The Length-Lex Representation for Constraint Programming over Sets

by

Yip, Yue Kwen Justin

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May, 2011

This dissertation by Yip, Yue Kwen Justin is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____
Prof. Pascal Van Hentenryck, Director

Recommended to the Graduate Council

Date _____ _____
Prof. Claire Mathieu, Reader

Date _____ _____
Prof. Carmen Gervet, Reader
(German University in Cario)

Approved by the Graduate Council

Date _____ _____
Prof. Peter M. Weber
Dean of the Graduate School

Acknowledgements

This doctoral degree is to some extent an unintended consequence for someone who is impatient and earned poor grades in college. For me, the main reason for going to graduate school abroad was the genuine sibling rivalry among my high-achieving cousins Avan and Fiona, and brother Austin. I came to Brown as a master's student four years ago, with the hope of getting a decent job in the Silicon Valley after graduation. Under Pascal's excellent guidance, I discovered that conducting research is fun and stimulating, and that pursuing a doctoral degree is more straightforward than I had anticipated. I have been granted the luxury of focusing on a single task and have been sheltered from a lot of external pressures. These years have been the most exciting time of my life to date.

I would like wholeheartedly to thank Pascal, who encouraged me to pursue a doctoral degree and enlightened me to focus on one goal at a time (which resulted in failing a class). He has given me much freedom in conducting research, as well as vacations and particularly flexible working hours. I am also grateful for his reassuring guidance and his comments on my Facebook wall during stressful moments, and when I was feeling discouraged.

For me, research in optimization is all about competition. I got ample opportunities to compete with top-notch researchers from all over the world. I wish to thank all my competitors; they are the ones who keep me awake at night, challenge my intellectual limit, push me to work hard and make me stronger. I am also thankful to Jimmy Lee, who gave me a C+ in his constraint programming class, hired me to work in his research group for a year, and recommended me to Brown.

It is grateful to have met a lot of new friends at Brown. The optimization gang: Carleton, Serdar, Yuri, Kevin, Maire, Pierre, and Gregoire. The badminton players: Ohm, Jonah, Jim, Nell, Xu, Minh, See, Jude, Ben, Qile, Xi, Kang, and Steve. My running mates: Olya, Aparna, and Micha. The room-402 crowd: Jesse, Eric, Laura, and Anna. And of course Wenjin, Aggeliki, FengHao, and

DeQing, who shared a lot of great moments with me.

I would like to thank my father Tim and mother Manie for their unconditional support in allowing me to achieve my goal in life. And finally, I would like to thank Daisy for being with me, especially during the difficult and stressful times.

Providence, May 2011 Justin Yip

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Constraint programming is a declarative programming paradigm for solving hard combinatorial problems. From a modeling standpoint, its key idea is to view a combinatorial optimization application as a combination of a model and a search procedure and to express models and search procedures at a high level of abstraction. In particular, a constraint-programming model expresses a complex application in terms of its combinatorial substructures, providing users with a high-level modeling language and communicating the problem structure to the underlying solver. From a computational standpoint, the key idea underlying constraint programming is to use combinatorial constraints to prune the search space by removing values from the variable domains that cannot appear in any feasible solution and to use feasibility information for branching. Constraint programming over finite domain variables has been extensively studied since the development of the CHIP system in the 1980s. In recent years, increased attention has been devoted to constraint programming over more complex combinatorial objects such as sets, graphs, and permutations.

This thesis considers constraint programming over set variables, since sets are natural and fundamental combinatorial objects to model a wide range of configuration problems naturally. However, set variables raise fundamentally novel representation issues, since their domains often contain an exponential number of sets. How to represent this exponential number of sets and how to use this representation for pruning the search space effectively is the fundamental open issue in constraint programming over sets.

This thesis focuses the length-lex representation that was shown to offer some theoretical advantages over earlier attempts [29]. Reference [29] introduced the length-lex domain, demonstrated its potential benefits, and left a number of significant open issues about its theoretical and algorithmic properties, as well as about experimental behavior of the domain. As a result, this thesis aims at showing that

length-lex is an effective set domain representation for constraint programming.

