal with entailment, we need to modify both `Initialise` and `MsetLeq`. In Algorithm 42, we show how we revise Algorithm 36. We add line 0 to initialise the flag *entailed* to $false$. We replace line 5 of Algorithm 36 with lines 5-7. Before calling `MsetLeq` , we now initialise our new occurrence vectors $occ(\texttt{ceiling}(\vec{X}))$ and $occ(\texttt{floor}(\vec{Y}))$ in a similar way to that of $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$: we create a pair of vectors $\vec{ex}$ and $\vec{ey}$ of length $u - l + 1$ where each ex_i and ey_i are first set to 0. Then, for each value v in $\{\!\{\texttt{ceiling}(\vec{X})\}\!\}$, we increment ex_v by 1. Similarly, for each v in $\{\!\{\texttt{floor}(\vec{Y})\}\!\}$, we

Algorithm 42: `Initialise`

```
    Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
    Result : $occ(\texttt{floor}(\vec{X}))$, $occ(\texttt{ceiling}(\vec{Y}))$, $occ(\texttt{ceiling}(\vec{X}))$, $occ(\texttt{floor}(\vec{Y}))$, and
             $entailed$ are initialised, GAC($\vec{X} \leq_m \vec{Y}$)
  $entailed := false$;
  ⋮
  $\vec{ex} := occ(\texttt{ceiling}(\vec{X}))$;
  $\vec{ey} := occ(\texttt{floor}(\vec{Y}))$;
  MsetLeq;
```

increment ey_v by 1. These vectors are then used in `MsetLeq` to detect entailment. It is possible to maintain $\vec{ex}$ and $\vec{ey}$ incrementally. When the maximum value in some $\mathcal{D}(X_i)$ changes, we update $\vec{ex}$ by incrementing the entry corresponding to new $max(X_i)$ by 1, and decrementing the entry corresponding to old $max(X_i)$ by 1. Likewise, when the minimum value in some $\mathcal{D}(Y_i)$ changes, we update $\vec{ey}$ by incrementing the entry corresponding to new $min(Y_i)$ by 1, and decrementing the entry corresponding to old $min(Y_i)$ by 1.

In Algorithm 43, we show how we modify the filtering algorithm given in Algorithm 37 to deal with the three possible scenarios described above. We add line **A0** where we return if the constraint has already been entailed in the past. Moreover, just before setting our pointers and flags, we check whether the recent modifications that triggered the algorithm resulted in entailment. If this is the case, we first set *entailed* to *true* and then return from the algorithm. Furthermore, we check entailment after the algorithm goes through its variables. Lines **B1-B11** visit the variables of $\vec{X}$ and prune inconsistent values from the upper bounds, affecting $\vec{ex}$. Even if we have $\vec{ex} >_{lex} \vec{ey}$ when the algorithm is called, we might get $\vec{ex} \leq_{lex} \vec{ey}$ just before the algorithm proceeds to the variables of $\vec{Y}$. In such case, we return from the algorithm after setting *entailed* to *true*. As an example, assume we have $\vec{X} \leq_m \vec{Y}$, and `MsetLeq` is called with $\vec{X} = \langle\{1,2\},\{1,2,4\}\rangle$ and $\vec{Y} = \langle\{2,3\},\{2,3\}\rangle$. As 4 in $\mathcal{D}(X_1)$ lacks support, it is pruned. Now we have $\vec{ex} = \vec{ey}$. Alternatively, the constraint might be entailed after the algorithm visits the variables of $\vec{Y}$ and prunes inconsistent values from the lower bounds, affecting $\vec{ey}$. In this case, we return from the algorithm by setting *entailed* to *true*. As an example, assume we also have 0 in $\mathcal{D}(Y_1)$ in the previous example. The constraint is entailed only after the variables of $\vec{Y}$ are visited and 0 is removed.

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

7.5.3 Multiset Disequality and Equality

Given two vectors $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a multiset disequality constraint as $\vec{X} \neq_m \vec{Y}$, which ensures that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ respectively, when viewed as multisets, are different. Two multisets of integers $\mathbf{x} = \{\!\{x_0, \ldots, x_{n-1}\}\!\}$ and $\mathbf{y} = \{\!\{y_0, \ldots, y_{n-1}\}\!\}$ are different $\mathbf{x} \neq \mathbf{y}$ iff $\mathbf{x} <_m \mathbf{y}$ or $\mathbf{y} <_m \mathbf{x}$. We can therefore decompose a multiset disequality constraint $\vec{X} \neq_m \vec{Y}$ by insisting that one of the vectors must be less than the other under multiset ordering:

$$\vec{X} <_m \vec{Y} \ \vee \ \vec{Y} <_m \vec{X}$$

Algorithm 43: `MsetLeq`

```
     Data   : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$
     Result : GAC($\vec{X} \leq_m \vec{Y}$)
A0   if $entailed$ then return;
⇒    if $\vec{ex} \leq_{lex} \vec{ey}$ then $entailed := true$, return;
A1   SetPointersAndFlags;
B1   foreach $i \in [0, n)$ do
B2       if $min(X_i) \neq max(X_i)$ then
B3           if $min(X_i) \geq \alpha$ then
⇒                $ex_{max(X_i)} := ex_{max(X_i)} - 1$, setMax($X_i, min(X_i)$);
⇒                $ex_{max(X_i)} := ex_{max(X_i)} + 1$;
             end
B4           if $max(X_i) \geq \alpha \ \wedge \ min(X_i) < \alpha$ then
B5⇒              $ex_{max(X_i)} := ex_{max(X_i)} - 1$, setMax($X_i, \alpha$);
                 $\vdots$
⇒                $ex_{max(X_i)} := ex_{max(X_i)} + 1$;
             end
         end
     end
⇒    if $\vec{ex} \leq_{lex} \vec{ey}$ then $entailed := true$, return;
C1   foreach $i \in [0, n)$ do
C2       if $min(Y_i) \neq max(Y_i)$ then
C3           if $max(Y_i) > \alpha$ then
⇒                $ey_{min(Y_i)} := ey_{min(Y_i)} - 1$, setMin($Y_i, max(Y_i)$), $ey_{min(Y_i)} := ey_{min(Y_i)} + 1$;
             end
C4           if $max(Y_i) = \alpha \ \wedge \ min(Y_i) \leq \beta$ then
C5               if $ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma$ then
C6⇒                  $ey_{min(Y_i)} := ey_{min(Y_i)} - 1$, setMin($Y_i, \beta$);
                     $\vdots$
⇒                    $ey_{min(Y_i)} := ey_{min(Y_i)} + 1$
                 end
             end
         end
     end
⇒    if $\vec{ex} \leq_{lex} \vec{ey}$ then $entailed := true$, return;
```

Most solvers will delay such a disjunction until one of the disjuncts becomes $false$ (see Chapter 5.5.2). At this point, we can enforce GAC on the other disjunct. As the following theorem shows, we lose in the amount of constraint propagation with such a decomposition.

Theorem 75 *GAC($\vec{X} \neq_m \vec{Y}$) is strictly stronger than $\vec{X} <_m \vec{Y} \ \vee \ \vec{Y} <_m \vec{X}$, assuming that GAC is enforced on the delayed disjunctive constraint.*

Proof: GAC($\vec{X} \neq_m \vec{Y}$) is as strong as its decomposition. To show strictness, consider $\vec{X} = \langle\{0, 1, 2\}\rangle$ and $\vec{Y} = \langle\{1\}\rangle$ where 1 in $\mathcal{D}(X_0)$ cannot be extended to a consistent assignment and therefore $\vec{X} \neq_m \vec{Y}$ is not GAC. Since neither of $\vec{X} <_m \vec{Y}$ and $\vec{Y} <_m \vec{X}$ is $false$ yet, the decomposition does not post any constraint, leaving the vectors unchanged. QED.

Given two vectors $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a multiset equality constraint as $\vec{X} =_m \vec{Y}$, which ensures that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ respectively, when viewed as multisets, are equivalent. Two multisets of integers $\mathbf{x} = \{\!\{x_0, \ldots, x_{n-1}\}\!\}$ and $\mathbf{y} = \{\!\{y_0, \ldots, y_{n-1}\}\!\}$ are equivalent $\mathbf{x} = \mathbf{y}$ iff $\mathbf{x} \leq_m \mathbf{y}$ and $\mathbf{y} \leq_m \mathbf{x}$. We can therefore decompose a multiset equality constraint $\vec{X} =_m \vec{Y}$ by insisting that each of the vectors must be less than or equal to the other under multiset ordering:

$$\vec{X} \leq_m \vec{Y} \ \wedge \ \vec{Y} \leq_m \vec{X}$$

In this way, both of the constraints in the conjunction are posted (see Chapter 5.5.2). We again lose in the amount of constraint propagation with such a decomposition.

Theorem 76 *GAC($\vec{X} =_m \vec{Y}$) is strictly stronger than GAC($\vec{X} \leq_m \vec{Y}$) and GAC($\vec{Y} \leq_m \vec{X}$).*

Proof: GAC($\vec{X} =_m \vec{Y}$) is as strong as its decomposition. To show strictness, consider $\vec{X} = \langle\{0, 1, 2\}\rangle$ and $\vec{Y} = \langle\{0, 2\}\rangle$ where 1 in $\mathcal{D}(X_0)$ cannot be extended to a consistent assignment and therefore $\vec{X} =_m \vec{Y}$ is not GAC. The constraints $\vec{X} \leq_m \vec{Y}$ and $\vec{Y} \leq_m \vec{X}$ are both GAC. The vectors are thus left unchanged. QED.

7.6 Alternative Approaches

There are at least two alternative ways of posting multiset ordering constraints. We can either post arithmetic inequality constraints, or decompose them into other constraints. In this section, we explore each of these approaches and argue that it can be preferable to propagate multiset ordering constraints using our filtering algorithms.

7.6.1 Arithmetic Constraint

We can achieve multiset ordering between two vectors by assigning a weight to each value, summing the weights along each vector, and then insisting the sums to be non-decreasing. Since the ordering is determined according to the maximum value in the vectors, the weight should increase with the value. A suitable weighting scheme was proposed in [KS02], where each value v gets assigned the weight n^v, where n is the length of the vectors. $\vec{X} \leq_m \vec{Y}$ on vectors of length n can then be enforced via the following arithmetic inequality constraint:

$$n^{X_0} + \ldots + n^{X_{n-1}} \leq n^{Y_0} + \ldots + n^{Y_{n-1}}$$

Therefore, a vector containing one element with value v and $n - 1$ 0s is greater than a vector whose n elements are only $v - 1$. This is in fact similar to the transformation of a leximin fuzzy CSP into an equivalent MAX CSP [SFV95]. Strict multiset ordering constraint $\vec{X} <_m \vec{Y}$ is enforced by disallowing equality:

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

BC on such arithmetic constraints does the same pruning as GAC on the original multiset ordering constraints. However, such arithmetic constraints are feasible only for small n and u, where u is the maximum value in the domains of the variables. As n and u get large, n^{X_i} or n^{Y_i} will be a very large number and therefore it might be impossible to implement the multiset ordering constraint. Also, some constraint solvers like ECLiPSe constraint solver 5.3 will simply delay such constraints until all the variable are assigned a value. Consequently, it can be preferable to post and propagate the multiset ordering constraints using our global constraints.

Theorem 77 *GAC($\vec{X} \leq_m \vec{Y}$) and GAC($\vec{X} <_m \vec{Y}$) are equivalent to BC on the corresponding arithmetic constraints.*

Proof: We just consider GAC($\vec{X} \leq_m \vec{Y}$) as the proof for GAC($\vec{X} <_m \vec{Y}$) is entirely analogous. As $\vec{X} \leq_m \vec{Y}$ and the corresponding arithmetic constraint are logically equivalent, BC($\vec{X} \leq_m \vec{Y}$) and BC on the arithmetic constraint are equivalent. By Theorem 55, BC($\vec{X} \leq_m \vec{Y}$) is equivalent to GAC($\vec{X} \leq_m \vec{Y}$). QED.

7.6.2 Decomposition

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

The multiset view of two vectors of integers $\vec{x}$ and $\vec{y}$ are multiset ordered $\{\!\{\vec{x}\}\!\} \leq_m \{\!\{\vec{y}\}\!\}$ iff $occ(\vec{x}) \leq_{lex} occ(\vec{y})$ by Theorem 58. One way of decomposing the multiset ordering constraint $\vec{X} \leq_m \vec{Y}$ is thus insisting that the occurrence vectors associated with the vectors assigned to $\vec{X}$ and $\vec{Y}$ are lexicographically ordered. Such occurrence vectors can be constructed via an extended global cardinality constraint (*gcc*). Given a vector of variables $\vec{X}$ and a vector of values $\vec{d}$, the constraint $gcc(\vec{X}, \vec{d}, \vec{OX})$ ensures that OX_i is the number of variables in $\vec{X}$ assigned to d_i. To ensure multiset ordering, we can enforce lexicographic ordering constraint on a pair of occurrence vectors constructed via *gcc* where $\vec{d}$ is the vector of values that the variables can be assigned to, arranged in descending order, without any repetition:

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

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

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

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

The *gcc* constraint is available in, for instance, ILOG Solver 5.3 [ILO02], SICStus Prolog constraint solver 3.10.1 [SIC03], and FaCiLe constraint solver 1.0 [FaC01]. These constraint solvers propagate the *gcc* constraint using the algorithm proposed in [Rég96]. Among the various filtering algorithms of *gcc*, which maintain either GAC [Rég96][QvBL+03] or BC [QvBL+03][KT03a][KT03b], only the algorithms in [KT03a] and [KT03b] prune values from $\vec{OX}$ and $\vec{OY}$. Even though the algorithm integrated in ILOG Solver 5.3 may also prune the occurrence vectors, this may not always be the case. For instance, when we have $gcc(\langle\{1\}, \{1,2\}, \{1,2\}, \{2\}, \{3,4\}, \{3,4\}\rangle, \langle 4,3,2,1\rangle, \langle\{1\}, \{1\}, \{1,2\},$

$\{1,2,3\}\rangle$, $\vec{OX}$ remains unchanged but 1 in $\mathcal{D}(OX_3)$ is not consistent. This shows that there is currently very limited support in the constraint toolkits to propagate the multiset ordering constraint using the *gcc* decomposition. Also, as the following theorems demonstrate, *gcc* decomposition of a multiset ordering constraint hinders constraint propagation.

Theorem 78 *GAC($\vec{X} \leq_m \vec{Y}$) is strictly stronger than GAC(gcc($\vec{X}, \vec{d}, \vec{OX}$)), GAC(gcc($\vec{Y}$, $\vec{d}, \vec{OY}$)), and GAC($\vec{OX} \leq_{lex} \vec{OY}$), where $\vec{d}$ is the vector of values that the variables can take, arranged in descending order, without any repetition.*

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

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

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

Theorem 79 *GAC($\vec{X} <_m \vec{Y}$) is strictly stronger than GAC(gcc($\vec{X}, \vec{d}, \vec{OX}$)), GAC(gcc($\vec{Y}$, $\vec{d}, \vec{OY}$)), and GAC($\vec{OX} <_{lex} \vec{OY}$), where $\vec{d}$ is the vector of values that the variables can take, arranged in descending order, without any repetition.*

Proof: The example in Theorem 78 shows the strictness. QED.

In Theorem 70, we have established that $occ(\vec{x}) \leq_{lex} occ(\vec{y})$ iff $sort(\vec{x}) \leq_{lex} sort(\vec{y})$. Putting Theorems 58 and 70 together, the multiset view of two vectors of integers $\vec{x}$ and $\vec{y}$ are multiset ordered $\{\!\{\vec{x}\}\!\} \leq_m \{\!\{\vec{y}\}\!\}$ iff $sort(\vec{x}) \leq_{lex} sort(\vec{y})$. This suggests another way of decomposing a multiset ordering constraint $\vec{X} \leq_m \vec{Y}$: we insist that the sorted versions of the vectors assigned to $\vec{X}$ and $\vec{Y}$ are lexicographically ordered. For this purpose, we can use the constraint *sorted* which is available in, for instance, ECLiPSe constraint solver 5.6 [ECL03], SICStus Prolog constraint solver 3.10.1 [SIC03], and FaCiLe constraint solver 1.0 [FaC01]. Given a vector of variables $\vec{X}$, $sorted(\vec{X}, \vec{SX})$ ensures that $\vec{SX}$ is of length n and is a sorted permutation of $\vec{X}$. To ensure multiset ordering, we can enforce lexicographic ordering constraint on a pair of vectors which are constrained to be the sorted versions of the original vectors in descending order:

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

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

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

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

The *sorted* constraint has previously been studied and some BC filtering algorithms have been proposed [BC97][BC00][MT00]. Unfortunately, we lose in the amount of constraint propagation also by the *sort* decomposition of a multiset ordering constraint.

Theorem 80 *GAC($\vec{X} \leq_m \vec{Y}$) is strictly stronger than GAC(sorted($\vec{X}, \vec{SX}$)), GAC(sorted ($\vec{Y}, \vec{SY}$)), and GAC($\vec{SX} \leq_{lex} \vec{SY}$).*

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

$$\begin{array}{rcl} \vec{SX} & = & \langle\{2,3\}, \ \ \{0,2\}\rangle \\ \vec{SY} & = & \langle\{2,3\}, \ \ \ \ \{1\}\rangle \end{array}$$

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

Theorem 81 *GAC($\vec{X} <_m \vec{Y}$) is strictly stronger than GAC(sorted($\vec{X}, \vec{SX}$)), GAC(sorted ($\vec{Y}, \vec{SY}$)), and GAC($\vec{SX} <_{lex} \vec{SY}$).*

Proof: The example in Theorem 80 shows strictness. QED.

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

Theorem 82 *The sort decomposition of $\vec{X} \leq_m \vec{Y}$ is strictly stronger than the gcc decomposition of $\vec{X} \leq_m \vec{Y}$.*

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

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

Theorem 83 *The sort decomposition of $\vec{X} <_m \vec{Y}$ is strictly stronger than the gcc decomposition of $\vec{X} <_m \vec{Y}$.*

Proof: The example in Theorem 82 shows strictness. QED.

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

7.7 Multiple Vectors

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

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

Theorem 84 *GAC($\vec{X}_i \leq_m \vec{X}_j$) for all $0 \leq i < j \leq n-1$ is strictly stronger than GAC($\vec{X}_i \leq_m \vec{X}_{i+1}$) for all $0 \leq i < n-1$.*

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

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

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

Theorem 85 *GAC($\vec{X}_i <_m \vec{X}_j$) for all $0 \leq i < j \leq n-1$ is strictly stronger than GAC($\vec{X}_i <_m \vec{X}_{i+1}$) for all $0 \leq i < n-1$.*

Proof: The example in Theorem 84 shows strictness. QED.

Theorem 86 *GAC($\forall ij\ 0 \leq i < j \leq n-1\,.\ \vec{X}_i \leq_m \vec{X}_j$) is strictly stronger than GAC($\vec{X}_i \leq_m \vec{X}_j$) for all $0 \leq i < j \leq n-1$.*

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

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

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

Theorem 87 *GAC($\forall ij\ 0 \leq i < j \leq n-1\ .\ \vec{X}_i <_m \vec{X}_j$) is strictly stronger than GAC($\vec{X}_i <_m \vec{X}_j$) for all $0 \leq i < j \leq n-1$.*

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

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

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

7.8 Experimental Results

We implemented our global constraints $\leq_m$ and $<_m$ in C++ using ILOG Solver 5.3 [ILO02]. The global constraints encapsulate the corresponding filtering algorithm that either maintains GAC on (strict) multiset ordering constraint or establishes failure at each node of the search tree.