This thesis is supported by a number of theoretical, algorithmic, and experimental contributions, which include a series of generic polynomial-time complete filtering algorithms, primal/dual filters, advanced modeling techniques, and exponential checkers and propagators for set constraints. These techniques are highly modular, which makes it possible to apply them on several standard benchmarks. The theoretical and algorithmic results have been implemented in a prototype which is shown to be several orders of magnitude faster than earlier techniques on these benchmarks. These results provide both theoretical and empirical valiadation to the thesis that length-lex is an efficient, effective, and robust representation for constraint programming over sets.

1.1 Outline

This thesis assumes readers have basic background knowledge in constraint programming. Readers may refer to Van Hentenryck [67, 68], Marriott and Stuckey [49], Dechter[12], and Apt [1].

Background Chapter 2 presents set variables and their applications. It gives a simple and elegant model for the social golfer problem, which is used as a running example. Computational issues with the set domains are also discussed, as well as prior art. Chapter 3 introduces the length-lex set representation which is at the core of this thesis.

Basic Generic and Efficient Propagators The thesis contributions start in Chapters 4 which gives a simple and generic method for implementing a bound-consistent propagator for unary constraints. It proposes a special class of length-lex interval which enjoys the compactness of a subset-bound lattice, making inferencing extremely easy. It then demonstrates how to implement a propagator only assuming a feasibility routine for this special class. Chapter 5 generalizes the idea o a

binary constraint and, more generally, to fixed-arity constraints.

Symmetry-Breaking Propagators Chapter 6 presents novel symmetry-breaking techniques using the length-lex variables. It first shows that combining symmetry-breaking constraints with binary constraint dramatically reduces the search space. It then introduces a global symmetry-breaking alldisjoint filtering rule which gives a global perspective for propagation. Finally, it demonstrates a dual modeling framework for breaking value symmetries, demonstrating that this is an ideal vehicle for symmetry breaking.

Exponential Propagators Chapter 7 discusses the length-lex representation from another perspective: the impact of using the length-lex representation on the constraint-propagation algorithm. It shows that, even for simple unary intersection constraints, the constraint propagation algorithm may take exponential time to converge to a fixpoint. However, such negative theoretical result cannot explain the superiority of length-lex over traditional set representation. The chapter argues that the exponential behavior is indeed a feature of length-lex, which moves some of the exponential behavior from the search to the constraint-propagation algorithm, where the constraint semantics can be exploited. The chapter pushes the result even further by introducing exponential propagators to speed up the convergence rate of the constraint-propagation algorithm. Variants of the exponential propagator are also discussed.

Global Set Propagators Chapter 8 introduces a few checker and filters for intersection constraints that provides global perspectives. What makes them particular appealing is that they are independent of the underlying domain representation. The evaluation shows that they effectively reduce search space regardless of the representation used.

Interplay with Other Set Domains One of the key aspects of constraint programming is its high modularity. It is important for length-lex be able to work seamlessly with other domain representations. Chapter 9 addresses this issue and gives a lightweight method for applying multiple domain representations on the same model. It also introduces exponential propagators working on the combination of the length-lex and subset-bound domains.

Advanced Symmetry Breaking Basic symmetry, e.g., interchangeability between two variables, can be removed by simply applying the LexLeader method. Eliminating compositional symmetry, e.g., interchangeability among a set of variables and a set of values, is more difficult, since it requires an exponential number of ordering constraints. A common way to avoid posting too many constraint is to use only a subset and leaving some symmetrical solutions and subtrees behind. But this method leads to thrashing behavior since the solver keeps revisiting symmetric subtrees. Chapter 10 tackles this issue on matrix models, which are closely related to set models, with exponential and complete feasibility checkers. The chapter shows that the checkers improve the performance by orders-of-magnitude when comparing with the traditional doubleLex method.

Evaluation and Conclusion Chapter 11 evaluates the length-lex set representation on several standard benchmarks and compares with other technique used. Four benchmarks are used for evaluating length-lex and they indicate that length-lex and its variants are very robust and efficient. In particular, the length-lex domain now solves many benchmark instances that were unsolvable in constraint programming before. Chapter 12 concludes this thesis and discusses future perspectives.

Appendix The appendix includes a couple of supporting models and algorithms used in the implementation. Chapter A gives different models for the social golfer problem. Chapter B presents the basic idea of very efficient generic bound-consistent algorithms for unary and binary length-lex constraints, further exploiting the semantics of length-lex and maintaining some incremental data structure for efficient filtering. Chapter C focuses on effective propagation on subset-bound variables. It supports the claim that it is important to focus on propagation for solving set models.