We performed a wide range of experiments to compare our global constraints with the alternative ways of posing multiset ordering constraints which are presented in Section 7.6. Due the absence of the *sorted* constraint in Solver 5.3, the multiset ordering constraint is decomposed via the *gcc* decomposition using the **IloDistribute** constraint. This constraint is the *gcc* constraint, and even though its filtering algorithm [Rég96] maintains GAC, it does not always prune the occurrence vectors.

In the experiments, we have a matrix of decision variables where the rows and/or columns are (partially) symmetric. To break the symmetry, we pose multiset ordering constraints on the symmetric rows or columns, and test two goals when looking for one solution or the optimal solution. First, does our filtering algorithm(s) do more inference in practice than its decomposition? Similarly, is the algorithm more efficient in practice than its decomposition? Second, is it feasible to post the arithmetic constraint? How does our algorithm compare to BC on the arithmetic constraint?

We tested our global constraints on three problem domains: the sport scheduling problem, the rack configuration problem, and the progressive party problem.

The results of the experiments are shown in tables where a "-" means no result is obtained in 1 hour (3600 secs). Whilst the number of choice points gives the number of alternatives explored in the search tree, the number of fails gives the number of incorrect decisions at choice points. The best result of each entry in a table is typeset in bold. If posing multiset ordering on the rows is done via a technique called $Tech$ then we write $Tech$ R. Similarly, posing multiset ordering on the columns using $Tech$ is specified as $Tech$ C. In theory posing multiset ordering constraints between every pair of rows (similarly for columns) leads to more pruning than posing between adjacent rows (see Section 7.7). To see whether this is true in practise, multiset ordering constraints are enforced between the adjacent and all pairs of rows.

The experiments are conducted using ILOG Solver 5.3 on a 1Ghz pentium III processor with 256Mb RAM under Windows XP.

n	Model		Fails	Choice points	Time (secs.)
5	Adjacent Pairs	`MsetLess` C	**1**	**10**	**0.8**
		Arithmetic Constraint C	1	10	0.9
		gcc C	2	11	1.2
	All Pairs	`MsetLess` C	1	10	0.8
7	Adjacent Pairs	`MsetLess` C	**69**	**87**	**0.8**
		Arithmetic Constraint C	69	87	1.3
		gcc C	74	92	1.3
	All Pairs	`MsetLess` C	69	87	1.5
9	Adjacent Pairs	`MsetLess` C	760,973	761,003	**121.3**
		Arithmetic Constraint C	760,973	761,003	2500.0
		gcc C	2,616,148	2,616,176	656.4
	All Pairs	`MsetLess` C	**757,644**	**757,674**	158.3

Table 7.1: Sport scheduling problem: `MsetLess` vs *gcc* decomposition and the arithmetic constraint with column-wise labelling. For one column, we first label the first slots; for the other, we first label the second slots.

Sport Scheduling Problem This was introduced in Chapter 3.2.4. In Figure 3.9, one way of modelling the problem is presented. The (extended) weeks over which the tournament is held, as well the periods are indistinguishable. The rows and the columns of T and G are therefore symmetric. Note that we treat T as a 2-d matrix where the rows represent the periods and columns represent the (extended) weeks, and each entry of the matrix is a pair of variables.

Consider the rows of T which represent the periods. The global cardinality constraints posted on the rows ensure that each of $1 \dots n$ occur exactly twice in every row. This means that in any solution to the problem, the rows are equal, when viewed as multisets. Now consider the columns of T which denote the (extended) weeks. The *all-different* constraints posted on the columns state that each column is a permutation of $1 \dots n$. This suggests that in any solution to the problem, also the columns are equal when viewed as multisets. Therefore, we cannot utilise multiset ordering constraints to break row and/or column symmetry of this model of the problem.

Scheduling a tournament between n teams means arranging $n(n-1)/2$ games. The model described in Figure 3.9 assumes n is an even number. If n is an odd number instead then we can still schedule $n(n-1)/2$ games provided that the games are played over n weeks and each week is divided into $(n-1)/2$ periods. The problem now requires that each team plays at most once a week, and every team plays exactly twice in the same period over the tournament. This version of the problem can be modelled using the original model in Figure 3.9, as the *all-different* constraints on the rows and the cardinality constraints on the columns enforce the new problem constraints.

We can now post multiset ordering constraints on the columns of T to break column symmetry. Since the games are all different, no pair of columns can be equal, when viewed as multisets. Hence, we insist that the columns corresponding to the n weeks $\vec{C_0}, \vec{C_1}, \dots, \vec{C}_{n-1}$ are strict multiset ordered: $\vec{C_0} <_m \vec{C_1} \dots <_m \vec{C}_{n-1}$. We enforce the multiset ordering constraints by either using our filtering algorithm `MsetLess`, or the *gcc* decomposition, or the arithmetic constraint.

As the multiset ordering constraints are posted on the columns, we instantiate T

n	Model		Fails	Choice points	Time (secs.)
5	Adjacent Pairs	`MsetLess` C	**12**	**19**	**0.8**
		Arithmetic Constraint C	12	19	1.1
		gcc C	15	22	0.9
	All Pairs	`MsetLess` C	12	19	0.8
7	Adjacent Pairs	`MsetLess` C	249	265	**0.8**
		Arithmetic Constraint C	249	265	1.8
		gcc C	?	?	?
	All Pairs	`MsetLess` C	**230**	**246**	1.1
9	Adjacent Pairs	`MsetLess` C	525,755	525,789	**71.0**
		Arithmetic Constraint C	525,755	525,789	1,427.4
		gcc C	?	?	?
	All Pairs	`MsetLess` C	**476,834**	**476,868**	80.0

Table 7.2: Sport scheduling problem: `MsetLess` vs *gcc* decomposition and the arithmetic constraint with column-wise labelling. For one column, we label top-down; for the other, we label bottom-up. The first row is filled after the first two columns are instantiated. A "?" means the solution obtained does not satisfy the multiset ordering constraints.

column-by-column in two different ways: (1) We explore all the columns top-down. For one column, we first label the first slots; for the other, we first label the second slots. (2) For one column, we label top-down; for the other, we label bottom-up. The first row is filled after the first two columns are instantiated. The results of the experiments are shown in Tables 7.1 and 7.2.

In Table 7.1, we observe that `MsetLess` is superior to the *gcc* decomposition. As the problem gets more difficult, `MsetLess` does more pruning and solves the problem quicker. In Table 7.2, we are unable to compare `MsetLess` and the *gcc* decomposition except for the first instance, because the latter gives solutions which do not satisfy the multiset ordering constraints. We here witness that if the filtering algorithm of the *gcc* constraint does not always prune the occurrence vectors, then the *gcc* decomposition of the multiset ordering constraint is not correct.

Both tables indicate a substantial gain in efficiency by using `MsetLess` in preference to the arithmetic constraint. Even though the same search tree is created by the two, constructing and propagating the arithmetic constraints are much more costly than running `MsetLess` to solve the multiset ordering constraints. We have shown that in theory enforcing multiset ordering constraints between all pair of columns can increase the amount of pruning. We see some evidence of this in the tables; however, the gain in the amount of constraint propagation results in longer run-times.

In summary, for the sport scheduling problem, posting and propagating the multiset ordering constraints using `MsetLess` is clearly profitable over decomposing the constraints or posting the arithmetic constraints, with static labelling heuristics that instantiate the matrix column-wise. The amount of constraint propagation can be increased by enforcing the multiset constraints between every pair of columns, but this carries a penalty in the cost of constraint propagation.

Rack Configuration Problem This was introduced in Chapter 3.2.3. One way of modelling the problem is given in Figure 3.5. The 2-d matrix C has partial row symmetry,

Rack Model	Power	Connectors	Price
1	150	8	150
2	200	16	200

Card Type	Power
1	20
2	40
3	50
4	75

Table 7.3: Rack model and card type specifications in the rack configuration problem [ILO02].

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

Table 7.4: Demand specification for the cards in the rack configuration problem.

because the racks of the same rack model are indistinguishable and therefore their card assignments can be interchanged.

To break the row symmetry, we post multiset ordering constraints on the rows conditionally. Given two racks i and j where $i < j$, we enforce that the rows corresponding to such racks are multiset ordered if the racks are assigned the same rack model. That is:

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

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

We consider several instances of the rack configuration problem, which are described in Tables 7.3 and 7.4. In the experiments, we use the rack model and card type specifications given in [ILO02], but we vary the demand of the card types randomly. As in [ILO02], we search for the optimal solution by exploring the racks in turn. For each $j \in \mathcal{R}acks$, we first instantiate R_j and then label the jth row in C in increasing order of $i \in \mathcal{C}types$ to determine how many cards from each card type are plugged into the rack j.

The results of the experiments are shown in Table 7.5. `MsetLeq` is clearly much more efficient than the arithmetic constraint on every instance considered. Enforcing the multiset ordering constraints between every pair of symmetric rows as opposed to the adjacent ones does not give any worthwhile reductions in the size of the search tree. Any gain is compensated by the increased run-times.

In summary, `MsetLeq` is evidently superior to the arithmetic constraint for the rack configuration problem. Due to the restrictions in the implementation of the **IloDis-**

	Adjacent Pairs				All Pairs		
Inst.	`MsetLeq` R			Arithmetic Constraint R	`MsetLeq` R		
#	Fails	Choice points	Time (secs.)	Time (secs.)	Fails	Choice points	Time (secs.)
1	**3,052**	**3,063**	**0.2**	2.8	3,052	3,063	0.2
2	15,650	15,657	**0.6**	15.6	**15,554**	**15,601**	0.8
3	**3,990**	**3,999**	**0.2**	2.6	3,990	3,999	0.3
4	**8,864**	**8,872**	**0.4**	7.1	8,864	8,872	0.5
5	40,851	40,858	**1.5**	41.3	**40,549**	**40,556**	2.0
6	42,013	42,026	**1.6**	35.2	**41,790**	**41,803**	2.0

Table 7.5: Rack configuration problem: `MsetLeq` vs the arithmetic constraint.

tribute constraint, we cannot judge how much more propagation `MsetLeq` achieves over the *gcc* decomposition. With the static labelling heuristic that first instantiates the model of a rack and then generates the number of cards plugged into the rack, imposing the multiset ordering constraints between every pair of symmetric rows gives additional but modest pruning, with a penalty in the cost of propagation.

Progressive Party Problem This was introduced in Chapter 3.2.5. In Figure 3.10, one way of modelling the problem is given. The time periods as well as the guests with equal crew size are indistinguishable. Hence, this model of the problem has partial row symmetry between the indistinguishable guests of H, and column symmetry.

Due to the problem constraints, no pair of rows/columns can be equal, but they can be equal when viewed as multisets. We address only the row or column symmetry, and insist that the symmetric rows or columns are multiset ordered. Given a set of indistinguishable guests $\{g_i, g_{i+1}, \ldots, g_j\}$, we break the partial row symmetry by enforcing that the rows corresponding to such guests $\vec{R_i}, \vec{R}_{i+1}, \ldots, \vec{R_j}$ are multiset ordered: $\vec{R_i} \leq_m \vec{R}_{i+1} \ldots \leq_m \vec{R_j}$. To break the column symmetry, we enforce that the columns $\vec{C_0}, \vec{C_1}, \ldots, \vec{C}_{p-1}$ corresponding to the p time periods are multiset ordered: $\vec{C_0} \leq_m \vec{C_1} \ldots \leq_m \vec{C}_{p-1}$. We impose the multiset ordering constraints by either using our filtering algorithm `MsetLeq`, or the *gcc* decomposition, or the arithmetic constraint.

We consider several instances of the progressive party problem, including the one mentioned in Chapter 3.2.5. 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. Note that posing the arithmetic constraint on the columns of H is not feasible for this problem. The largest value a variable can get is 29^{13}, which is larger than 2^{31}, the maximum integer size allowed in Solver 5.3. Table 7.6 shows the data. The last column of Table 7.6 gives the percentage of the total capacity used, which is a measure of constrainedness [Wal99].

As in [SBHW96], we give priority to the largest crews, so the guest boats are ordered in descending order of their size. Also, when assigning a host to a guest, we try a value first which is most likely to succeed. We therefore order the host boats in descending order of their spare capacity. When we post multiset ordering constraints only on the columns to break column symmetry, we instantiate H column-wise. Similarly, when we post multiset ordering constraints only on the rows to break row symmetry, we instantiate H row-wise.

The results of the experiments are shown in Tables 7.7 and 7.8. Note that all the

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 7.6: Instance specification for the progressive party problem.

Instance	`MsetLeq` C			*gcc* C		
#	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
1	**7,038**	**7,168**	**3.1**	8,387	8,517	3.0
2	**127**	**253**	**0.5**	130	257	0.5
3	**40,166**	**40,297**	**20.2**	43,808	43,939	22.3
4	-	-	-	?	?	?
5	**241,945**	**242,075**	**95.0**	?	?	?
6	**40,166**	**40,297**	**20.5**	43,808	43,939	22.2
7	-	-	-	?	?	?
8	-	-	-	?	?	?
9	**241,945**	**242,075**	**95.0**	?	?	?

Table 7.7: Progressive party problem: `MsetLeq` vs *gcc* decomposition with column-wise labelling. A "?" means the solution obtained does not satisfy the multiset ordering constraints.

problem instances are solved for 5 time periods. Even though in theory posing multiset ordering constraints between every pair of symmetric rows (similarly for columns) leads to more pruning than posing between the adjacent ones, we could not see any evidence of this in this problem. We therefore report the results of posting multiset ordering constraints just between the adjacent rows.

We notice in Table 7.7 that the *gcc* decomposition gives incorrect results for half of the instances, when we post the multiset ordering constraints on the columns and instantiate the matrix column-wise. We once again witness that if the filtering algorithm of the *gcc* constraint does not always prune the occurrence vectors, then the *gcc* decomposition of the multiset ordering constraint is not correct. Also, some instances are very difficult to solve within an hour limit with this labelling heuristic. According to the results of the rest of the instances, `MsetLeq` is superior to the *gcc* decomposition, as `MsetLeq` does more inference, reducing the size of the search tree. However, the fact that the *gcc* decomposition fails to give correct results for some instances raises doubt on the accuracy of the results of the remaining instances. Since an incorrect solution is due to success at a node of the search tree where a failure should be established, several failures could have been missed during

Instance	`MsetLeq` R			Arithmetic Constraint R	*gcc* R		
#	Fails	Choice points	Time (secs.)	Time (secs.)	Fails	Choice points	Time (secs.)
1	**10,839**	**10,963**	**8.3**	16.0	20,367	20,491	11.6
2	**56,209**	**56,327**	**46.8**	123.7	57,949	58,067	48.6
3	**27,461**	**27,575**	**17.1**	39.1	42,741	42,855	20.5
4	**420,774**	**420,888**	**280.5**	621.7	586,902	587,016	298.1
5	-	-	-	-	-	-	-
6	**5,052**	**5,170**	**3.8**	7.3	8,002	8,123	4.3
7	**86,432**	**86,547**	**65.5**	135.2	128,080	128,195	75.7
8	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-

Table 7.8: Progressive party problem: `MsetLeq` vs *gcc* decomposition and the arithmetic constraint with row-wise labelling.

search even if the solution found satisfies the multiset ordering constraints.

In Table 7.8, we contrast `MsetLeq` with both the *gcc* decomposition and the arithmetic constraint, when we post the multiset ordering constraints on the rows and instantiate the matrix row-wise. This time the *gcc* decomposition gives correct results, and the instances that could not be solved previously are now solved in relatively shorter times. The results show that `MsetLeq` maintains a significant advantage over the *gcc* decomposition. The solutions to the instances that can be solved within an hour limit are found quicker with much less failures. As the multiset ordering constraints are posted on the symmetric rows and the rows are of length 5, it is now feasible to post the arithmetic constraint. On the other hand, `MsetLeq` remains to be the most efficient way of propagating the multiset ordering constraints.

In summary, for the progressive party problem, it is feasible to post the arithmetic constraint only on the rows of the matrix for the instances that we considered. However, it is much more costly to post and propagate the arithmetic constraint than using `MsetLeq` to solve the multiset ordering constraints. The *gcc* decomposition is inferior to `MsetLeq` when the multiset ordering constraints are posted along the rows (resp. columns) and the matrix is instantiated row-wise (resp. column-wise).

7.9 Implementation

We implemented our global constraints in C++ using ILOG Solver 5.3 [ILO02]. As the constraint system is an event based system, it is crucial to decide at which propagation events to wake up the constraints and when to propagate the constraints, so as to integrate them into the constraint system. In this section, we go through the details of the implementation of $\leq_m$, but a similar procedure has been adapted also for $<_m$.

As discussed in Chapter 5.10, three propagation events are available in Solver for an integer variable: **whenValue**, **whenRange**, and **whenDomain**. When $\vec{X} \leq_m \vec{Y}$ is GAC, every value in $\mathcal{D}(X_i)$ is supported by $\langle min(X_0), \ldots, min(X_{i-1}), min(X_{i+1}), \ldots, min(X_{n-1})\rangle$ and $\langle max(Y_0), \ldots, max(Y_{n-1})\rangle$; every value in $\mathcal{D}(Y_i)$ is supported by $\langle min(X_0), \ldots, min(X_{n-1})\rangle$ and $\langle max(Y_0), \ldots, max(Y_{i-1}), max(Y_{i+1}), \ldots, max(Y_{n-1})\rangle$. Any modification to the variables affecting the bounds should wake up the constraint. So, we attach **whenRange** propagation event to all the variables of the vectors. This is in fact not exactly what we would like to do, because we need to wake up the constraint only when $min(X_i)$ or $max(Y_i)$ of some i in $[0, n)$ changes. On the other hand, any event triggered by modifications to $max(X_i)$ or $min(Y_i)$ can easily be discarded by watching the domain-delta[1].

An important issue is when to propagate the constraint. As discussed in Chapter 5.10, two ways of propagating a constraint are available in Solver: (1) respond to each propagation event individually; (2) wait until all propagation events accumulate. The first method of propagation is preferable over the second with a highly incremental algorithm in which the data structures can be restored easily and efficiently at every propagation event. In the case of our global constraint $\leq_m$, the filtering algorithm has both incremental and non-incremental nature. It is incremental in terms of keeping the occurrence vectors $\vec{ox}$ and $\vec{oy}$ up-to-date. These vectors are created when the constraint is first posted, and each time $min(X_i)$ or $max(Y_i)$ of some i in $[0, n)$ changes, the necessary entries in the vectors are revised in constant time. The algorithm is non-incremental in terms of computing the pointers and flags. Every time the algorithm is called, α, β, γ, and σ are recomputed. This computation requires, in the worst case, a complete scan of the occurrence vectors. Even though computing the pointers and flags at every propagation step does not seem to be too costly, responding to each event individually means visiting all the $2n$ variables after every event. To avoid this, we prefer the second method of propagation, in which the pointers and flags are recomputed and the variables of $\vec{X}$ and $\vec{Y}$ are examined once after all the events accumulate.