1.2 Publications

Part of the work presented in this thesis has been published (or submitted) in the following forms:

1. Pascal Van Hentenryck, Justin Yip, Carmen Gervet, Grgoire Dooms: Bound Consistency for Binary Length-Lex Set Constraints. AAAI 2008: 375-380

2. Justin Yip, Pascal Van Hentenryck: Length-lex bound consistency for knapsack constraints. SAC 2009: 1397-1401

3. Justin Yip, Pascal Van Hentenryck: Evaluation of Length-Lex Set Variables. CP 2009: 817-832

4. Justin Yip, Pascal Van Hentenryck, Carmen Gervet: Boosting Set Constraint Propagation for Network Design. CPAIOR 2010: 339-353

5. Justin Yip, Pascal Van Hentenryck: Exponential Propagation for Set Variables. CP 2010: 499-513

6. Justin Yip, Pascal Van Hentenryck: Symmetry Breaking via LexLeader Feasibility Checker. IJCAI 2011.

7. Justin Yip, Pascal Van Hentenryck: Checking and Filtering Global Set Constraints. CP 2011, submitted.

Chapter 2

Set Variables

2.1 Overview

A *set variable* takes *a set of values* which generalizes the classical finite-domain world in which variable takes exactly one value. Set variables are natural vehicles for modeling problems which exhibits variable interchangeability. They allow the model to preserve the problem structure, enable propagators to exploit the semantics to reduce the search space effectively.

A variety of problem classes can be modeled naturally with set variables, including scheduling, network configuration, coding, combinatorics, and cryptography. These problem classes typically exhibit numerous symmetries. It is essential for the modeling tool to take symmetry into account in order to avoid visiting symmetrically equivalent subtrees during search. Set variables is an effective tool. This chapter illustrates these functionalities on a scheduling problem which exhibits many complex symmetries. The problem has been extensively studied in the constraint-programming community. Different aspects of the problem has been widely investigated: The model[3, 44], breaking symmetries[19, 61, 16, 22, 46, 54], constraint propagation[55, 56, 70], and search strategies[32, 10, 14].

2.1.1 The Social Golfer Problem

The *social golfer problem* consists in organizing a social event for a group of golfers across several weeks such that each participant has the maximum chance to socialize. It is defined by a 3-tuple

parameter (G, S, W). The task is to organize a golf event for $N = G \times S$ golfers. The event lasts for W weeks. Each weekend, all golfers are distributed into G groups and each group consists of S people. To maximize the *sociability* (every golfer should meet the most number of people), it is required that every pair of golfers play at the same group at most once.

Figure 2.1 illustrates a solution for instance $(3, 3, 3)$. A total of $9 = 3 \times 3$ golfers are allocated into 3 groups of 3 people for 3 weekends. Golfers are named from 1 to 9. Each row illustrates the configuration of a particular week. Every three columns form a group. Every pair of groups share at most one golfer in common.

	Group 1			Group 2			Group 3		
Week 1	1	2	3	4	5	6	7	8	9
Week 2	1	4	7	2	5	8	3	6	9
Week 3	1	5	9	2	6	7	3	4	8

Figure 2.1: Solution for Social Golfer Problem $(3, 3, 3)$

The problem can be modeled as a finite-domain CSP, where each cell in the table corresponds to one variables whose domain is the set of all golfers. The model is given in Section A.1.1. However, the model fails to take some very basic symmetry properties into account. And the solver is likely to revisit symmetrical subtrees during search.

It is important to addresses symmetries directly in the model. The social golfer problem mainly consists of 4 types of symmetry:

1. Positions within each group are interchangeable. Players 1 and 2 in group 1 of week 1 are interchangeable. Given any solution, interchanging players within a group produces another solution. (Figure 2.2)

2. Groups are interchangeable. Groups 1 and 2 can be interchanged and to produce another solution. (Figure 2.3)

3. Weeks are interchangeable. Weeks 1 and 2 can be interchanged and to produce another solution. (Figure 2.4)

4. Players are interchangeable. Player 3 is replaced by player 6, 6 by 9, and 9 by 3. The resulting table is still a solution. (Figure 2.5)