Having decided at which propagation events to wake up the constraint and when to propagate the constraint, we first implement **post** which is called by Solver when the constraint is first posted. This procedure initialises the data structures (e.g. the occurrence vectors), defines on which events the constraint propagates, and then propagates the constraint. Since the occurrence vectors are maintained incrementally, they are defined in Solver as reversible objects. To be able to propagate selectively, we post different event demons on the variables of $\vec{X}$ and $\vec{Y}$. However, the propagation algorithm is implemented using **propagate**, because every demon now delays its response, and one propagation takes place after all the events are gathered.

`Initialise` in Algorithm 36 is in fact what **post** deploys except that after initialising the occurrence vectors, we attach an event demon to a **whenRange** event for each variable of the vectors.

Procedure post

```
1    $l := min(\{\{\texttt{floor}(\vec{X})\}\} \cup \{\{\texttt{floor}(\vec{Y})\}\})$;
     ⋮
4    $\vec{oy} := occc(\texttt{ceiling}(\vec{Y}))$;
5    foreach $i \in [0, n)$ do
5.1      $X_i$.whenRange(EventDemonForX($i$));
5.2      $Y_i$.whenRange(EventDemonForY($i$));
     end
```

[1] The domain-delta is a special set in Solver where the modifications of the domain of a variable are stored. This domain-delta can be accessed during the propagation of the constraints posted on the variable.

`EventDemonForX`(i) is triggered whenever $min(X_i)$ or $max(X_i)$ of some i in $[0, n)$ is modified. In line 1, events that are not of interest are filtered. In Solver, $\delta(V)$ (reads as domain-delta) is a special set where the modifications of $\mathcal{D}(V)$ are stored. The member function **getMinDelta()** returns $min(X_i) - min(\delta(X_i))$. To know whether the propagation event that triggered the demon is due to a change to $min(X_i)$, it suffices to test the value of X_i.**getMinDelta()**. If the test returns 0 then $min(X_i)$ has not been modified. In line 1.5, the demon calls Solver's **push** so that the response to each propagation event of interest is delayed. Beforehand, $\vec{ox}$ is updated. In line 1.1, the previous minimum of $\mathcal{D}(X_i)$ is extracted as dec using Solver's member function **getOldMin()** which returns $min(\delta(X_i))$. The new minimum of $\mathcal{D}(X_i)$ is recorded as inc in line 1.2. Then, in lines 1.3 and 1.4, ox_{dec} is decremented by 1 and ox_{inc} is incremented by 1, respectively. This is because while the modification to $min(X_i)$ reduced the number of occurrences of dec in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ by 1, it increased the number of occurrences of inc in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ by 1.

```
Procedure EventDemonForX($i$)
1    if $X_i$.getMinDelta()$\neq 0$ then
1.1      $dec := X_i$.getOldMin();
1.2      $inc := X_i$.getMin();
1.3      $ox_{dec} := ox_{dec} - 1$;
1.4      $ox_{inc} := ox_{inc} + 1$;
1.5      push();
     end
```

`EventDemonForY`(i) is similar to `EventDemonForX`(i) except that now the events triggered by modifications to $min(Y_i)$ are discarded, using the member function **getMaxDelta()**. Also, before pushing the event to the queue, $\vec{oy}$ is updated by incrementing and decrementing the entries corresponding to the old and new maximum of $\mathcal{D}(Y_i)$ by 1, respectively. The previous maximum of $\mathcal{D}(Y_i)$ is obtained by using Solver's member function **getOldMax()** which returns $max(\delta(Y_i))$.

```
Procedure EventDemonForY($i$)
1    if $Y_i$.getMaxDelta()$\neq 0$ then
1.1      $dec := Y_i$.getOldMax();
1.2      $inc := Y_i$.getMax();
1.3      $oy_{dec} := oy_{dec} - 1$;
1.4      $oy_{inc} := oy_{inc} + 1$;
1.5      push();
     end
```

Solver automatically aggregates all such calls to **push** into a single invocation of the **propagate** procedure which propagates the constraint. Note that **propagate** is also automatically called once the constraint is posted.

Procedure **propagate** calls the filtering algorithm.

```
Procedure propagate
1    MsetLeq;
```

7.10 Summary

In this chapter, we have presented some global constraints for multiset orderings which are useful for breaking row and column symmetries of a matrix of decision variables.

There are at least two ways of decomposing a multiset ordering constraint. Both of the decompositions are inferior to maintaining GAC on the constraint. Alternatively, by assigning a weight to each value in the domain of the variables, one can pose arithmetic inequality constraints to ensure multiset orderings. However, this approach is not feasible when the vectors and domain sizes are large, which is often the case. We have therefore developed an efficient filtering algorithm which either proves that $\vec{X} \leq_m \vec{Y}$ is disentailed, or ensures GAC on $\vec{X} \leq_m \vec{Y}$. The algorithm runs in time $O(nb + d)$, where b is the cost of adjusting the bounds of a variable, and $d = u - l + 1$, u is $max(\{\!\{\texttt{ceiling}(\vec{X})\}\!\} \cup \{\!\{\texttt{ceiling}(\vec{Y})\}\!\})$ and l is $min(\{\!\{\texttt{floor}(\vec{X})\}\!\} \cup \{\!\{\texttt{floor}(\vec{Y})\}\!\})$. If $d \ll n$ then the algorithm is $O(n * b)$. We expect this as the number of distinct values in a multiset is typically less than its cardinality to permit repetition. Since adjusting the bounds is a constant time operation, b is always a constant. The complexity of the algorithm is optimal as there are $O(n)$ variables to consider.

The filtering algorithm exploits two theoretical results. The first reduces the problem to testing support for upper bounds of $\vec{X}$ and lower bounds of $\vec{Y}$ on suitable ground vectors. The second reduces these tests to lexicographically ordering suitable occurrence vectors. The pointers and flags defined on the occurrence vectors give us a linear time complexity as opposed to quadratic when each variable is naively examined in turn. The algorithm can easily be modified for $\vec{X} <_m \vec{Y}$ by changing the definition of one of the flags. Moreover, the ease of maintaining the occurrence vectors incrementally helps detect entailment in a simple and dual manner to detecting disentailment.

Another variant of the algorithm is when $d \gg n$. This algorithm exploits a theoretical result which reduces the tests for identifying support to lexicographically ordering suitable sorted vectors. The complexity is then independent of domain size and is $O(n\ log(n))$, as the cost of sorting dominates.

We have studied the propagation of multiset equality and disequality constraints using multiset ordering constraints, and demonstrated that decomposing a chain of multiset ordering constraints between adjacent or all pairs of vectors hinders constraint propagation.

Our experiments on the sport scheduling problem, the rack configuration problem, and the progressive party problem confirm our theoretical studies. The results of the experiments can be summarised as follows. First, even if it is feasible to post the arithmetic constraint, it is much more efficient to propagate the multiset ordering constraints using our filtering algorithm. Second, decomposing the multiset constraints carries penalty either in the amount or the cost of constraint propagation. Also, the *gcc* decomposition can give incorrect results if the filtering algorithm of the *gcc* constraint does not always prune the occurrence vectors. Finally, enforcing multiset ordering constraints between all pair of rows (resp. for columns) can increase the amount of pruning, but this usually comes with a cost, resulting in longer run-times.

Chapter 8

Symmetry Breaking with Ordering Constraints

8.1 Introduction

In Chapter 4, we have proposed some ordering constraints for breaking row and column symmetries. In particular, we have shown that we can consistently enforce lexicographic ordering or multiset ordering constraints on the symmetric rows and columns. These constraints can also be combined to obtain new symmetry breaking constraints. In this chapter, we perform a wide range of experiments to evaluate our hypothesis: (1) the ordering constraints are often not redundant and bring additional pruning; (2) the ordering constraints are effective in breaking row and column symmetries as they significantly reduce the size of the search tree and the time to solve the problem; (3) the ordering constraints that are incomparable in theory are incomparable also in practice.

The experiments are performed using some of the problems discussed in Chapter 3. In each experiment, we have a matrix of decision variables where the rows and/or columns are (partially) symmetric. To break the symmetry, we post ordering constraints on the rows and/or columns, and search for one solution or the optimal solution. Symmetry breaking is important even if we are interested in only one solution as we may explore many failed and symmetrically equivalent states during search. Finding an optimal solution requires searching the entire search space, and thus can be seen as looking for all solutions. If the matrix model has previously been studied, we use the suggested labelling heuristic(s). Otherwise, we use the labelling heuristic tuned by our initial experimentation which we did as follows. We tried many static orderings as well as the *smallest-domain first* principle for multi-valued domains. Surprisingly, this dynamic heuristic did not work well for any of the problems considered. Hence, we picked the best static heuristic for each problem.

The results of the experiments are shown in tables and column charts, where a "-" and a missing column mean no result is obtained in 1 hour (3600 secs). Whilst the number of choice points is the number of alternatives explored in the search tree, the number of fails is the number of incorrect decisions at choice points. The best result of each entry in a table is typeset in bold. For ease of presentation, we write $\preceq R$ for posting the ordering constraint $\preceq$ on the rows, $\preceq C$ for posting $\preceq$ on the columns, and $\preceq RC$ for posting $\preceq$ on the rows and columns. The ordering constraints are enforced just between the adjacent rows and/or columns as we have found it not worthwhile to post them between all pairs.

The experiments are conducted using ILOG Solver 5.3 on a 1Ghz pentium III processor with 256Mb RAM under Windows XP.

8.2 Social Golfers Problem

This was introduced in Chapter 3.2.4. Figure 3.8 shows a modification of the set variable based model given in Figure 3.7. In this model, a 3-d 0/1 matrix T of $\mathcal{G}roups \times \mathcal{W}eeks \times \mathcal{G}olfers$ is used to decide which golfer plays in which group of which week.

The matrix T has symmetry along each of the three dimensions: the groups are indistinguishable, and so are the weeks and the golfers. As the contents of a group from one week to the other weeks are independent of each other, one way of reducing symmetry in this model is to:

- set the golfers of the first week;
- partition the golfers of the last group of the first week into the last s groups in the other weeks.

As an example, assume we wish to schedule $\mathcal{G}olfers = \{1, 2, \ldots, 12\}$ to play in 4 groups of size 3 for 3 weeks. We set the first week by placing the first set of 3 players to the last group, the next set of 3 players to the group before last, and so on. This does not conflict with the problem constraints as the golfers can be scheduled arbitrarily in the first week. The first 3 players, which play together in the last group of the first week, cannot meet each other again and therefore need to be in different groups in the following weeks. We partition these golfers into the last 3 groups in the other weeks. Again, this partitioning is consistent with the problem constraints, as each of these golfers can be placed in an arbitrary group provided that they are all in different groups. Consequently, we get:

$$\begin{array}{rrrrr}
T_{i,j} & & \rightarrow \mathcal{G}roups \rightarrow & & \\
\downarrow & \{12,11,10\} & \{9,8,7\} & \{6,5,4\} & \{3,2,1\} \\
\mathcal{W}eeks & \{?,?,?\} & \{?,?,3\} & \{?,?,2\} & \{?,?,1\} \\
\downarrow & \{?,?,?\} & \{?,?,3\} & \{?,?,2\} & \{?,?,1\}
\end{array}$$

where a "?" is used to designate a variable, and the golfers playing in each group of each week are grouped together in a set, for ease of presentation. We refer to such constraints as basic symmetry breaking constraints.

Even though the basic symmetry breaking constraints help reduce some of the symmetry, it is possible to break more symmetry by also enforcing ordering constraints on each of the three dimensions of T. Due to the problem constraints, no pair of groups, weeks, and golfers can have identical assignments. To break the symmetry between the weeks, we insist that the 2-d slices flattened onto vectors $\vec{W}_0, \vec{W}_1, \ldots, \vec{W}_{w-1}$ which represent the given w weeks are strict lexicographically ordered: $\vec{W}_0 <_{lex} \vec{W}_1 \ldots <_{lex} \vec{W}_{w-1}$. As for the symmetry between the golfers, we enforce that the 2-d slices flattened onto vectors $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_{g*s-1}$ which represent the given $g * s$ golfers are strict lexicographically ordered: $\vec{G}_0 <_{lex} \vec{G}_1 \ldots <_{lex} \vec{G}_{g*s-1}$. Considering that the contents of a group from one week to the next are independent of each other and the groups within a week must be disjoint, we tackle the symmetry between the groups by imposing strict lexicographic ordering constraints on the columns (i.e. the group dimension) of each week:

$$\forall j \in \mathcal{W}eeks\,.\ \vec{T}_{0,j} <_{lex} \vec{T}_{1,j} \ldots <_{lex} \vec{T}_{g-1,j}$$

as opposed to flattening the 2-d slices describing the groups onto vectors and constraining these vectors to be strict lexicographically ordered. We refer to the model in which the strict lexicographic ordering constraints are imposed together with the basic symmetry

Instance #	w, g, s	Instance #	w, g, s
1	$2, 5, 4$	11	$4, 9, 4$
2	$2, 6, 4$	12	$5, 4, 3$
3	$2, 7, 4$	13	$5, 5, 4$
4	$2, 8, 5$	14	$5, 7, 4$
5	$3, 5, 4$	15	$5, 8, 3$
6	$3, 6, 4$	16	$6, 4, 3$
7	$3, 7, 4$	17	$6, 5, 3$
8	$4, 5, 4$	18	$6, 6, 3$
9	$4, 6, 5$	19	$7, 5, 3$
10	$4, 7, 4$	20	$7, 5, 5$

Table 8.1: The instances used in the social golfers problem.

breaking constraints as Model 1, and to the model in which no ordering constraints but only the basic symmetry breaking constraints imposed as Model 2. Note that the basic symmetry breaking constraints and the lexicographic ordering constraints do not conflict.

An alternative way of breaking symmetry using strict lexicographic ordering constraints is to impose strict anti-lexicographic ordering constraints on the slices representing the weeks and on the slices representing the golfers, as well as on the columns of every slice corresponding to a week. In this case, we modify the basic symmetry breaking constraints so that they agree with the strict anti-lexicographic ordering constraints. First, we set the first week starting from the *first* group. Second, we partition the golfers of the *first* group of the first week into the *first* s groups in the other weeks. In the example of scheduling $\mathcal{G}olfers = \{1, 2, \ldots, 12\}$ to play in 4 groups of size 3 for 3 weeks, we get:

$$\begin{array}{rcccc} T_{i,j} & & \rightarrow \mathcal{G}roups \rightarrow & & \\ \downarrow & \{1,2,3\} & \{4,5,6\} & \{7,8,9\} & \{10,11,12\} \\ \mathcal{W}eeks & \{1,?,?\} & \{2,?,?\} & \{3,?,?\} & \{?,?,?\} \\ \downarrow & \{1,?,?\} & \{2,?,?\} & \{3,?,?\} & \{?,?,?\} \end{array}$$

by posting the modified basic symmetry breaking constraints that are consistent with the strict anti-lexicographic ordering constraints. We refer to the model in which the strict anti-lexicographic ordering constraints are imposed together with the modified basic symmetry breaking constraints as Model 3, and to the model in which no ordering constraints but only the modified basic symmetry breaking constraints imposed as Model 4.

Can we utilise multiset ordering constraints to break symmetry in this model of the problem? Consider the dimension of T which describes the weeks. Each slice represents a week and is a 2-d 0/1 matrix of $\mathcal{G}roups \times \mathcal{G}olfers$. Sum constraints are posted on the columns (i.e. the group dimension) of every week to ensure that every group is of size s:

$$\forall j \in \mathcal{W}eeks \,.\, \forall i \in \mathcal{G}roups \,.\, \sum_{k \in \mathcal{G}olfers} T_{i,j,k} = s$$

In any solution to the problem, the columns within each week are thus equal when viewed as multisets. A consequence of this is that each slice corresponding to a week is constrained to have $g * s$ 1s. In any solution to the problem, the 2-d slices flattened onto vectors which represent the weeks are thus also equal when viewed as multisets. Now consider the dimension of T which represent the golfers. Each slice corresponds to a golfer and is a 2-d 0/1 matrix of $\mathcal{G}roups \times \mathcal{W}eeks$. Sum constraints on the rows (i.e. the week dimension)

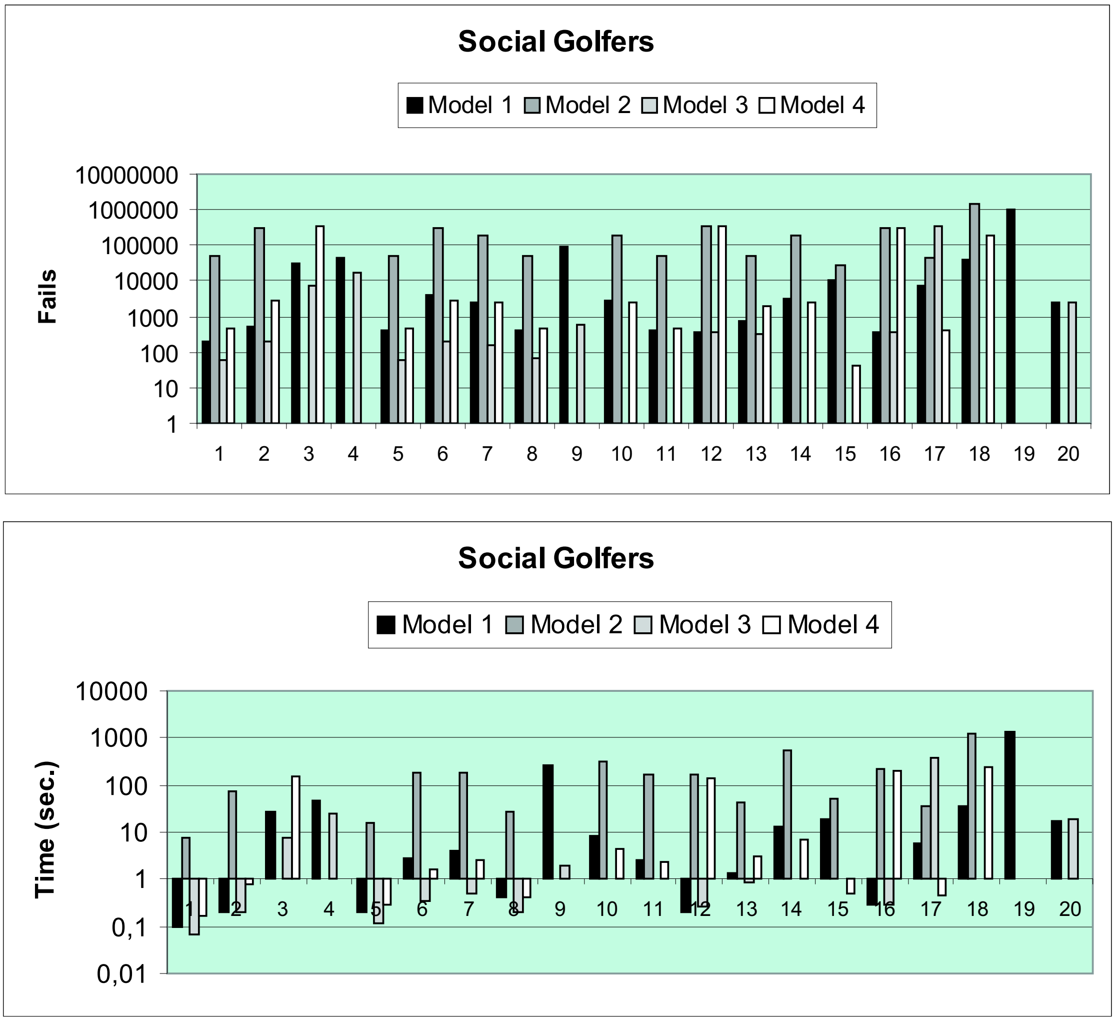

Figure 8.1: Social golfers problem by filling T week-by-week, setting each golfer in turn.

of every such slice ensure that each golfer plays once a week:

$$\forall k \in \mathcal{G}olfers\,.\ \ \forall i \in \mathcal{W}eeks\,.\ \sum_{j \in \mathcal{G}roups} T_{i,j,k} = 1$$

Since there are w rows in each slice, every slice is constrained to have w 1s. This means that in any solution to the problem, the 2-d slices flattened onto vectors which represent the golfers are also equal when viewed as multisets. Therefore, we cannot make use of multiset ordering constraints to break symmetry in this model of the problem.