	Group 1			Group 2			Group 3		
Week 1	*2*	*1*	3	4	5	6	7	8	9
Week 2	1	4	7	2	5	8	3	6	9
Week 3	1	5	9	2	6	7	3	4	8

Figure 2.2: Positions within a Group are Interchangeable

	Group 1			*Group 2*			Group 3		
Week 1	*4*	*5*	*6*	*1*	*2*	*3*	7	8	9
Week 2	1	4	7	2	5	8	3	6	9
Week 3	1	5	9	2	6	7	3	4	8

Figure 2.3: Groups are Interchangeable

	Group 1			Group 2			Group 3		
Week 1	*1*	*4*	*7*	*2*	*5*	*8*	*3*	*6*	*9*
Week 2	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Week 3	1	5	9	2	6	7	3	4	8

Figure 2.4: Weeks are Interchangeable

	Group 1			Group 2			Group 3		
Week 1	1	2	*6*	4	5	*9*	7	8	*3*
Week 2	1	4	7	2	5	8	*6*	*9*	*3*
Week 3	1	5	*3*	2	*9*	7	*6*	4	8

Figure 2.5: Players are Interchangeable

2.1.2 Eliminating Symmetry with Set Variables

The first symmetry class can be eliminated by the use of set variables. Observe that positions within a group serve no function in the model. Constraints in the model are only concerned with who is in which group, but not the position within the group. *Hence, a group should be viewed as a set instead of a collection of positions,* and *set variables* are therefore a very appealing alternative for modeling such problem.

We give a model for the social golfer problem using set variables in Figure 2.6. We denote the set of players $\{1, ..., N\}$ as $\mathcal{P}$, groups $\{1, ..., G\}$ as $\mathcal{G}$, and weeks $\{1, ..., W\}$ as $\mathcal{W}$. $X_{w,g}$ is a set variable that corresponds to a group and its value is a subset of players (Line 2.1). Every group has a fixed cardinality of S (Line 2.2). Groups of the same week should be disjoint since any golfer can only

join one group in a week (Line 2.3). The constraint "every pair of golfers can play at most once" can be rephrased as every pair of groups should have at most once golfer in common, since otherwise the overlapping players meet in both groups (Line 2.4).

The social golfer problem is thus modeled using only 4 sets of compact constraints. The problem structure is kept in the model, which allows propagators to exploit it. The model eliminates the first symmetry class, that intra-group positions are interchangeable, with the use of set variables.

$$X_{w,g} \subseteq \mathcal{P} \qquad \forall g \in \mathcal{G}, w \in \mathcal{W} \tag{2.1}$$
$$|X_{w,g}| = S \qquad \forall g \in \mathcal{G}, w \in \mathcal{W} \tag{2.2}$$
$$X_{w,g} \cap X_{w,g'} = \emptyset \qquad \forall g < g' \in \mathcal{G}, w \in \mathcal{W} \tag{2.3}$$
$$|X_{w,g} \cap X_{w',g'}| \leq 1 \qquad \forall g, g' \in \mathcal{G}, w < w' \in \mathcal{W} \tag{2.4}$$

Figure 2.6: Social Golfer Model in Set-CSP

2.1.3 Eliminating Symmetry with Symmetry-Breaking Constraints

The second and third symmetry classes can be eliminated by applying the LexLeader method[11]. The main idea is to exclude all but the canonical solution in a symmetry class by posting symmetry-breaking constraints. The canonical solution is usually the lexicographically smallest solution in a symmetry class for a predefined ordering. Consider an assignment of the first week, $[\{1,2,3\},\{4,5,6\},\{7,8,9\}]$, there exists another symmetrical assignment by swapping the first two groups, $[\{4,5,6\},\{1,2,3\},\{7,8,9\}]$. The LexLeader method eliminates the second solution using lexicographical ordering constraint which ensures that the first group is lexicographically smaller than the second, and the second smaller than the third. For example, we define the solution illustrated in Figure 2.1 as the canonical solution, To eliminate other symmetrical solutions, we post symmetry-breaking constraints.

For group interchangeability, we post

$$X_{w,g} \prec X_{w,g'} \qquad \forall g < g' \in \mathcal{G}, \forall w \in \mathcal{W} \tag{2.5}$$

where $\prec$ is a lexicographic-ordering. The solution in Figure 2.3 is no longer a feasible solution due

to Constraints 2.5. Similarly, week interchangeability can be eliminated by posting

$$X_{w,1} \prec X_{w',1} \qquad \forall w < w' \in \mathcal{W}. \tag{2.6}$$

The solution in Figure 2.4 is eliminated.