In our experiments, we consider several instances of the social golfers problem (see Table 8.1) taken from [FFH$^+$02], [Pug02c], and [CSP], and compare Model 1, Model 2, Model 3, and Model 4. We search for a solution by exploring the 2-d slices representing the weeks in turn. For each week, we label the matrix along its rows (i.e. golfer-by-golfer) from top to bottom, exploring the domain of each variable in ascending order.

The results of the experiments are shown in Figure 8.1 and summarised in Figure 8.2 where we partition the instances according to the model(s) that solves them with the least search effort. Among the 20 instances we have considered, 50% are best solved by Model 3 with significant gains, both in the size of search trees and run-times, over the other models. Model 1, on the other hand, is superior to all the remaining models on only

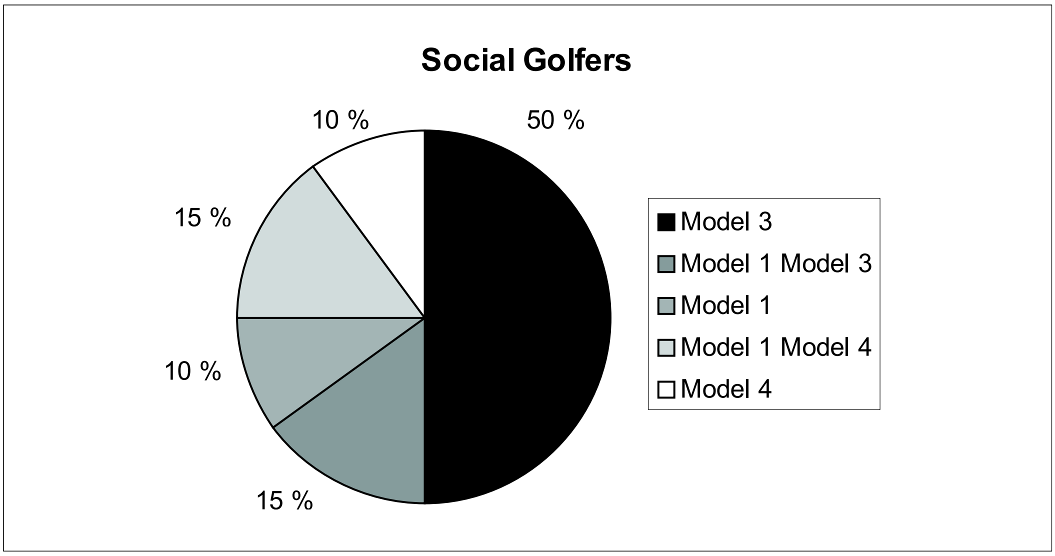

Figure 8.2: Summary of the results in Figure 8.1. The instances are grouped by the model(s) solving them with the least search effort.

10% of the instances. We observe that 15% are best solved by Model 3 and Model 1 with similarities in the amount of pruning and solving times. Model 3 is therefore the leading model for the instances that we have considered.

Posting ordering constraints is not useful for solving some instances of this problem. Model 4 solves 10% of the instances with remarkable gains in the amount of search effort required over the other models. Also, for 15% of the instances, we observe modest differences between Model 1 and Model 4, which are superior to all the remaining models.

In summary, with the static labelling heuristic that fills T a week at a time assigning each golfer in turn, imposing ordering constraints gives significantly smaller search trees and shorter run-times when solving 75% of the considered instances of the social golfers problem. Due to the problem constraints, we cannot utilise multiset ordering constraints to break the symmetry. We therefore impose only strict lexicographic (resp. anti-lexicographic) ordering constraints on the slices of T representing the weeks and on the slices describing the golfers, as well as on the columns of each slice representing a week. The best results are obtained by imposing strict anti-lexicographic ordering constraints.

8.3 Balanced Incomplete Block Design Problem

This was introduced in Chapter 3.2.1. One way of modelling the problem, as shown in Figure 3.1, is to employ a 2-d 0/1 matrix X of $\mathcal{B} \times \mathcal{V}$ to determine the elements of each subset. Since the elements as well as the subsets containing the elements are indistinguishable, the matrix X has row and column symmetry.

As each row is constrained to have r 1s and each column k 1s, a simple way to reduce symmetry is to set the first row and column. For instance, consider the BIBD instance $\langle 7, 7, 3, 3, 1\rangle$. We can set the first row by instantiating its first 3 variables by 1 and its remaining variables by 0. As the first variable of the first column is now 1, we can as well set the first column by assigning 1 to its first 3 variables and 0 to its remaining variables.

Setting the first row and column help reduce some of the row and column symmetries.

Instance #	v, b, r, k, λ	Instance #	v, b, r, k, λ
1	$6, 80, 40, 3, 16$	11	$15, 70, 14, 3, 2$
2	$7, 84, 36, 3, 12$	12	$12, 88, 22, 3, 4$
3	$7, 91, 39, 3, 13$	13	$9, 120, 40, 3, 10$
4	$9, 72, 24, 3, 6$	14	$19, 57, 9, 3, 1$
5	$13, 52, 12, 3, 2$	15	$10, 120, 36, 3, 8$
6	$9, 84, 28, 3, 7$	16	$11, 110, 30, 3, 6$
7	$9, 96, 32, 3, 8$	17	$16, 80, 15, 3, 2$
8	$10, 90, 27, 3, 6$	18	$13, 104, 24, 3, 4$
9	$9, 108, 36, 3, 9$	19	$15, 21, 7, 5, 2$
10	$13, 78, 18, 3, 3$	20	$22, 22, 7, 7, 2$
		21	$16, 32, 12, 6, 4$

Table 8.2: The instances used in the BIBD problem.

However, we can still freely permute any two rows and/or columns that start with the same value. We can therefore impose ordering constraints on the rows and columns to break more symmetry. Due to the constraints on the rows, no pair of rows can be equal unless $r = \lambda$. To break the row symmetry, we enforce that the rows $\vec{R_0}, \vec{R_1}, \ldots, \vec{R}_{v-1}$ corresponding to the v elements are strict anti-lexicographically ordered: $\vec{R_0} >_{lex} \vec{R_1} \ldots >_{lex} \vec{R}_{v-1}$. As for the column symmetry, we enforce that the columns $\vec{C_0}, \vec{C_1}, \ldots, \vec{C}_{b-1}$ corresponding to the b subsets of $\mathcal{V}$ are anti-lexicographically ordered: $\vec{C_0} \geq_{lex} \vec{C_1} \ldots \geq_{lex} \vec{C}_{b-1}$. We refer to the model in which we have $>_{lex}$R $\geq_{lex}$C as well as the first row and column are set as Model 1, and to the model in which no ordering constraints are imposed but only the first row and column are set as Model 2. Note that setting the first row and column in descending order does not conflict with the anti-lexicographic ordering constraints.

Each row of X is constrained to have r $1s$. Therefore, in any solution to the problem, the rows, when viewed as multisets, are equal. Similarly, the columns, when viewed as multisets, are equal in any solution to the problem due to the constraints restricting each column to contain k 1s. Consequently, we cannot make use of multiset ordering constraints to break the row and column symmetries in this model of the problem.

In our experiments, we select some large instances (see Table 8.2) from [CD96]. We adopt a static variable and value ordering and search for a solution by instantiating the matrix X along its rows from top to bottom and exploring the domain of each variable in ascending order. As our experiments have revealed that $>_{lex}$R $\geq_{lex}$C is always superior to $<_{lex}$R $\leq_{lex}$C with this labelling heuristic, we compare only Model 1 and Model 2.

The results of the experiments are shown in Figure 8.3. Instantiating the matrix along its rows from top to bottom works extremely well with anti-lexicographic ordering constraints on the rows and columns[1]. Many instances are solved by Model 1 within a few seconds. On the other hand, Model 2 can solve only 4 instances within an hour time limit. In Figure 8.3, we observe a substantial gain in the amount of pruning and efficiency by solving the problem using Model 1 in preference to Model 2 with two exceptions. The instances $\langle 7, 84, 36, 3, 12\rangle$ and $\langle 7, 91, 39, 3, 13\rangle$ are solved backtrack-free by Model 2 relatively quicker than Model 1. Due to the logarithmic scale in the graphs, we do not report any result of fails for these instances for Model 2.

In summary, with the static labelling heuristic that instantiates X along its rows from top to bottom, imposing strict anti-lexicographic ordering constraints on the rows and anti-lexicographic ordering constraints on the columns significantly reduces the search

[1] This was also pointed out by Jean-François Puget in [Pug02a].

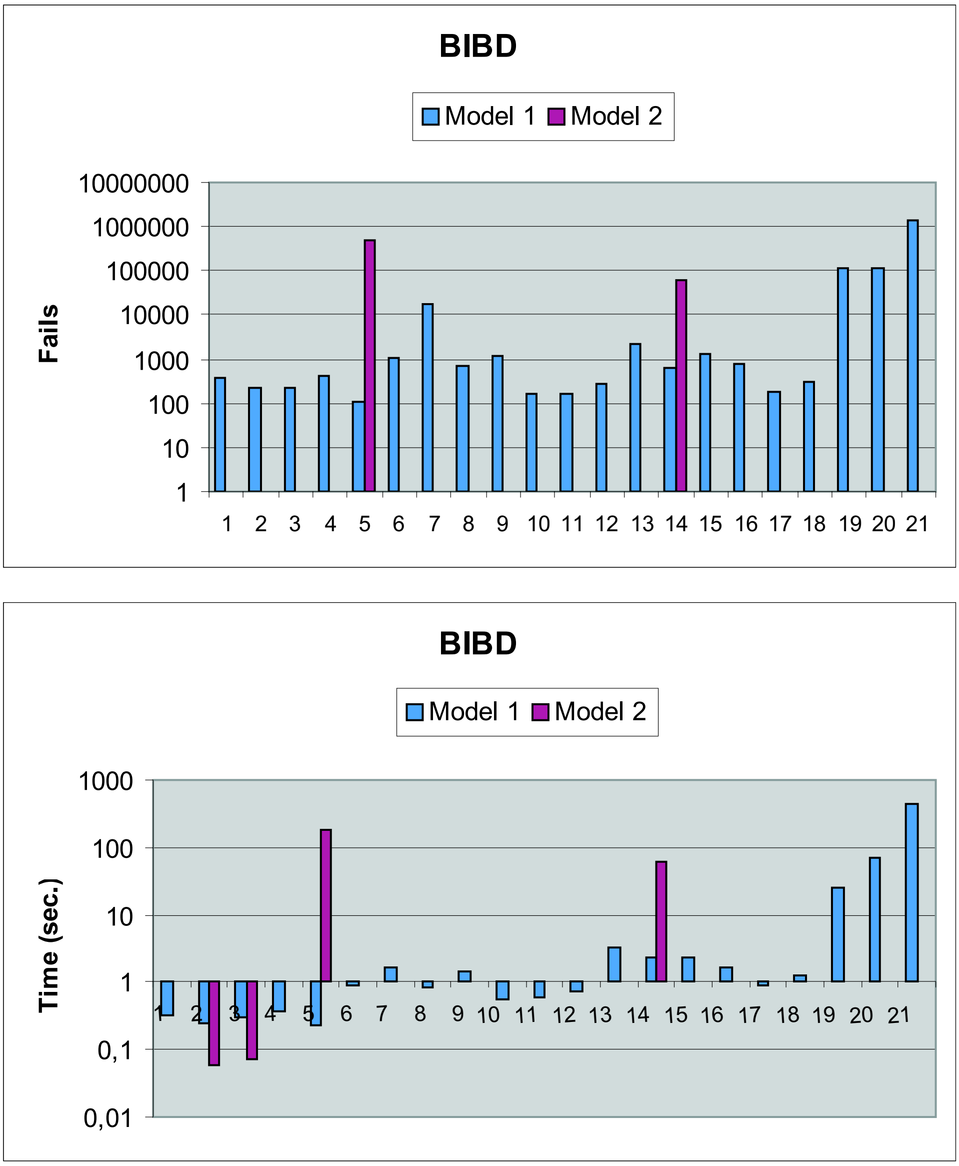

Figure 8.3: BIBD with row-wise labelling of X.

effort required to solve 90.5% of the considered instances of the BIBD problem. Many instances are solved within a few seconds, which cannot otherwise be solved within an hour limit. Due to the problem constraints, we cannot make use of multiset ordering constraints to break the row and column symmetries of X. Our initial experiments have shown that $<_{lex}$R $\leq_{lex}$C is inferior to $>_{lex}$R $\geq_{lex}$C for the instances we have considered.

8.4 Progressive Party Problem

This was introduced in Chapter 3.2.5. In Figure 3.10, one way of modelling the problem is given. In this model, a 2-d matrix H of $\mathcal{P}eriods \times \mathcal{G}uests$, taking values from $\mathcal{H}osts$, is used to represent the assignment of hosts to guests in time periods. The time periods as well as the guests with equal crew size are indistinguishable. Hence, this model of the problem has partial row symmetry between the indistinguishable guests of H, and column symmetry.

Model	Problem					
	$5, 13, 29$			$6, 13, 29$		
	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
No symmetry breaking	180,738	180,860	75.9	-	-	-
$<_{lex}$RC	2,720	2,842	2.7	-	-	-
$\leq_m$RC	-	-	-	-	-	-
$\leq_m$R $\geq_m$C	9,207	9,329	8.0	-	-	-
$\leq_m$R $<_{lex}$C	10,853	10,977	8.6	-	-	-
$\leq_m$R $>_{lex}$C	2,289	2,405	2.6	-	-	-
$<_{lex}$R $\leq_m$C	**2,016**	**2,137**	**2.0**	-	-	-
$<_{lex}$R $\geq_m$C	-	-	-	-	-	-

Table 8.3: Progressive party problem with row-wise labelling of H.

In any solution to the problem, the rows and/or columns are not necessarily equal when viewed as multisets. To break the row and column symmetries, we can therefore utilise both lexicographic ordering and multiset ordering constraints, as well as combine lexicographic ordering constraints in one dimension of the matrix with multiset ordering constraints in the other. Due to the problem constraints, no pair of rows/columns can have equal assignments, but they can be equal when viewed as multisets. This gives us the models $<_{lex}$RC, $\leq_m$RC, $\leq_m$R $\geq_m$C, $\leq_m$R $<_{lex}$C, $\leq_m$R $>_{lex}$C, $<_{lex}$R $\leq_m$C, and $<_{lex}$R $\geq_m$C. As the matrix H has partial row symmetry, the ordering constraints on the rows are posted on only the symmetric rows. The ordering constraints on the columns are, however, posted on all the columns.

In our experiments, we compare the models described above in contrast to the initial model of the problem in which no symmetry breaking ordering constraints are imposed. We consider the original instance of the progressive party problem, which was mentioned in Chapter 3.2.5, with 5 and 6 time periods, referred to as $\langle 5, 13, 29\rangle$ and $\langle 6, 13, 29\rangle$. As in [SBHW96], we give priority to the largest crews, so the guest boats are ordered in descending order of their size. Also, when assigning a host to a guest, we try a value first which is most likely to succeed. We therefore order the host boats in descending order of their spare capacity. We adopt two static variable orderings, and instantiate H either along its rows from top to bottom, or along its columns from left to right.

The results of the experiments are shown in Tables 8.3 and 8.4. With row-wise labelling of H, we cannot solve $\langle 6, 13, 29\rangle$ with or without the symmetry breaking ordering constraints. As for $\langle 5, 13, 29\rangle$, whilst many of the models we have considered give significantly smaller search trees and shorter run-times, $\leq_m$RC and $<_{lex}$R $\geq_m$C cannot return an answer within an hour time limit. The smallest search tree and also the shortest solving time is obtained by $<_{lex}$R $\leq_m$C, in which case the reduction in the search effort is noteworthy compared to the model in which no ordering constrains are imposed. This supports our conjecture that lexicographic ordering constraints in one dimension of a matrix combined with multiset ordering constraints in the other can break more symmetry than lexicographic ordering or multiset ordering constraints on both dimensions. In our experiments, we have also tested the dual models $>_{lex}$RC, $\geq_m$RC, $\geq_m$R $\leq_m$C, $\geq_m$R $>_{lex}$C, $\geq_m$R $<_{lex}$C, $>_{lex}$R $\geq_m$C, and $>_{lex}$R $\leq_m$C with row-wise labelling of H. However, none of these models return an answer within an hour time limit.

Model	Problem					
	$5, 13, 29$			$6, 13, 29$		
	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
No symmetry breaking	20,546	20,676	9.0	20,722	20,871	12.3
$<_{lex}$RC	20,546	20,676	9.0	20,722	20,871	12.4
$\leq_m$RC	-	-	-	-	-	-
$\leq_m$R $\geq_m$C	-	-	-	-	-	-
$\leq_m$R $<_{lex}$C	-	-	-	-	-	-
$\leq_m$R $>_{lex}$C	-	-	-	-	-	-
$<_{lex}$R $\leq_m$C	**7,038**	**7,168**	**3.4**	**7,053**	**7,202**	**4.6**
$<_{lex}$R $\geq_m$C	-	-	-	-	-	-

Table 8.4: Progressive party problem with column-wise labelling of H.

With column-wise labelling of H, we are able to solve $\langle 6, 13, 29\rangle$. Surprisingly, $<_{lex}$RC does the same inference as the model with no ordering constraints, giving the same search tree and solving time for both instances. The only model which reduces the search effort is $<_{lex}$R $\leq_m$C, with significant decreases in the number of the fails as well as the solving times. We once again witness the superiority of combining lexicographic ordering constraints in one dimension with multiset ordering constraints in the other over lexicographic ordering or multiset ordering constraints on both dimensions. With this labelling heuristic, we have also experimented with the dual models $>_{lex}$RC, $\geq_m$RC, $\geq_m$R $\leq_m$C, $\geq_m$R $>_{lex}$C, $\geq_m$R $<_{lex}$C, $>_{lex}$R $\geq_m$C, and $>_{lex}$R $\leq_m$C. For both instances, we have obtained an answer in one hour only by $>_{lex}$R $\leq_m$C. This model solves $\langle 5, 13, 29\rangle$ with 341 fails and in 2 seconds, which is the best result we have for this instance. However, $>_{lex}$R $\leq_m$C solves $\langle 6, 13, 29\rangle$ with 17,803 fails and in 8.2 seconds, and thus $<_{lex}$R $\leq_m$C remain the most effective model for $\langle 6, 13, 29\rangle$.

In summary, enforcing ordering constraints on the rows and columns of H leads to noteworthy reductions in the effort required to solve the $\langle 5, 13, 29\rangle$ instance with row-wise labelling of H, and to solve the two considered instances of the progressive party problem with column-wise labelling of H. As the problem constraints do not imply multiset equality between the symmetric rows and columns, there are many alternative ways of posting the ordering constraints. The best results are obtained by $<_{lex}$R $\leq_m$C and $>_{lex}$R $\leq_m$C. This shows a clear advantage of combining lexicographic ordering and multiset ordering constraints.

8.5 Rack Configuration Problem

This was introduced in Chapter 3.2.3. One way of modelling the problem is given in Figure 3.5. In this model, a 1-d matrix R of $\mathcal{R}acks$, taking values from $\mathcal{R}ackModels$, is used to represent the assignment of rack models to racks. The number of cards – of a particular card type – plugged into a particular rack is determined by a 2-d matrix C of $\mathcal{C}types \times \mathcal{R}acks$. There are two kinds of symmetry in this model. First, the variables of R are symmetric as the racks are indistinguishable. Second, C has partial row symmetry, because the racks of the same rack model are indistinguishable and therefore their card assignments can be interchanged.

Rack Model	Power	Connectors	Price
1	150	8	150
2	200	16	200

Table 8.5: Rack model specification in the rack configuration problem [ILO02].

Card Type	Power
1	20
2	30
3	40
4	50
5	60

Instance	Demand				
#	Type 1	Type 2	Type 3	Type 4	Type 5
1	6	8	6	2	4
2	8	5	6	4	3
3	8	7	6	6	5
4	10	8	6	2	2
5	10	8	6	4	2
6	10	9	9	8	5

Table 8.6: Card type and demand specifications in the rack configuration problem.

By insisting that the variables representing the given m racks are ordered $R_0 \leq R_1 \leq R_{m-1}$, we break the symmetry between the racks. This, however, leaves the symmetry between the racks of the same rack model. In any solution to the problem, the rows of C that correspond to the racks of the same rack model are not necessarily equal when viewed as multisets. To break the partial row symmetry, we can therefore utilise both lexicographic ordering and multiset ordering constraints. This gives us the models $\leq_{lex}$R and $\leq_m$R, in which the rack variables are ordered and the ordering constraints on the rows are posted conditionally. Given $n = |Ctypes|$ and $m = |Racks|$, we insist in $\leq_{lex}$R that two adjacent rows of C are lexicographically ordered if the corresponding racks are assigned the same rack model:

$$\forall j \in [0, m-1) \,.\, R_j = R_{j+1} \rightarrow \langle C_{0,j}, \ldots, C_{n-1,j} \rangle \leq_{lex} \langle C_{0,j+1}, \ldots, C_{n-1,j+1} \rangle$$

whereas in $\leq_m$R we enforce multiset ordering constraint on the adjacent rows if the corresponding racks are assigned the same rack model:

$$\forall j \in [0, m-1) \,.\, R_j = R_{j+1} \rightarrow \langle C_{0,j}, \ldots, C_{n-1,j} \rangle \leq_{m} \langle C_{0,j+1}, \ldots, C_{n-1,j+1} \rangle$$

In our experiments, we compare $\leq_{lex}$R and $\leq_m$R against the model in which only the symmetry between the racks is tackled. We consider several instances of the rack configuration problem, which are described in Tables 8.5 and 8.6. Whilst we respect the rack model specification given in [ILO02], we increase the number of card types to 5, each characterised by the power it requires, and vary the demand of the card types randomly. As in [ILO02], we search for the optimal solution by exploring the racks in turn. For each $j \in \mathcal{R}acks$, we first instantiate R_j and then label the jth row in C in increasing order of $i \in \mathcal{C}types$ to determine how many cards from each card type are plugged into the rack j.

Inst. #	Model	Fails	Choice points	Time (secs.)
1	No symmetry breaking	13,617, 898	13,617,911	366.5
	$\leq_{lex}$R	**1,938,252**	**1,938,265**	**70.3**
	$\leq_{m}$R	4,689,464	4,689,376	182.0
2	No symmetry breaking	25,115,179	25,115,190	685.0
	$\leq_{lex}$R	**3,373,548**	**3,373,555**	**114.6**
	$\leq_{m}$R	7,845,034	7,845,045	302.0
3	No symmetry breaking	14,956,347	14,956,346	458.4
	$\leq_{lex}$R	**1,004,284**	**1,004,283**	**38.0**
	$\leq_{m}$R	4,367,915	4,367,914	196.4
4	No symmetry breaking	10,675,887	10,675, 900	294.3
	$\leq_{lex}$R	**1,529,197**	**1,529,210**	**55.0**
	$\leq_{m}$R	2,664,774	2,664,787	115.5
5	No symmetry breaking	95,553,241	95,553,250	2458.2
	$\leq_{lex}$R	**10,377,867**	**10,377,876**	**363.0**
	$\leq_{m}$R	28,212,479	28,212,486	1102.7
6	No symmetry breaking	1,377,881	1,377,880	47.0
	$\leq_{lex}$R	**115,309**	**115,308**	**5.2**
	$\leq_{m}$R	692,058	692,057	31.8

Table 8.7: Rack configuration problem with exploring each rack in turn by first instantiating its model and then determining its cards.

The results of the experiments are shown in Table 8.7. Clearly, symmetry breaking with ordering constraints is beneficial for all the instances we have considered. The instances are solved with substantial gains, both in the size of the search trees and run-times, by imposing the ordering constraints. Even though both $\leq_{lex}$R and $\leq_{m}$R greatly reduce the search effort required to solve the instances, $\leq_{lex}$R is superior to $\leq_{m}$R with notable differences in the amount of pruning and solving times. We have also experimented with the dual models $\geq_{lex}$R and $\geq_{m}$R using the same labelling heuristic. However, this has not led to any promising improvement over the results presented in Table 8.7.

In summary, with the static labelling heuristic that explores each rack in turn by first instantiating its model and then determining its cards, imposing ordering constraints gives significantly smaller search trees and shorter run-times when solving all the considered instances of the rack configuration problem. As the problem constraints do not imply multiset equality between the symmetric rows, we can tackle the partial row symmetry by imposing either multiset (resp. anti-multiset) ordering or lexicographic (resp. anti-lexicographic) ordering constraints on the rows. The best results are obtained by $\leq_{lex}$R.

8.6 Generating Hamming Codes

This was introduced in Chapter 3.2.7. One way of modelling the problem, as shown in Figure 3.12, is to employ a 2-d 0/1 matrix X of $\mathcal{B} \times \mathcal{C}odes$ to determine the b-bit codes. Since the codes as well as the positions in the codes are indistinguishable, the matrix X has both row and column symmetry.

In any solution to the problem, the rows and/or columns are not necessarily equal

Model	Problem					
	$10, 15, 9$			$10, 10, 5$		
	Fails	Choice points	Time (secs.)	Fails	Choice points	Time (secs.)
No symmetry breaking	-	-	-	114,072	114,120	10.0
$<_{lex}$R $\leq_{lex}$C	297,421	297,420	30.0	**2,939**	**2,985**	**0.4**
$\leq_{m}$RC	-	-	-	-	-	-
$\leq_{m}$R $\geq_{m}$C	-	-	-	-	-	-
$\leq_{m}$R $\leq_{lex}$C	**78,412**	**78,411**	**7.2**	114,072	114,120	10.0
$\leq_{m}$R $\geq_{lex}$C	169,665	169,664	13.6	548,026	548,057	53.0
$<_{lex}$R $\leq_{m}$C	-	-	-	-	-	-
$<_{lex}$R $\geq_{m}$C	-	-	-	-	-	-

Table 8.8: Generating Hamming codes with row-wise labelling of X.

when viewed as multisets. To break the row and column symmetries, we can therefore utilise both lexicographic ordering and multiset ordering constraints, as well as combine lexicographic ordering constraints in one dimension of the matrix with multiset ordering constraints in the other. Due to the constraints on the rows, no pair of rows can have equal assignments, but they can be equal when viewed as multisets. This gives us the models $<_{lex}$R $\leq_{lex}$C, $\leq_{m}$RC, $\leq_{m}$R $\geq_{m}$C, $\leq_{m}$R $\leq_{lex}$C, $\leq_{m}$R $\geq_{lex}$C, $<_{lex}$R $\leq_{m}$C, and $<_{lex}$R $\geq_{m}$C. As X is a 0/1 matrix, we enforce multiset ordering constraints on the rows and/or columns by insisting that the sums of the rows and/or columns are ordered.

In our experiments, we consider two instances of the problem, $\langle 10, 15, 9\rangle$ and $\langle 10, 10, 5\rangle$, and compare the models described above in contrast to the initial model of the problem in which no symmetry breaking ordering constraints are imposed. We search for a solution by instantiating the matrix X along its row from top to bottom, exploring the domain of each variable in ascending order.

The results of the experiments are shown in Table 8.8. Whilst many of the models fail to return an answer for $\langle 10, 15, 9\rangle$ in an hour, $<_{lex}$R $\leq_{lex}$C, $\leq_{m}$R $\leq_{lex}$C, and $\leq_{m}$R $\geq_{lex}$C dramatically reduce the size of the search tree, proving unsatisfiability in less than a minute. The smallest search tree and also the shortest solving time is obtained by $\leq_{m}$R $\leq_{lex}$C which is much more effective and efficient than $<_{lex}$R $\leq_{lex}$C and $\leq_{m}$R $\geq_{lex}$C. We once again witness the superiority of combining lexicographic ordering constraints in one dimension of a matrix with multiset ordering constraints in the other over lexicographic ordering or multiset ordering constraints on both dimensions. As for $\langle 10, 10, 5\rangle$, we get an answer in an hour only with $<_{lex}$R $\leq_{lex}$C, $\leq_{m}$R $\leq_{lex}$C, and $\leq_{m}$R $\geq_{lex}$C as in the case of solving the previous instance. The difference is that we can now solve the problem even in the absence of the ordering constraints. Surprisingly, $\leq_{m}$R $\leq_{lex}$C does the same inference as the model with no ordering constraints, giving the same search tree and solving time. Moreover, $\leq_{m}$R $\geq_{lex}$C increases the size of the search tree, making it more difficult to solve the problem. On the other hand, $<_{lex}$R $\leq_{lex}$C significantly reduces the search effort. Hence, $<_{lex}$R $\leq_{lex}$C is the most effective model for this instance of the problem.

We have also experimented with the dual models $>_{lex}$R $\geq_{lex}$C, $\geq_{m}$RC, $\geq_{m}$R $\leq_{m}$C, $\geq_{m}$R $\geq_{lex}$C, $\geq_{m}$R $\leq_{lex}$C, $>_{lex}$R $\geq_{m}$C, and $>_{lex}$R $\leq_{m}$C using the same labelling heuristic. Each model creates the same search tree as its dual model for $\langle 10, 15, 9\rangle$ which has no solution. For $\langle 10, 10, 5\rangle$, which has a solution, $<_{lex}$R $\leq_{lex}$C remain the best model.

size	Model	Fails	Choice points	Time (secs.)
5	No symmetry breaking	95	129	25.4
	$<_{lex}$R	**32**	**66**	**25.1**
	$\leq_m$R	201	225	25.0
10	No symmetry breaking	262	319	25.8
	$<_{lex}$R	**41**	**98**	**25.0**
	$\leq_m$R	525	566	25.3
15	No symmetry breaking	479	560	26.6
	$<_{lex}$R	**68**	**149**	**25.2**
	$\leq_m$R	1,583	1,640	27.1
20	No symmetry breaking	769	879	28.0
	$<_{lex}$R	**100**	**210**	**25.5**
	$\leq_m$R	-	-	-
25	No symmetry breaking	1,083	1,218	29.0
	$<_{lex}$R	**108**	**243**	**28.4**
	$\leq_m$R	-	-	-

Table 8.9: Word design problem with row-wise labelling of M.

In summary, with the static labelling heuristic that instantiates X along its rows from top to bottom, imposing ordering constraints on the rows and columns of X substantially reduces the search effort required to solve the two considered instances of the problem of generating Hamming codes. As the problem constraints do not imply multiset equality between the symmetric rows and columns, there are many alternative ways of posting the ordering constraints. Whilst $<_{lex}$R $\leq_m$C is the most effective model for one instance, it is $<_{lex}$R $\leq_{lex}$C for the other. This confirms the theoretical result on the incomparability of lexicographic ordering constraints on both dimensions of a matrix with lexicographic ordering constraints in one dimension and multiset ordering constraints in the other.

8.7 Word Design Problem

This was introduced in Chapter 3.2.6. One way of modelling the problem is given in Figure 3.11. In this model, a 2-d matrix M of $\mathcal{L} \times \mathcal{W}ords$, taking values from $\{0, 1, 2, 3\}$, is used to represent the symbols at each position of each word. This matrix has row symmetry as the order of the words in the set are insignificant.

As seen in Figure 3.11, the words are not necessarily equal when viewed as multisets. Even though no pair of words can be identical, they can be equal when viewed as multisets. We can thus post either strict lexicographic ordering constraints or multiset ordering constraints on the rows to break the row symmetry. In our experiments, we compare $<_{lex}$R and $\leq_m$R against the initial model without any symmetry breaking ordering constraints by increasing the size of the set of words. As for the labelling heuristic, we label M row-wise from top to bottom, exploring the domain of each variable in ascending order.

The results of our first experiments are shown in Table 8.9. We observe that $\leq_{lex}$R decreases the size of the search tree in each case. The gain in run-times are apparent for generating larger sets of words which are not shown in Table 8.9. Surprisingly, $\leq_m$R is not an effective model, as it always leads to larger search trees. This suggests that the multiset ordering constraints on the rows conflict with the labelling heuristic.

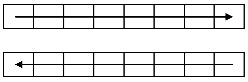

Figure 8.4: Row-snake labelling.

size	Model	Fails	Choice points	Time (secs.)
5	No symmetry breaking	33	65	25.1
	$<_{lex}$R	**5**	**35**	**25.0**
	$\leq_m$R	45	73	25.8
10	No symmetry breaking	188	252	25.6
	$<_{lex}$R	**22**	**71**	**25.0**
	$\leq_m$R	338	384	26.0
15	No symmetry breaking	**367**	**457**	**29.0**
	$<_{lex}$R	-	-	-
	$\leq_m$R	594	656	28.8
20	No symmetry breaking	**530**	**642**	**27.5**
	$<_{lex}$R	-	-	-
	$\leq_m$R	257,760	257,838	713.0
25	No symmetry breaking	**739**	**875**	**28.2**
	$<_{lex}$R	-	-	-
	$\leq_m$R	-	-	-

Table 8.10: Word design problem with row-snake labelling of M.

The reason for $\leq_m$R to give more failures compared to the other models is that by instantiating the matrix row-wise with or without the lexicographic ordering constraints on the rows, we get solutions where the rows that are equal when viewed as multisets are not necessarily consecutive. For instance, for $size = 5$ we get $\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 \end{pmatrix}$. With the multiset ordering constraints, however, whenever the second row is set to $\langle 0,0,0,0,3,3,3,3\rangle$, the rest of the rows can be neither $\langle 0,0,1,1,0,0,1,1\rangle$, nor $\langle 0,0,1,1,1,1,0,0\rangle$, nor $\langle 0,0,1,1,2,2,3,3\rangle$. For instance, for $size = 5$, $\leq_m$R finds $\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 3 & 3 & 3 & 3 \\ 0 & 0 & 3 & 3 & 0 & 0 & 3 & 3 \\ 0 & 0 & 3 & 3 & 3 & 3 & 0 & 0 \\ 0 & 2 & 2 & 3 & 2 & 3 & 3 & 3 \end{pmatrix}$ which shows that many instantiations that contain $0, 1, 2$ but not 3 are rejected for the third row, even if they satisfy the problem constraints. Hence, the number of failures are increased.

This has suggested seeking a labelling heuristic which forces the rows that are equal when viewed as multisets to be consecutive. One heuristic is to label the rows in alternating directions. We call this "row-snake labelling" and depict it in Figure 8.4. Row-snake labelling forces two multisets to be consecutive, and is a better labelling heuristic than the previous one. The model without any ordering constraints now explores smaller search trees. On the other hand, the lexicographic ordering constraints on the rows now conflict with the row-snake labelling. The last three instances cannot be solved by $\leq_{lex}$R in an hour. As expected, the results of $\leq_m$R improve. For instance, $\leq_m$R is now able to solve $size = 20$ within the time limit. However, the number of failures remain increased compared to the model with no ordering constraints. See Table 8.10.

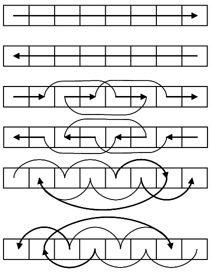

Figure 8.5: Multi-directional row labelling.

size	Model	Fails	Choice points	Time (secs.)
5	No symmetry breaking	**0**	**40**	**25.0**
	$<_{lex}$R	**0**	**38**	**25.0**
	$\leq_m$R	**0**	**40**	**25.0**
10	No symmetry breaking	**0**	**64**	**24.8**
	$<_{lex}$R	2	64	24.7
	$\leq_m$R	**0**	**64**	**24.8**
15	No symmetry breaking	**87**	**179**	**25.3**
	$<_{lex}$R	4,029	4,100	60.4
	$\leq_m$R	367	451	25.5
20	No symmetry breaking	**228**	**350**	**26.0**
	$<_{lex}$R	-	-	-
	$\leq_m$R	1,656	1,755	28.0
25	No symmetry breaking	**364**	**508**	**26.6**
	$<_{lex}$R	-	-	-
	$\leq_m$R	2,790	2,899	30.0

Table 8.11: Word design problem with multi-directional row labelling of M.

As labelling the matrix along its rows in alternating directions seem to be a good heuristic for the word design problem, we have devised another heuristic which explores the rows in 6 possible directions. This multi-directional row labelling forces 6 multisets to be consecutive and is depicted in Figure 8.5. This heuristic works notably well for the model without any ordering constraints. The search trees are now even smaller. In particular, we obtain solutions for $size = 5$ and $size = 10$ backtrack-free. Also, the results of $\leq_m$R improve further. For instance, $\leq_m$R is now able to solve $size = 25$ in less than a minute. Unfortunately, with the multi-directional row labelling, neither $\leq_{lex}$R nor $\leq_m$R are beneficial for solving the problem effectively. Imposing lexicographic ordering or multiset ordering constraints on the rows enlarges the search tree. See Table 8.11.

We have also experimented with the dual models $>_{lex}$R and $\geq_m R$ using the same strategies described above. However, this has not led to any significant improvements over the results presented in Tables 8.9, 8.10, and 8.11.