These symmetry-breaking constraints are independent from other constraints in the model. They are all basic building blocks, which can be re-used easily at other models. They are posted on the model like the intersection constraints. We will further discuss the advantage of using these constraints as well as some associated advanced modeling techniques.

2.1.4 Eliminating Value Symmetry with Dual Modeling

Players are interchangeable in the schedule, a property called *value interchangeability*. Swapping the assignment between any pair of players preserves solution. The variable $X_{w,g}$ takes a subset of players which are interchangeable. These symmetries can be eliminated through dual modeling[17, 44]. In essence, dual modeling takes a completely opposite perspective which interchanges the role of variables and values. Value interchangeability now becomes variable interchangeability and can therefore be eliminated by the LexLeader method.

In particular, in the dual modeling method, a set of dual variables is introduced, each associated with a value. The dual variable Y_p takes a subset of week-group tuple, which indicates the group golfer p plays (Line 2.7). The primal and dual set of variables are connected with channeling constraints (Line 2.8). Interchangeability among players, which now becomes dual variable interchangeability, is eliminated by lexicographical-ordering constraints (Line 2.9).

$$Y_p \subseteq \{(w,g) \mid g \in \mathcal{G}, w \in \mathcal{W}\} \qquad \forall p \in \mathcal{P} \tag{2.7}$$

$$p \in X_{w,g} \Leftrightarrow (w,g) \in Y_p \qquad \forall g \in \mathcal{G}, w \in \mathcal{W}, p \in \mathcal{P} \tag{2.8}$$

$$Y_p \prec Y_{p'} \qquad \forall p < p' \in \mathcal{P} \tag{2.9}$$

Figure 2.7 illustrates the value of dual variables associated with players 3, 6, and 9. The second column violates the symmetry breaking constraints as $Y_3 \not\prec Y_6$. Hence the solution in Figure 2.5 is

removed by the dual-modeling method.

	Figure 2.1	Figure 2.5
Y_3	$\{(1,1),(2,3),(3,3)\}$	$\{(1,3),(2,3),(3,1)\}$
Y_6	$\{(1,2),(2,3),(3,2)\}$	$\{(1,1),(2,3),(3,3)\}$
Y_9	$\{(1,3),(2,3),(3,1)\}$	$\{(1,2),(2,3),(3,2)\}$

Figure 2.7: The Dual Perspective of the Social Golfer Problem : Assigning Players to Groups.

2.2 Problems with Domain Representation

Set variables are rich modeling objects which inherently capture the problem semantics in its domain representation. They eliminiate the needs of decomposing a natural object into arrays of finite-domain variables. The problem structure is kept in the model, which allows propagators to make full use of it. Nonetheless, due to the richness of set variables, they suffer a fundamental problem: the potentially exponential domain size.

A variable domain stores all possible values the variable can take. It provides the most usual means of communication between propagators. Propagators remove values from domains and they are invoked one after another. In the finite-domain world, finite-domain variables usually maintain an explicit list of domain values. However, set variables cannot apply the same technique, since it usually contains an exponential number of domain values.

Consider the social golfer instance $(10, 8, 4)$. There are 80 golfers in total, each variable $X_{w,g}$ has a cardinality of 8, which yields $\binom{80}{8} = 28,987,537,150$ different sets of golfers. Even compact bitwise representation will require a few GB of memory for storage per variable domain. Expressing a set domain explicitly, like in the finite-domain world, is computationally expensive. Since the inception of set variables in constraint programming, efforts have been made to achieve an effective and efficient domain approximation or a compact set representation. The remaining of this chapter gives an overview of different attempts of set domain representations and discusses their strength and weakness.

2.3 Subset-Bound Domain

The Subset-Bound representation is the first attempt of approximating a set domain. Puget first discussed it in [53], and Gervet formalized it into a generic framework and introduced basic reduction rules [27, 26, 28]. The subset-bound domain approximates a set domain by maintaining two bounds: the required set r which stores the elements that belongs to *all solutions* and the possible set p which stores the elements that belongs to *some solutions*. Its domain is the set of sets that satisfy these two bounds.