In summary, an effective way of generating large sets of words is to label M along its rows in multiple directions. As the problem constraints do not imply multiset equality between the rows, we can break the row symmetry by posting either lexicographic ordering or multiset ordering constraints on the rows. However, either of them clashes with the labelling heuristic, and therefore we do not obtain smaller search trees by symmetry breaking with ordering constraints in the word design problem.

8.8 Summary

In this chapter, we have experimented with the ordering constraints proposed in Chapter 4 to judge whether they can often be utilised for and are effective in breaking row and column symmetries, and whether the ordering constraints which are incomparable in theory are incomparable also in practice. The experiments were performed using 6 problems, each originating from a different application domain. The results support our three-fold hypothesis:

1. We can utilise (anti-)lexicographic ordering constraints to break symmetry in all the problems we have considered. Only in two problems is multiset ordering already implied by the problem constraints. Hence, (anti-)multiset ordering constraints are useful for breaking symmetry in the majority of the problems. This supports our hypothesis that **the ordering constraints are often not redundant and bring additional pruning.**

2. Symmetry breaking with ordering constraints has proven very beneficial for solving the first 5 problems we have considered. In the presence of the ordering constraints, the problems are solved much quicker as the search trees explored are significantly smaller. In particular, some instances of the social golfers problem, many instances of the balanced incomplete block design problem, and the first instance of the problem of generating Hamming codes can be solved within an hour time limit only if some ordering constraints are imposed. This supports our hypothesis that **the ordering constraints are effective in breaking row and column symmetries as they significantly reduce the size of the search tree and the time to solve the problem.** On the other hand, symmetry breaking with ordering constraints has negative effects on solving the word design problem effectively. The search trees get larger, making it more difficult to solve the problem, in the presence of the ordering constraints. This is because the labelling heuristic which is powerful for generating large sets of words conflict with the ordering constraints.

3. Even though lexicographic ordering constraints can be utilised for a wider range of problems, the most effective symmetry breaking constraints are sometimes obtained by combining lexicographic ordering constraints in one dimension of a matrix with multiset ordering constraints in the other. In our experiments, we see evidences of this in the progressive party problem and in the problem of generating Hamming codes. Our theoretical results on the incomparability of lexicographic ordering constraints, multiset ordering constraints, and their combinations are confirmed by the results obtained from the progressive party problem, the rack configuration problem, the problem of generating Hamming codes, and the word design problem. Whilst $\leq_{lex}$R dominates $\leq_{m}$R in the rack configuration problem, $\leq_{m}$R is superior to $\leq_{lex}$R in the word design problem using the multi-directional row labelling which is useful for generating large sets of words effectively. Whilst $<_{lex}$R $\leq_{m}$C is the most effective model for one instance of the problem of generating Hamming codes, $<_{lex}$R $\leq_{lex}$C is the best for another instance. This supports our hypothesis that **the ordering constraints that are incomparable in theory are incomparable also in practice.**

We learn from our experiments with the social golfers problem and the balanced incomplete block design problem that imposing anti-lexicographic ordering constraints, as opposed to the lexicographic ordering constraints, could give much more effective models and therefore should seriously be considered when symmetry breaking with lexicographic ordering constraints. Another lesson we learn from our experiments with generating Hamming codes is that multiset ordering constraints in one dimension of a matrix combined with lexicographic ordering constraints in the other can be the best way of symmetry breaking even if we have only 0/1 variables.

Chapter 9

Conclusions and Future Work

In this dissertation, we have shown that row and column symmetry is a common type of symmetry in constraint programming. We have argued that symmetry breaking methods like SES, SBDS, and SBDD have difficulty in dealing with the super-exponential number of symmetries in a problem with row and column symmetry. We have therefore proposed some ordering constraints which can effectively break such symmetries. To use these constraints in practice, we have developed some optimal linear time propagators. We have demonstrated the effectiveness of the ordering constraints on a wide range of problems.

This final chapter brings the dissertation to a conclusion and is organised as follows. In Section 9.1, we present the contributions of this dissertation by relating our achievements to our goals and by discussing the results in the context of the thesis defended:

> *Row and column symmetry is a common type of symmetry in constraint programming. Ordering constraints can effectively break this symmetry. Efficient global constraints can be designed for propagating such ordering constraints.*

The general lessons learnt are discussed in Section 9.2. We point out the limitations of our work in Section 9.3 and present our plans for future work in Section 9.4. Finally, we conclude in Section 9.5.

9.1 Contributions

We have observed in Chapter 3 that one common pattern in constraint programs is a matrix model. A wide range of problems originating from diverse application areas including combinatorics, design, configuration, scheduling, timetabling, bioinformatics, code generation can be effectively represented and efficiently solved using a matrix model. In Chapter 3, we have also identified that there are at least two patterns that arise commonly within matrix models: row and column symmetry, and value symmetry. Of the 12 matrix models we have studied, each has at least one matrix with (partial) row and/or (partial) column symmetry, and 4 of them have (partial) value symmetry.

Symmetry in constraint programs can significantly slow down the search process, as it generates symmetrically equivalent states in the search space. Some symmetry breaking methods have been devised in the past years, such as SES [BW99][BW02], SBDS [GS00], and SBDD [FM01][FSS01], all of which can directly be used to break all row and column symmetries of a matrix. However, they may not be a good way of dealing with row and column symmetry for the following reasons. An $n \times m$ matrix with row and column symmetry has $n!m!$ symmetries, which increase super-exponentially. SES and SBDS treat each symmetry individually, which is impractical when the number of symmetries is large. Similarly, the dominance checks of SBDD can be very expensive in the presence of many symmetries.

Row and column symmetry is an important class of symmetries, as it is very common and existing methods have difficulty in dealing with the super-exponential number of symmetries in a problem with row and column symmetry. We have shown that value symmetry in a matrix can easily be transformed to, for instance, row symmetry. The main objective of the work presented in this dissertation is therefore to break row and column symmetries of a matrix effectively. To achieve our objective, the approach adopted in this research is to attempt to impose ordering constraints on the rows and columns of a matrix. Consequently, our goals were:

1. to investigate the types of ordering constraints that can be posted on a matrix to break row and column symmetries;

2. to devise global constraints to post and propagate the ordering constraints effectively and efficiently;

3. to show the effectiveness of the ordering constraints in breaking row and column symmetries.

In Chapter 4, we have considered which ordering constraints can be utilised to break the row and column symmetries of a matrix effectively. We have shown that we can insist that the rows and columns are lexicographically ordered. Alternatively, we can view each row and column as a multiset and impose that the rows and columns are multiset ordered. By constraining the rows (resp. columns) to be multiset ordered, we do not distinguish the columns (resp. rows). We can therefore combine multiset ordering constraints in one dimension with lexicographic ordering constraints in the other. Lexicographic ordering and multiset ordering are incomparable. As a result, we have shown that imposing lexicographic ordering constraints on the rows and columns, imposing multiset ordering constraints on the rows and columns, and imposing one ordering in one dimension and the other ordering in the other are all incomparable. We have theoretically studied the effectiveness of the ordering constraints in breaking row and column symmetries. We have shown that in theory the ordering constraints may not eliminate all symmetries and argued that it is difficult to assess theoretically whether we can significantly reduce the search effort by imposing ordering constraints. We have extended our results to cope with symmetries in any number of dimensions, with partial symmetries, and with symmetric values. Finally, we have identified special and useful cases where all row and column symmetries can be broken by posting only a linear number of constraints. As a result, our first goal has been achieved.

Our next concern was to devise global constraints to post and propagate the ordering constraints effectively and efficiently. As we can enforce the rows (resp. columns) to be in lexicographic or multiset order by imposing the ordering between the adjacent or all pairs of row (resp. column) vectors, we have focused on propagating the ordering constraint posted on a pair of vectors.

In Chapter 5, we have considered propagating the lexicographic ordering constraint enforced on a pair of vectors. We have shown that there are at least two ways of decomposing this constraint, both of which are inferior to maintaining GAC on the constraint. Combining the decompositions is equivalent to maintaining GAC; however, this carries a penalty in the cost of constraint propagation. Alternatively, by using the domain size of the variables in the vectors, we can pose arithmetic inequality constraints to ensure lexicographic ordering. This approach is feasible only if the vectors and the domain sizes are not too large. We have therefore developed an optimal linear time algorithm which maintains GAC on the lexicographic ordering constraint. We have extended the algorithm for the strict lexicographic ordering constraint, entailment, vectors of any length, and vectors whose variables are repeated and shared. We have studied the propagation of vector equality and disequality constraints using lexicographic ordering constraints, and demonstrated that decomposing a chain of lexicographic ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. Finally, we have provided experimental evidence of the effectiveness and the efficiency of the algorithm.

We have identified in Chapter 6 that another common pattern in constraint programs is the lexicographic ordering constraint on a pair of vectors of 0/1 variables together with a sum constraint on each vector. This pattern frequently occurs in problems involving demand, capacity or partitioning that are modelled using 0/1 matrices with row and/or column symmetry. We have argued that propagating the lexicographic ordering constraint by taking into account the sum constraints can lead to more pruning than the total pruning obtained by propagating the lexicographic ordering constraint and the sum constraints independently. This increases the effectiveness of the propagation of the lexicographic ordering constraint. We have therefore introduced a new global constraint on 0/1 variables that combines together the lexicographic ordering constraint with two sum constraints. An alternative way of propagating this combination of constraints is using the algorithm by Bessière and Régin for enforcing GAC on an arbitrary conjunction of constraints [BR98]. By exploiting the semantics of the constraints in this new global constraint, we have proposed an optimal linear time algorithm which maintains GAC. We have extended the algorithm for combining the strict lexicographic ordering constraint with sums, for entailment, and sums that are not ground but bounded. We have shown that decomposing a chain of these ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. Finally, we have demonstrated experimentally that the algorithm is most useful when there is a very large space to explore, such as when the problem is unsatisfiable, or when the labelling heuristics are poor or conflict with the lexicographic ordering constraints.

In Chapter 7, we have considered propagating the multiset ordering constraint enforced on a pair of vectors. We have shown that there are at least two ways of decomposing this constraint, both of which are inferior to maintaining GAC on the constraint. Alternatively, by assigning a weight to each value in the domain of the variables, one can pose arithmetic inequality constraints to ensure multiset ordering. However, this approach is not feasible when the vectors and domain sizes are large, which is often the case. We have therefore developed an efficient linear time algorithm which maintains GAC on the multiset ordering constraint. The algorithm works also when the vectors are of different length. We have extended the algorithm for the strict multiset ordering constraint and entailment. We have proposed an alternative filtering algorithm whose complexity is $O(n\ log(n))$ where n is the length of the vectors and which is useful when the domains are large. We have studied the propagation of multiset equality and disequality constraints using multiset ordering constraints, and demonstrated that decomposing a chain of multiset ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. Finally, we have provided experimental evidence of the effectiveness and the efficiency of the algorithm.

As a result of the work carried out in Chapters 5, 6, and 7, our second goal has been achieved.

In Chapter 8, we have performed a wide range of experiments to show the effectiveness of the ordering constraints in breaking row and column symmetries. We have observed that the ordering constraints are often not redundant and bring additional pruning. Moreover, our theoretical results on the incomparability of lexicographic ordering constraints, multiset ordering constraints, and their combinations are confirmed by our experimental results. Furthermore, for the majority of the problems considered, the ordering constraints are very effective in breaking row and column symmetries, significantly reducing the size of the search tree and the time to solve the problem. Consequently, our third goal has been achieved.

This has all provided convincing evidence in support of our thesis:

> *Row and column symmetry is a common type of symmetry in constraint programming. Ordering constraints can effectively break this symmetry. Efficient global constraints can be designed for propagating such ordering constraints.*

9.2 Lessons Learnt

In achieving our goals, we have learnt some general lessons about symmetry breaking with ordering constraints, propagating constraints, and combining constraints. In the following, we discuss our experiences which can provide insights for future research.

9.2.1 Symmetry Breaking with Ordering Constraints

When breaking symmetry using ordering constraints, we may have the choice of imposing alternative orderings each with a different property. For instance, we can utilise both lexicographic ordering and multiset ordering constraints to break row and column symmetry. Lexicographic ordering is a total ordering, but multiset ordering is only a preordering. An immediate question is which type of ordering constraints are preferable. Also, does posting more constraints mean obtaining smaller search trees and shorter run-times?

Total vs Non-total Ordering Constraints

Whilst imposing a total ordering like lexicographic ordering on the rows of a matrix with row symmetry breaks all symmetries, imposing a non-total ordering like multiset ordering may leave some symmetries. This may encourage us to consider only total orderings when symmetry breaking with ordering constraints. On the other hand, enforcing multiset ordering constraints may give smaller search trees and shorter run-times than enforcing lexicographic ordering constraints when the labelling heuristic conflicts with the lexicographic ordering constraints, or pushes the search towards solutions satisfying the lexicographic ordering constraints. Hence, symmetry breaking by imposing an ordering which is not necessarily total can sometimes be more effective in practise than by imposing a total ordering.

The most effective constraints for breaking row and column symmetry are sometimes obtained by combining lexicographic ordering constraints in one dimension of a matrix with multiset ordering constraints in the other. Such a combination can be useful even if we have 0/1 variables, in which case multiset ordering reduces to sum ordering. This suggests that we should consider benefiting from the complementary strengths of different orderings when symmetry breaking with ordering constraints.

The Number of Ordering Constraints

The more restricted ordering we impose the more symmetries we can break. For instance, in an equivalence class of assignments to a matrix, the set of assignments where the rows and columns are multiset ordered is a subset of the set of assignments where the rows are multiset ordered. On the other hand, enforcing the rows and columns to be multiset ordered may give larger search trees and longer run-times than enforcing only the rows to be multiset ordered when the labelling heuristic pushes the search towards solutions where only the rows are multiset ordered. Consequently, symmetry breaking by imposing fewer ordering constraints can sometimes be more effective in practice than by imposing more ordering constraints.

9.2.2 Propagating Constraints

A global constraint can be propagated by either using a specialised algorithm or decomposing it into simpler constraints. Which method is preferable depends on a number of factors. If decomposing is not a good idea, then we need to worry about developing an efficient algorithm. By learning from the successful algorithms of related constraints, we can partially automate this process. However, we still need to know the semantics of the constraint. Also, implementation issues have considerable impacts on the efficiency of the algorithm.

Algorithm vs Decomposition

One way of propagating a global constraint is to decompose it into simpler constraints. The total pruning obtained by the propagation of each simpler constraint is likely to be weaker than maintaining GAC on the original constraint, as the global view of the constraint is lost in the decomposition. In such a case, an algorithm which maintains GAC is a more effective way of propagating the constraint. However, posting different decompositions of a constraint simultaneously can give GAC. For instance, there are at least two ways of decomposing the lexicographic ordering constraint. We have shown that these decompositions are incomparable and that an algorithm which maintains GAC does more pruning than each of the decompositions. On the other hand, we have also proven that the two decompositions together maintain GAC, behaving similarly to the algorithm. Therefore, possible decompositions of a constraint need to be analysed before attempting to develop a specialised algorithm.

We have shown that one way of decomposing the multiset ordering constraint posted on a pair of vectors is to insist that the occurrence vectors associated with the original vectors are lexicographically ordered. Such occurrence vectors can be constructed via an extended global cardinality constraint which prunes values also from the occurrence vectors. The global cardinality constraint is available in many CP toolkits, but the associated algorithms do not (always) prune values from the occurrence vectors. As we have witnessed, this decomposition may not be correct using the cardinality constraints currently available. Hence, when decomposing a constraint, it is important to know which algorithms the CP system uses to propagate the simpler constraints. Decomposing a constraint can lead to incorrect results if the filtering algorithm of a constraint in the decomposition does not behave in the way wanted.

GAC Algorithms for Ordering Constraints

The GAC algorithms of the ordering constraints we have considered have a lot in common, and this can be exploited when designing a GAC algorithm for another ordering constraint. Assume that we want to design an algorithm to maintain GAC on a constraint which combines an ordering constraint $\vec{X} \preceq \vec{Y}$, with a constraint C_1 on $\vec{X}$ and a constraint C_2 on $\vec{Y}$. For instance, $\preceq$ is $\leq_{lex}$, and C_1 and C_2 are *true* for the lexicographic ordering constraint; but $\preceq$ is $\leq_{lex}$, C_1 is $\sum_i X_i = Sx$, and C_2 is $\sum_i Y_i = Sy$ for the lexicographic ordering with sum constraints.

First, when looking for a support for a value v of a variable V, we consider the "best" support by minimising $\vec{X}$ and maximising $\vec{Y}$ when $V \leftarrow v$. That is, for $X_i \leftarrow v$, we check whether $min\{\vec{x} \mid C_1 \ \wedge \ x_i = v \ \wedge \ \vec{x} \in \vec{X}\} \preceq max\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$; and for $Y_i \leftarrow v$, we check whether $min\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\} \preceq max\{\vec{y} \mid C_2 \ \wedge \ y_i = v \ \wedge \ \vec{y} \in \vec{Y}\}$. If the test is *true* then v is supported. If $\vec{X}$ (resp. $\vec{Y}$) gets even smaller (resp. larger) by $X_i \leftarrow w$ (resp. $Y_i \leftarrow w$), then the support of v also supports w due to the transitivity property of the ordering relations. This avoids seeking support for every value in the domains. For instance, the algorithms of the lexicographic ordering and the multiset ordering constraints consider the values in the domain of X_i starting from the largest element. Whenever a value v has a support, any value $w < v$ is also supported, because we have $min\{\vec{x} \mid x_i = w \ \wedge \ \vec{x} \in \vec{X}\} <_{lex} min\{\vec{x} \mid x_i = v \ \wedge \ \vec{x} \in \vec{X}\}$ and $min\{\vec{x} \mid x_i = w \ \wedge \ \vec{x} \in \vec{X}\} <_m min\{\vec{x} \mid x_i = v \ \wedge \ \vec{x} \in \vec{X}\}$.

Second, the prunings of the algorithms tighten $max\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\}$ so that we achieve (1) $max\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\} \preceq max\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$; and similarly the prunings tighten $min\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$ so that we achieve (2) $min\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\} \preceq min\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$. As ordering relations are transitive, the conditions (1) and (2) are sufficient for GAC. Third, the prunings do not require any calls back to the filtering algorithm as they do touch neither $min\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\}$ nor $max\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$ which provide support for the values in the vectors. Finally, by transitivity, the constraint is *false* iff $min\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\} \succ max\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$, and is *true* iff $max\{\vec{x} \mid C_1 \ \wedge \ \vec{x} \in \vec{X}\} \preceq min\{\vec{y} \mid C_2 \ \wedge \ \vec{y} \in \vec{Y}\}$.

This could help automate the process of designing a GAC algorithm for an ordering constraint. However, we still need to know the semantics of the constraint to look for supports quickly without having to reconstruct the minimal and the maximal vectors repeatedly.

Implementing Propagation Algorithms

Implementing a propagation algorithm raises a number of issues which can have significant effects on the efficiency of the algorithm: how do we compute the data structures, and when do we propagate the constraint?

We have observed that if a data structure is easy to restore using its previous value, then it is more efficient to maintain it incrementally than computing it from scratch every time we need it. For instance, the algorithm of the multiset ordering constraint $\vec{X} \leq_m \vec{Y}$ uses a pair of occurrence vectors $\vec{ox}$ and $\vec{oy}$ to propagate the constraint. Whilst ox_i gives the number of occurrences of $max\{\!\{\texttt{floor}(\vec{X})\}\!\} - i$ in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$, oy_i gives the number of occurrences of $max\{\!\{\texttt{ceiling}(\vec{Y})\}\!\} - i$ in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. One option is to reconstruct these vectors at every propagation step. This requires a complete scan of $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$, and thus the cost is $O(n)$. Another option is to keep the occurrence vectors up-to-date as values are pruned from the domains. We do this as follows. When the minimum value in some $\mathcal{D}(X_i)$ changes, we update $\vec{ox}$ by incrementing the entry corresponding to new $min(X_i)$ by 1, and decrementing the entry corresponding to old $min(X_i)$ by 1. Similarly, when the maximum value in some $\mathcal{D}(Y_i)$ changes, we update $\vec{oy}$ by incrementing the entry corresponding to new $max(Y_i)$ by 1, and decrementing the entry corresponding to old $max(Y_i)$ by 1. As an update is a constant time operation, this approach reduces the complexity of propagation by a factor of $O(n)$.

If a filtering algorithm is incremental and this comes with a low cost, then propagating the constraint is more efficient by responding to each propagation event individually than by responding only once after all events accumulate. For instance, consider the algorithm of the lexicographic ordering constraint $\vec{X} \leq_{lex} \vec{Y}$ which is incremental. Due to the other constraints on $\vec{X}$ and $\vec{Y}$, the algorithm can be triggered by several propagation events. This could be problematic for two reasons. First, we need to propagate the constraint several times. Second, while handling one event, we are not aware of the impact of the other events on the domains, and thus the algorithm may not prune all the values that need to be pruned. This means that there may be many unnecessary propagations which may result in increased run-times. This has motivated us to design another algorithm where we collect all the events and respond only once with one invocation of the algorithm. Our experiments have shown that the initial algorithm is faster than the new algorithm. Even if we have to deal with many propagation events, the efficiency of the original algorithm can overcome the cost of delaying events and maintaining a propagation queue.

9.2.3 Combining Constraints

A strategy for developing global constraints that will 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. What are the benefits of combining constraints together? Is it always a good idea to combine constraints?

Why Combine?

We have argued in Section 9.2.2 that one way of propagating a global constraint is to decompose it into simpler constraints. The total pruning obtained by the propagation of each simpler constraint is likely to be weaker than maintaining GAC on the original constraint, as the global view of the constraint is lost in the decomposition. We can see decomposing a constraint c into simpler constraints $c_1, \ldots, c_n$ as combining $c_1, \ldots, c_n$ together to get a new constraint c. Hence, an advantage of combining constraints is to get additional pruning during propagation.

One of the most common criticisms of using ordering constraints to break symmetry is the possibility that the labelling heuristic and the symmetry breaking constraints conflict, resulting in larger search trees and longer run-times. A constraint which combines together symmetry breaking and problem constraints can give additional pruning, and this can help compensate for the labelling heuristic trying to push the search in a different direction to the symmetry breaking constraints. For instance, a model of the BIBD problem is a matrix with row and column symmetry. We have shown that we can get worse results by posting lexicographic ordering constraints on the rows and columns than no symmetry breaking when solving BIBDs with a labelling heuristic which conflicts with the lexicographic ordering constraints. As the rows and the columns of the matrix are also constrained by sum constraints, an alternative strategy is to replace each lexicographic ordering constraint with the constraint which combines the lexicographic ordering with sum constraints. We have shown that this gives much smaller search trees and shorter run-times than no symmetry breaking even though the labelling heuristic conflicts with the symmetry breaking constraints. Combining constraints is thus a step towards tackling a major drawback of using symmetry breaking constraints.

Why Not Combine?

Our experiments with the lexicographic ordering with sum constraints have shown that this combination of constraints is only useful when the symmetry breaking conflicts with the labelling heuristic, the labelling heuristic is poor, or there is a very large search space to explore. This has led us to investigate the reasons of this.

Katsirelos and Bacchus have proposed a simple heuristic for combining constraints together [KB01]. The heuristic suggests grouping constraints together if they share many variables in common. This heuristic would suggest that combining the lexicographical ordering and sum constraints would be very useful as they intersect on many variables. However, this ignores how the constraints are propagated. Since the lexicographic ordering constraint is concerned only with the variables at a certain position, we have observed that there is often only a limited interaction between the lexicographic ordering constraint and sum constraints. This explains why combining the 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 the lexicographic ordering constraint 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 [CB02a]. 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 have indicated no gain over posting lexicographic ordering constraints between the adjacent vectors. 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 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.

9.3 Limitations

In this section, we consider the limitations of our research from three points of view: the scope of the research, the use of ordering constraints for breaking row and column symmetry, and the value of algorithms for the ordering constraints.

9.3.1 Scope of the Dissertation

In this work, we have focused on one type of row and column symmetry. We have defined that a 2-dimensional matrix has row (resp. column) symmetry iff its rows (resp. columns) represent indistinguishable objects and are therefore symmetric. The entire work presented in this dissertation is thus applicable to this type of row and column symmetry.

There are other kinds of row and column symmetry, and posting ordering constraints on the rows and columns may not be the right way of breaking these symmetries.

As an example, consider that we want to generate a Gray code. A Gray code[1] represents each number in the sequence of integers $0 \dots 2^n - 1$ as a binary string of length n in an order such that adjacent integers have Gray code representations that differ in only one bit position. For instance, the following is a 3 bit Gray code sequence:

$$\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 1 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{array}$$

A model of this problem is an $n \times 2^n$ 0/1 matrix where the rows represent the integers and the columns represent the positions of bits in the integers. As the order of the positions are not important, all the columns are symmetric. Whilst the rows can be symmetric, swapping some rows may violate the problem constraints. For instance, swapping the first two integers in the example above gives us:

$$\begin{array}{ccc}
0 & 0 & 1 \\
0 & 0 & 0 \\
0 & 1 & 1 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{array}$$

in which the second and the third integers have Gray code representations that differ in two bit positions. However, swapping the first and the third integers of the original sequence does not violate the problem constraints:

$$\begin{array}{ccc}
0 & 1 & 1 \\
0 & 0 & 1 \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{array}$$

We cannot, however, know at the modelling level which rows can be symmetric or not, as this is determined only during search as the variables are assigned. Consequently, we cannot add unconditional ordering constraints to the model in order to break the symmetry between the rows. This type of symmetry is very similar to *conditional symmetries* defined in [GMS03].

[1] `http://mathworld.wolfram.com/GrayCode.html`

9.3.2 Breaking Row and Column Symmetry with Ordering Constraints

Even though we have shown that ordering constraints can effectively break row and column symmetry, we have not tackled some issues raised by the use of ordering constraints for breaking symmetry. We discuss the limitations from a theoretical and a practical perspective.

Theoretical Limitations

We have not found any polynomial set of constraints to break all row and column symmetries. Despite the attempts of several researchers [FH03][FP02a][FJM03], this remains an open question in the literature. Consequently, we do not necessarily break all symmetries. Lexicographic ordering is a total ordering and thus imposing lexicographic ordering constraints in one dimension of a matrix breaks all the symmetries of that dimension. However, enforcing both the rows and columns to be lexicographically ordered may leave some row and column symmetry. As multiset ordering is a preordering, imposing multiset ordering constraints in one or both dimensions may not break all the symmetries. The situation is not different when we enforce multiset ordering in one dimension and lexicographic ordering in the other dimension.

Given ordering constraints which are not implied by the problem constraints, the effectiveness of the constraints in breaking row and column symmetries depend on the labelling heuristic used to explore the search space. It is hard therefore to judge whether we can significantly reduce the size of the search space and time to solve the problem unless we carry out an experimental study. Often, alternative ways of searching for solutions need to be tried in order to find the best way of solving the problem with the ordering constraints.

Practical Limitations

We have tested the effectiveness of the ordering constraints in breaking row and column symmetries on a wide range of problems. For each problem, we have carried out an initial experimentation to find a good labelling heuristic for solving the problem. We have tried many static orderings such as labelling the matrix along its rows or columns. For the problems where the domains contain more values than 0/1, we have also tried the popular dynamic heuristic which selects next the variable that has the smallest domain. Surprisingly, this dynamic heuristic did not work well for any of the problems considered. Hence, we picked the best static heuristic for each problem to test the effectiveness of the ordering constraints. However, there are many other static and dynamic heuristics that we can try for each problem, and they may have dramatic effects on the effectiveness of the ordering constraints. We have not addressed this issue in this dissertation.

The labelling heuristic we use and the ordering constraints may conflict, resulting in larger search trees and longer run-times. If the heuristic we use is not necessarily a good one then switching to another heuristic may overcome the problem. However, if we have a very good heuristic which seems to solve large instances of the problem very efficiently but which conflicts with the ordering constraints, then using ordering constraints may not be an effective way of tackling row and column symmetry. We see evidence of this in solving the word design problem in Chapter 8.

9.3.3 Algorithms for Ordering Constraints

We have argued in Section 9.2.3 that we can see decomposing a constraint c into simpler constraints $c_1, \ldots, c_n$ as combining $c_1, \ldots, c_n$ together to get a new constraint c, and that an algorithm which maintains GAC on c is likely to be the most effective way of propagating c. We have provided both theoretical and empirical evidence of this with the algorithms of the lexicographic ordering constraint, the lexicographic ordering with sum constraints, and the multiset ordering constraint. However, using such an algorithm can sometimes be unfavourable.

Good Decompositions

We have shown that the lexicographic ordering constraint can be decomposed into a conjunction of constraints. This decomposition does the same pruning as the GAC algorithm when solving BIBDs using a static labelling heuristic. Therefore, the algorithm does not bring any gain over the decomposition in terms of the amount of pruning. We should, however, note that the algorithm is much more efficient than the decomposition.

We have compared the decomposition and the algorithm also using the progressive party problem. This time we have used a dynamic labelling heuristic. Our results have shown that for two instances the decomposition and the algorithm create the same search tree and spend the same time. For other two instances, however, we obtain much smaller search trees and shorter run-times using the decomposition. For the remaining five instances, the algorithm is superior to the decomposition.

Poor Combinations

According to our current experiments, the filtering algorithm of the combination of the lexicographic ordering constraint and two sum constraints is not useful when the problem is satisfiable, or when the labelling heuristics are neither poor nor conflict with the lexicographic ordering constraints.

9.4 Future Work

In the future, we plan to identify more useful patterns in constraint programs, investigate a number of interesting directions in breaking row and column symmetry with ordering constraints, improve the propagation of our ordering constraints, test whether the ordering constraints are useful also in other application areas, and finally formalise our heuristic on combining constraints.

9.4.1 Identification of Common Constraint Patterns

To help tackle the difficulty of effective modelling, we need to identify more recurring patterns in constraint programs and devise special-purpose methods to support these patterns [Wal03]. Possible interesting patterns are other classes of row and column symmetries.

As argued in Section 9.3, the columns of the matrix modelling the Gray code generation are symmetric, but two rows can be swapped provided that this does not violate the problem constraints. Whether two rows are symmetric or not can be determined only during search as the variables get assigned. To the best of our knowledge, this class of row and column symmetries has not been tackled. Another type of row and column symmetry is observed in the matrix model of the magic squares problem (prob019 in CSPLib). Two columns can be swapped iff the same columns of the matrix obtained by rotating the original matrix 90° clockwise are also swapped. Even though the symmetries can be broken using generic symmetry breaking methods, we are not aware of any particular study on this class of symmetries.

9.4.2 Breaking Row and Column Symmetry with Ordering Constraints

There are many interesting and potentially useful directions to explore in breaking row and column symmetry with ordering constraints. These include, but are not limited to, the investigation of more ordering constraints, comparison of our approach with other methods, assessment of the effectiveness of the ordering constraints in a more elaborate way, looking into ways of detecting row and column symmetry, and finally development of heuristics to choose between the ordering constraints.

More Ordering Constraints

An immediate question following the ordering constraints we have explored in Chapter 4 is whether there are other interesting orderings of vectors that we can impose to break row and column symmetry. To benefit from the complimentary strengths of different orderings, we should consider seeking ways of merging different ordering constraints. For instance, we can enforce $\vec{X} \leq_m \vec{Y}$ in such a way that $\vec{X} \leq_{lex} \vec{Y}$ is imposed whenever $\vec{X} =_m \vec{Y}$ is entailed. Another approach is to investigate whether we can post additional constraints together with our ordering constraints to break more symmetries. For instance, Frisch *et al.* have shown in [FJM03] that we can add the *allperm* constraint (which enforces that the first row is lexicographically less than or equal to all permutations of all other rows) together with the lexicographic ordering constraints on the rows and columns. Experimental results show that more symmetries can be broken effectively by adding the *allperm* constraint.

Comparison with Other Methods

Meseguer and Torras in [MT01] propose a variable ordering heuristic which selects first the variables involved in the largest number of symmetries local to the current state of search. This heuristic has been applied to solving large instances of the BIBD problem modelled using a matrix with row and column symmetry. The results indicate that there is a much higher likelihood of finding solutions early when this heuristic is used. An important question that we would like to answer is how our approach to breaking row and column symmetry compares in practice to the heuristic by Meseguer and Torras. For instance, when solving an optimisation problem, is it better to use the ordering constraints or the heuristic method? An alternative research direction is to investigate how such an heuristic method can successfully be combined with the ordering constraints, and in what ways a combined method would be preferable to using the heuristic or the ordering constraints alone.

Effectiveness of the Ordering Constraints

We have tested the effectiveness of the ordering constraints in breaking the row and column symmetries of a problem by (1) adding them to our model statically; (2) using a static labelling heuristic which seems to be a good way of solving the problem according to our initial experimentations. Interesting directions of research are to (1) investigate how the ordering constraints can be enforced dynamically during search, and to study the gains in effective symmetry breaking by taking such a dynamic approach in preference to the static approach taken so far; (2) to employ the ordering constraints for breaking the row and column symmetries of problems that are best solved by using dynamic labelling heuristics, and analyse how the search space changes in the presence of the ordering constraints.

Detecting Row and Column Symmetry

A modelling tool which can automatically detect row and column symmetry in a matrix and which can assist modellers by providing useful information such as which ordering constraints can be utilised, whether all symmetries can be broken using a linear number of constraints, etc would make the contributions of this dissertation reachable to a wider audience. Even though detecting symmetries has been shown to be graph isomorphism complete in the general case [Cra92], row and column symmetry could be detected using an automated modelling tool that has high-level constructs such as a relation between two indistinguishable sets [BFM03].

Heuristics

Our investigation in Chapter 4 has resulted in many ordering constraints that we can utilise to break row and column symmetries. Often, alternative ways of searching for solutions need to be tried in order to find the best way of solving the problem with the ordering constraints. This gives us many choices to consider when we want to break row and column symmetry. We therefore need to develop heuristics for deciding which ordering constraints are likely work well with a given labelling heuristic, as well as which labelling heuristics to use in accordance with the given ordering constraints.

9.4.3 Propagation of Ordering Constraints

We can improve the propagation of the ordering constraints by taking into account shared variables and multiple vectors.

Shared Variables

We have already shown how we can extend the algorithm of the lexicographic ordering constraint for vectors whose variables are repeated and shared. We can envisage how such an enhancement can be done also to the algorithm of the multiset ordering constraint. As the algorithm works on the occurrence vectors, and shared variables contribute to the occurrences of the same value, support for a value v of a variable V can be found by increasing/decreasing the occurrence of v according to how many times V occurs in the same vector. If a variable V is shared by the two vectors in the constraint, then we can eliminate one occurrence of V from each vector and consider the resulting vectors. We may consider shared and repeated variables also in the filtering algorithm of the lexicographic ordering with sum constraints.

Multiple Vectors

We have argued that combining the lexicographic ordering constraint with other constraints may not always be beneficial because lexicographic ordering is concerned only with the variables at a certain position and this gives a very limited interaction with the other constraints. Our experiments on BIBDs have shown that combining a chain of lexicographic ordering constraints does not bring any benefits. Hence, we do not expect significant gains by combining a chain of lexicographic ordering and sum constraints.

Unlike lexicographic ordering, multiset ordering is concerned with all the variables in the vectors. Each variable is examined during propagation. We therefore expect promising results by combining a chain of multiset ordering constraints, and want to devise an algorithm for this combination of the constraints. We should, however, note that even though all variables are examined, whether their domains are pruned depends on how the minimum and the maximum elements in the domains compare to two values (i.e. α and β) computed during propagation. It may well be the case in practice that only few domains are pruned, resulting in limited interaction with the other multiset ordering constraints. It is therefore not obvious to predict how valuable such a combination of constraints would be. In case of discouraging results, we should investigate whether combining a chain of ordering constraints are useful in general, considering the transitivity property of orderings.

9.4.4 Use of Ordering Constraints

So far we have used our ordering constraints only for symmetry breaking. One possible application for lexicographic ordering is to multi-criteria optimisation problems where the objective function consists of features which are ranked. In the lexicographic minimisation problem, objective vectors are compared lexicographically [EG00]. Another application for multiset ordering is to fuzzy CSPs. A fuzzy constraint associates a degree of satisfaction to an assignment tuple for the variables it constrains. To combine degrees of satisfaction, we can collect a vector of degrees of satisfaction, sort these values in ascending order, and compare them lexicographically [Far94]. This *leximin* combination operator induces an ordering identical to the multiset ordering except that the lower elements of the satisfaction scale are the more significant. It is simple to modify the multiset ordering constraint to consider the values in a reverse order.

To solve lexicographic minimisation and leximin fuzzy CSPs, we can use branch and bound, adding the related ordering constraint when we find a solution to ensure that future solutions are smaller/greater in the ordering.

9.4.5 Combining Constraints

Based on our experiences on combining a lexicographic ordering constraint with other constraints, we have suggested a new heuristic for combining constraints: the combination should like to prune a significant number of shared variables. We have a support for the usefulness of this heuristic in [BR98] where a sum constraint on a vector of variables and an *all-different* constraint on the same variables are combined together using the conjunctive consistency framework. Each variable interacts with every other variable in both constraints, so the combination is expected to increase the amount of constraint propagation. Indeed, the experimental results in [BR98] show that the extra pruning obtained by combining these constraints significantly reduces search effort. We should therefore formalise our heuristic and test it on diverse problems.

9.5 Conclusions

We have put forwards significant evidence to support our thesis:

> *Row and column symmetry is a common type of symmetry in constraint programming. Ordering constraints can effectively break this symmetry. Efficient global constraints can be designed for propagating such ordering constraints.*

We have discussed the general lessons we have learnt about symmetry breaking with ordering constraints, propagating constraints, and combining constraints. We have identified the major limitations of our research. In particular, we have criticised the scope of the dissertation, the use of ordering constraints for breaking row and column symmetry, and the value of algorithms for the ordering constraints. We have then described some directions for future work. More specifically, we have pointed out that we want to identify more useful patterns in constraint programs, investigate a number of interesting topics in breaking row and column symmetry with ordering constraints, improve the propagation of our ordering constraints, and test whether the ordering constraints are useful also in other application areas.