Definition 1 (sb-domain)**.** A subset-bound domain (sb-domain) $sb\langle r, p\rangle$ consists of a required set r, and a possible set p, and represents the set of sets

$$sb\langle r, p\rangle \equiv \{s \mid r \subseteq s \subseteq p\}.$$

Example 1 (sb-domain)**.** The sb-domain $sb\langle\{3\}, \{1, 2, 3, 5\}\rangle$ denotes the set $\{\{3\}, \{1, 3\}, \{2, 3\}, \{3, 5\}, \{1, 2, 3\}, \{1, 3, 5\}, \{2, 3, 5\}, \{1, 2, 3, 5\}\}$. Figure 2.8 illustrates the domain as a lattice under the $\subset$ relation. All 8 sets shown are possible domain values. The required set $r = \{3\}$ is at the bottom, and the possible set $p = \{1, 2, 3, 5\}$ is at the top. All sets in between are domain values. ◇

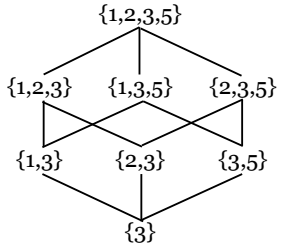

Figure 2.8: The lattice for $X \in sb\langle\{3\}, \{1, 2, 3, 5\}\rangle$

Propagators communicate via variable domains. Propagators are invoked one after another, each attempts to remove values from domains. The notion of consistency is introduced to characterize the strength of propagators. It indicates which values it prunes. Perhaps the most common notion used in the finite-domain CSP is Arc Consistency (AC), or more generally Generalized Arc Consistency

(GAC). Enforcing GAC ensures that every domain value belongs to some solutions. Consistency notions are also defined over the sb-domain. Since it is an approximation, we cannot remove a domain value directly from the domain, a weaker form of consistency is enforced: bound consistency. Informally, sb-bound consistency ensures that all required sets r_{X_i} contains all elements that belongs to all solutions and all possible sets p_{X_i} contains only elements that belongs to some solutions. Conceptually, the two bounds can be obtained in the following way: all solutions of a constraint are enumerated and listed explicitly, If an element belongs to a variable in all solutions, it goes to the required set of that variable. Similarly, if an element does not belong to any solution, it is removed from the possible set.

Definition 2 (sb-bound consistency)**.** A set constraint $C(X_1, ..., X_m)$ (X_i are set variables using the sb-domain) is said to be sb-bound consistent if and only if $\forall 1 \leq i \leq m$,

$$\begin{aligned} & \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \; s.t. \; C(x_1, ..., x_m) \\ \wedge \quad & r_{X_i} = \bigcap_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, .., x_m)} x_i \\ \wedge \quad & p_{X_i} = \bigcup_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, .., x_m)} x_i \end{aligned}$$

where $d(X_i) = sb\langle r_{X_i}, p_{X_i} \rangle$ denotes the domain of X_i.

Example 2. Consider two sb-domain variables $X \in sb\langle \{1\}, \{1, 3, 4\} \rangle$, $Y \in sb\langle \{\}, \{1, 2, 3\} \rangle$, and a binary disjoint constraint $X \cap Y = \emptyset$. The element 1 is in the required set of X, meaning that it belongs to all domain values of X. The disjoint constraint forbids the variable Y from taking values which contain the element 1. To enforce bound consistency, 1 is removed from the possible value of Y, yielding $Y \in sb\langle \{\}, \{2, 3\} \rangle$. Moreover, elements 3 and 4 in X's domain belong to some but not all solution. Hence, none of them belong to the required set. $\diamond$

0/1 Characteristic Function The subset-bound domain has an equivalent representation using a vector of finite-domain variables. This allows the subset-bound domain to seamlessly integrate into finite-domain CP solvers. The key idea is that the subset-bound domain only represents the information of whether an element belongs to the domain. Therefore we can use a 0/1-variable for each element to indicate such information. In particular, the two domain values represent the

element's state in the set variable domain: 0 means the element is excluded while 1 means includes (in the required set). The three states of an element in the subset-bound domain can be represented using a 0/1-variable. All we need is a n-length vector of 0/1-variable, one variable for each element. A characteristic function f maps a 0/1-vector to a set.