Breaking row and column symmetry is now a very active area of research [GS01][GPS02] [GHK02] [Pug02a] [FP02a][CB02b][CB02a][GHKL03] [FJM03][Pug03b][Pug03c][Pea03] [FH03][LL03]. This dissertation describes some of the first work for efficiently and effectively dealing with row and column symmetries.

Bibliography

The numbers in braces indicate on which pages each citation occurred.

[Apt03] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003. {17, 31, 32}

[ARMS02] F. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the 39th Design Automation Conference (DAC-02)*, pages 731–736. ACM, 2002. {18}

[Bal38] W.W. Rouse Ball. *Mathematical Recreations and Essays.* Macmillan, 11 edition, 1938. {49}

[BC97] N. Bleuzen-Guernalec and A. Colmerauer. Narrowing a block of sortings in quadratic time. In G. Smolka, editor, *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP-97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 1997. {190}

[BC00] N. Bleuzen-Guernalec and A. Colmerauer. Optimal narrowing of a block of sortings in optimal time. *Constraints*, 5(1-2):85–118, 2000. {190}

[Bel02] N. Beldiceanu. Global constraints. Tutorial presented at the 8th International Conference on Principles and Practice of Constraint Programming (CP-02), 2002. Available at `ftp://ftp.sics.se/pub/isl/papers/globalctr.pdf`. {34}

[Ben94] B. Benhamou. Study of symmetries in constraint satisfaction problems. In A. Borning, editor, *Proceedings of the 2nd Workshop on Principles and Practice of Constraint Programming (PPCP-94)*, volume 874 of *Lecture Notes in Computer Science*, pages 246–254. Springer, 1994. {37}

[BFM03] A. Bakewell, A.M. Frisch, and I. Miguel. Towards automatic modelling of constraint satisfaction problems: a system based on compositional refinement. In Notes of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, CP-03 Post-conference Workshop, 2003. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/`. {36, 232}

[BFP88] C. A. Brown, L. Finkelstein, and P. W. Purdom Jr. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Proceedings of the 6th International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC-6)*, volume 357 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1988. {18, 37}

[BFP96] C. A. Brown, L. Finkelstein, and P. W. Purdom Jr. Backtrack searching in the presence of symmetry. *Nordic Journal of Computing*, 3(3):203–219, 1996. {37}

[BFR99] C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc-consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999. {34}

[Bor98] J.E. Borrett. *Formulation selection for constraint satisfaction problem: a heuristic approach.* PhD thesis, University of Essex, 1998. {18, 41}

[BR75] R. Bitner and M. Reingold. Backtrack programming techniques. *Communications of the Association for Computing Machinery*, 18(11):651–656, 1975. {31}

[BR97] C. Bessière and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In M. E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 398–404. Morgan Kaufmann, 1997. {34, 155}

[BR98] C. Bessière and J.C. Régin. Local consistency on conjunctions of constraints. In Notes of the ECAI-98 Workshop on Non-binary Constraints, 1998. {34, 155, 222, 233}

[BR99] C. Bessière and J.C. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In J. Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 1999. {34}

[BT01] J.E. Borrett and E.P.K. Tsang. A context for constraint satisfaction problem formulation selection. *Constraints*, 6(4):299–327, 2001. {18, 41}

[BvH03] C. Bessière and P. van Hentenryck. To be or not to be...a global constraint. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 789–794. Springer, 2003. {33}

[BW99] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In J. Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999. {19, 20, 37, 59, 61, 80, 220}

[BW02] R. Backofen and S. Will. Excluding symmetries in constraint-based search. *Constraints*, 7(3-4):333–349, 2002. {19, 20, 37, 59, 61, 80, 220}

[CB02a] M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, Swedish Institute of Computer Science, 2002. Available at `ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T--2002-18--SE.ps.Z`. {111, 155, 161, 227, 234}

[CB02b] M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, Swedish Institute of Computer Science, 2002. Available at `ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T--2002-17--SE.ps.Z`. {111, 234}

[CCLW99] B.M.W. Cheng, K.M.F. Choi, J.H.M. Lee, and J.C.K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4:167–192, 1999. {42}

[CD96] C. H. Colbourn and J.H. Dinitz. *The CRC Handbook of Combinatorial Designs.* CRC Press, 1996. {42, 116, 208}

[CGLR96] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry breaking predicates for search problems. In L. Carlucci Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pages 148–159. Morgon Kaufmann, 1996. {18, 37, 62, 64, 66, 67, 73, 78, 80, 105}

[Cra92] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In Notes of the AAAI-92 Workshop on Tractable Reasoning, 1992. {232}

[CSP] CSPLib: A problem library for constraints. Available at `http://www.csplib.org`. A technical report about the library appears as [GW99]. {18, 42, 206, 241}

[DB97] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In M. E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 412–417. Morgan Kaufmann, 1997. {32}

[DL98] K. Darbi-Dowman and J. Little. Properties of some combinatorial optimisation problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal of Computing*, 10(3):276–286, 1998. {17}

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the Association for Computing Machinery (CACM)*, 22(8):465–476, 1979. {67}

[ECL03] P. Brisset and H. El Sakkout and T. Frühwirth and C. Gervet and W. Harvey and M. Meier and S. Novello and T. Le Provost and J. Schimpf and K. Shen and M. G. Wallace. *ECLiPSe Constraint Library Manual Release 5.6*, 2003. Available at `http://www.icparc.ic.ac.uk/eclipse/doc/doc/libman/libman.html`. {109, 190}

[EG00] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 22:425–460, 2000. {233}

[ES93] E. A. Emerson and A. A. Sistla. Symmetry and model checking. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV-93)*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 1993. {18}

[FaC01] N. Barnier and P. Brisset. *FaCiLe: A Functional Constraint Library Release 1.0*, 2001. Available at `http://www.recherche.enac.fr/opti/facile/doc/`. {51, 189, 190}

[Far94] H. Fargier. *Problèmes de satisfaction de constraintes flexibles: application à l'ordonnancement de production.* PhD thesis, University of Paul Sabatier, Tolouse, 1994. {233}

[FFH$^+$01a] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in matrix models. Technical Report APES-30-2001, APES Research Group, October 2001. Available at `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`. Also in [FP01]. {76, 80}

[FFH$^+$01b] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling. Technical Report APES-36-2001, APES Research Group, October 2001. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`. Also in Notes of the CP-01 Post-conference Workshop on Modelling and Problem Formulation (Formul-01). {18, 19, 42}

[FFH$^+$02] P. Flener, A.M. 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 the 8th International Conference on Principles and Practice of Constraint programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2002. {78, 79, 80, 206}

[FH03] A. M. Frisch and W. Harvey. Constraints for breaking all row and column symmetries in a three-by-two matrix. In [GHS03], 2003. {229, 234}

[FHK$^+$02] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2002. {105, 110, 155, 161}

[FHK$^+$03] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Multiset ordering constraints. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 221–226. Morgan Kaufmann, 2003. {176, 184}

[FJM03] A.M. Frisch, C. Jefferson, and I. Miguel. Constraints for breaking more row and column symmetries. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 318–332. Springer, 2003. {80, 229, 231, 234}

[FL99] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 956–961. Morgan Kaufmann, 1999. {18}

[FL02] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, pages 83–91. AAAI, 2002. {18}

[FM01] F. Focacci and M. Milano. Global cut framework for removing symmetries. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2001. {19, 20, 38, 59, 61, 80, 81, 82, 220}

[FMW01] A.M. Frisch, I. Miguel, and T. Walsh. Modelling a steel mill slab design problem. In Notes of the IJCAI-01 Workshop on Modelling and Solving Problems with Constraints, 2001. Available at `http://www.lirmm.fr/~bessiere/nbc_workshop.htm`. {46}

[FP01] P. Flener and J. Pearson, editors. Notes of the 1sth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-01), CP-01 Post-conference Workshop, 2001. Available at `http://www.it.uu.se/research/group/astra/SymCon01/`. {36, 238, 241}

[FP02a] P. Flener and J. Pearson. Breaking all the symmetries in matrix models: results, conjectures, and directions. In [FP02b], 2002. {80, 229, 234}

[FP02b] P. Flener and J. Pearson, editors. Notes of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02), CP-02 Post-conference Workshop, 2002. Available at `http://www.it.uu.se/research/group/astra/SymCon02/`. {36, 239, 241, 242}

[Fre85] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the Association for Computing Machinery*, 32(4):755–761, 1985. {32}

[Fre91] E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 227–233. AAAI Press/The MIT Press, 1991. {37}

[Fre98] E.C. Freuder. Modelling satisfaction and satisfactory modelling: Modelling problems so constraint engines can solve them. Invited talk at the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI-98), 1998. {18, 41}

[FSS01] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001. {19, 20, 38, 59, 61, 80, 81, 82, 220}

[Gal02] J. A. Gallian. A dynamic survey of graph labelling. *The Electronic Journal of Combinatorics*, DS6, 2002. Available at `http://www.combinatorics.org/Surveys`. {78}

[GAP00] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000. Available at `http://www.gap-system.org`. {81}

[GBY01] D. Gilbert, R. Backofen, and R. H. C. Yap. Introduction to the special issue on bioinformatics. *Constraints*, 6(2-3):139, 2001. {17}

[Ger97] C. Gervet. Interval propagation to reason about sets: definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997. {36, 43}

[GHK02] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: symmetry breaking during search. In P. van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 2002. {81, 234}

[GHKL03] I.P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD using computational group theory. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 333–362. Springer, 2003. {81, 234}

[GHS03] I.P. Gent, W. Harvey, and B.M. Smith, editors. Notes of the 3rd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-03), CP-03 Post-conference Workshop, 2003. Available at `http://scom.hud.ac.uk/scombms/SymCon03/`. {36, 238, 240, 242, 243, 244}

[GIM$^+$01] I.P. Gent, R. W. Irving, D. Manlove, P. Prosser, and B. M. Smith. A constraint programming approach to the stable marriage problem. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001. {110}

[GM03] A. Guerri and M. Milano. CP-IP techniques for the bid evaluation in combinatorial auctions. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 863–867. Springer, 2003. {17}

[GMS03] I.P. Gent, I. McDonald, and B.M. Smith. Conditional symmetry in the all-interval seris problem. In [GHS03], 2003. {228}

[GPS02] I. P. Gent, P. Prosser, and 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. Available at `http://www-users.cs.york.ac.uk/~tw/ecai02/`. {110, 118, 234}

[GS00] I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00)*, pages 599–603. IOS Press, 2000. {19, 20, 37, 59, 61, 80, 81, 220}

[GS01] I. P. Gent and B. M. Smith. Reducing symmetry in matrix models: SBDS vs. constraints. Technical Report APES-31-2001, APES Research Group, October 2001. Available at `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`. Also in [FP01]. {80, 81, 234}

[GW99] I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical Report APES-09-1999, APES Research Group, 1999. Available at `http://www.csplib.org`. A shorter version appears in the *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*. {237}

[Har02] W. Harvey. Personal e-mail communication, 2002. {110, 118}

[HE80] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980. {31}

[Hni02] B. Hnich. A matrix model of the word design problem, 2002. Available in CSPLib [CSP]. {53, 55}

[Hni03] B. Hnich. *Function variables for constraint programming.* PhD thesis, Uppsala University, 2003. Available at `http://publications.uu.se/theses/91-506-1650-1/`. {18, 41, 42}

[ID93] C. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. J. M. Claesen, and R. Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL-93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–111. North-Holland, 1993. {18}

[ILO02] ILOG S.A. *ILOG Solver 5.3 Reference and User Manual*, 2002. {12, 13, 14, 47, 48, 51, 55, 79, 111, 119, 156, 162, 189, 193, 196, 199, 212}

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press, 1970. {67}

[KB01] G. Katsirelos and F. Bacchus. GAC on conjunctions of constraints. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 610–614. Springer, 2001. {160, 227}

[KS02] Z. Kiziltan and B. M. Smith. Symmetry breaking constraints for matrix models. In [FP02b], 2002. {188}

[KT03a] I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume

2833 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2003. {35, 189}

[KT03b] I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. Technical Report MPI-I-2003-1-013, Max-Planck-Institut für Informatik, 2003. Available at `http://domino.mpi-sb.mpg.de/internet/reports.nsf/AG1NumberView?OpenView`. {35, 189}

[KvB97] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997. {31}

[KW02] Z. Kiziltan and T. Walsh. Constraint programming with multisets. In [FP02b], 2002. {178}

[Lam03] N.H. Lam. Completing comma-free codes. *Theoretical Computer Science*, 301(1-3):399–415, 2003. {82}

[LL03] Y.C. Law and J.H.M. Lee. Expressing symmetry breaking constraints using multiple viewpoints and channeling constraints. In [GHS03], 2003. {234}

[LQTvB03] A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 245–256. Morgan Kaufmann, 2003. {35}

[LR80] C.C. Lindner and A. Rosa. Topics on steiner systems. *Annals of Discrete Mathematics*, 7, 1980. {43, 157}

[Lub85] A. Lubiw. Doubly lexical orderings of matrices. In *Proceedings of the 17th Annual Association for Computing Machinery Symposium on Theory of Computing (STOC-85)*, pages 396–404. ACM Press, 1985. {63}

[Lub87] A. Lubiw. Doubly lexical orderings of matrices. *SIAM Journal on Computing*, 16(5):854–879, October 1987. {63}

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. {17, 31}

[Mar02] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming A*, 94(1):71–90, 2002. {18}

[Mar03] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98(1-3):3–21, 2003. {18}

[MM88] R. Mohr and G. Masini. Good old discrete relaxation. In Y. Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656. Pitmann Publishing, 1988. {32}

[MS98] K. Marriott and P.J. Stuckey. *Programming with constraints.* The MIT Press, 1998. {17, 31}

[MT99] P. Meseguer and C. Torras. Solving strategies for highly symmetric CSPs. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 400–405. Morgan Kaufmann, 1999. {12, 23, 37, 38, 43}

[MT00] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP-00)*, volume 1894 of *Lecture Notes in Computer Science*, pages 306–319. Springer, 2000. {190}

[MT01] P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1-2):133–163, 2001. {23, 35, 38, 158, 231}

[nau] B.D. McKay. ***nauty** Users' Guide (Version 2.2)*. Available at `http://cs.anu.edu.au/people/bdm/nauty`. {82}

[Oz03] T. Müller. *Problem Solving with Finite Set Constraints in Oz. A Tutorial*, 2003. Available at `http://www.mozart-oz.org/documentation/fst/`. {55}

[Pea03] J. Pearson. Comma free codes. In [GHS03], 2003. {82, 234}

[PS98] L. Proll and B. M. Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS Journal on Computing*, 10(3):265–275, 1998. {12, 13, 46, 47, 114, 115, 116}

[PS03] K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 930–934. Springer, 2003. {78}

[Pug93] J.F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In H. J. Komorowski and Z. W. Ras, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, volume 689 of *Lecture Notes in Computer Science*, pages 350–361. Springer, 1993. {21, 27, 36, 37, 62}

[Pug98] J.F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI-98)*, pages 359–366. AAAI Press/The MIT Press, 1998. {35}

[Pug02a] J.F. Puget. Slides for the presentation of [Pug02c], 2002. Available at `http://www.it.uu.se/research/group/astra/SymCon02/`. {23, 81, 158, 208, 234}

[Pug02b] J.F. Puget. Personal communication, 2002. {82}

[Pug02c] J.F. Puget. Symmetry breaking revisited. In P. van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice* *of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 447–461. Springer, 2002. {81, 206, 243}

[Pug03a] J.F. Puget. Slides for the presentation of [Pug03b], 2003. {23}

[Pug03b] J.F. Puget. Symmetry breaking using stabilizers. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2003. {37, 80, 234, 244}

[Pug03c] J.F. Puget. Using constraint programming to compute symmetries. In [GHS03], 2003. {234}

[QvBL$^+$03] C.-G. Quimper, P. van Beek, A. Lopez-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 600–614. Springer, 2003. {35, 189}

[Rég94] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367. AAAI Press, 1994. {35}

[Rég96] J.C. Régin. Generalized arc consistency for global cardinality constraints. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI-96)*, pages 209–215. AAAI Press/The MIT Press, 1996. {35, 189, 193}

[Ros00] F. Rossi. Constraint (logic) programming: a survey on research and applications. In K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints*, volume 1865 of *Lecture Notes in Computer Science*, pages 40–74. Springer, 2000. {17, 31}

[Ros03] F. Rossi, editor. *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*. Springer, 2003. {36}

[RP98] P. Roy and F. Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In Notes of the ECAI-98 Workshop on Non-binary Constraints, 1998. {37}

[RR00] J.C. Régin and M. Rueher. A global constraint combining a sum constraint and difference constraints. In R. Dechter, editor, *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP-00)*, volume 1894 of *Lecture Notes in Computer Science*, pages 384–395. Springer, 2000. {139, 155}

[SBHW96] B.M. Smith, S.C. Brailsford, P.M. Hubbard, and H.P. Williams. The progressive party problem: integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996. {12, 17, 52, 53, 112, 197, 210}

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986. {18, 42}

[SFV95] T. Schiex, H. Fargier, and G. Verfaille. Valued constraint satisfaction problems: hard and easy problems. In C. S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 631–637. Morgan Kaufmann, 1995. {188}

[Shl01] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In H. Kautz and B. Selman, editors, *Electronic Notes in Discrete Mathematics*, volume 9. Elsevier, 2001. {79, 80}

[SIC03] Swedish Institue of Computer Science. *SICStus Prolog User's Manual, Release 3.10.1*, April 2003. Available at `http://www.sics.se/isl/sicstuswww/site/documentation.html`. {51, 161, 189, 190}

[Smi01] B. M. Smith. Reducing symmetry in a combinatorial design problem. Technical Report 2001.01, School of Computing, University of Leeds, January 2001. Also in Notes of the 3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR-01). Available at `http://scom.hud.ac.uk/staff/scombms/papers.html`. {36}

[Tar85] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985. {85}

[Tsa93] E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993. {30, 74}

[vH01] W.J. van Hoeve. The alldifferent constraint: a survey. Unpublished manuscript, 2001. Available at `http://homepages.cwi.nl/~wjvh`. {35}

[vH02] P. van Hentenryck, editor. *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*. Springer, 2002. {36}

[vHMPR99] P. van Hentenryck, L. Michel, L. Perron, and J.C. Régin. Constraint programming in OPL. In G. Nadathur, editor, *Proceedings of the International Conference on the Principles and Practice of Declarative Programming (PPDP-99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116. Springer, 1999. {12, 18, 50, 51}

[vHS97] P. van Hentenryck and V. A. Saraswat. Constraint programming: strategic directions. *Constraints*, 2(1):7–33, 1997. {17}

[vHSD98] P. van Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998. {32}

[Wal96] M. G. Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996. {17, 31}

[Wal99] J. P. Walser. *Integer Optimization by Local Search – A Domain-Independent Approach*, volume 1637 of *Lecture Notes in Artificial Intelligence*. Springer, 1999. {112, 197}

[Wal01] T. Walsh, editor. *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001. {36}

[Wal03] T. Walsh. Constraint patterns. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 53–64. Springer, 2003. {18, 41, 230}

[WNS97] M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: a platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997. {109}

Index