Definition 3. A subset-bound domain can be defined as a vector of 0/1-variables $[X_i]$,

$$01sb\langle[l_1, u_1], ..., [l_n, u_n]\rangle \equiv \{f([v_1, ..., v_n]) \mid \forall 1 \leq i \leq n : l_i \leq v_i \leq u_i\}$$

where the characteristic function f is defined as $f([v_1, ..., v_n]) \equiv \{e \mid v_e = 1\}$, and $X_i \in [l_i, u_i]$.

Example 3. Using Example 1. The domain can be represented in the finite-domain world, we have $01sb\langle[0,1],[0,1],[1,1],[0,0],[0,1]\rangle$. Since elements 1,2,5 are possible elements, they can either be included or excluded from the set. Its corresponding 0/1-variable X_i can take either value 0 or 1. Similarly, X_3 has to take 1 since element 3 is required, X_4 has to take 0 since 4 is required. According to the definition, we have a set of characteristic vector $\{[0,0,1,0,0],[1,0,1,0,0],[0,1,1,0,0],[0,0,1,0,1],[1,1,1,0,0],[1,0,1,0,1],[0,1,1,0,1],[1,1,1,0,1]\}$. Each vector corresponds to a set listed in Example 1. ◇

Bound consistency can be defined in the same way. The lower bound of element l_e corresponds to the required set, the element is required when $l_e = 1$, since the e-th position of all vectors has to be 1. Likewise, the upper bound u_e corresponds to the possible set, the element is excluded from the possible set (meaning it cannot belong to any solution) when $u_e = 0$, the e-th position of all vectors has to be 0.

Definition 4 (01sb-bound consistency)**.** A 01set constraint $C(X_1, ..., X_m)$ (X_i is a vector of 0/1-variables $[X_{i,1}, ..., X_{i,n}]$) is said to be bound consistent if and only if $\forall 1 \leq i \leq m$,

$$\begin{aligned}
& \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \;\; s.t. \;\; C(x_1, ..., x_m) \\
\wedge \quad & \forall 1 \leq e \leq n, l_{i,e} = \Big(\bigwedge_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} e \in x_i \Big) \\
\wedge \quad & \forall 1 \leq e \leq n, u_{i,e} = \Big(\bigvee_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} e \in x_i \Big)
\end{aligned}$$

where $01sb\langle[l_{i,1}, u_{i,1}], ..., [l_{i,n}, u_{i,n}]\rangle$ is the 01sb-domain representation of set variable X_i.

Set variable can be represented in the finite-domain world. Accordingly, set constraint propagators can be generated from its high-level declarative specification[64, 63] At this stage, it does not bring any advantage in achieving stronger propagation, as there is an equivalent 0/1-vector representation. Despite of that, this leads to an very interesting observation: *it takes a vector of finite-domain variables to represent a set variable*. That said, *a unary set constraint is a global constraint in the finite-domain world*. The task of achieving complete (bound-consistent) propagators becomes computationally hard, even for unary constraints. We will further discuss this interesting phenomenon in later chapters.

2.4 Subset-Bound and Cardinality Domain

Cardinality constraints for set variables are very common in modeling. In the social golfer problem, the cardinality of all variables are bounded to s, the size of a group. However, for the subset-bound domain, it is hard to take the cardinality information into account. Since swapping two elements in the possible set has no effect on the cardinality. The cardinality constraint can hardly achieve any propagation. It is only used when the size of the possible or required set reaches the cardinality bound, that is the time when the variable is almost bound.

Example 4. Consider $X \in sb\langle\{\}, \{1, ..., 10\}\rangle$, and a cardinality constraint $|X| = 2$. Despite only 55 of the total $2^{10} = 1024$ domain values satisfies the cardinality constraint, neither the possible set nor the required set can be updated in order to achieve a more approximated view of the domain. Consider all 55 possible domain values, which is the set of the enumeration of all pairs, all elements belong to some set and no element belongs to all. No propagation takes place. $\diamond$

The subset-bound domain cannot capture cardinality information well. A terrible consequence is that other constraints cannot take the cardinality constraint into account during propagation. This severely weakens the strength of set variable.

Example 5. Suppose we have two subset-bound variables, $X \in sb\langle\{1\}, \{1, 2, 3, 4\}\rangle$ and $Y \in sb\langle\{\}, \{2, 3, 4, 5\}\rangle$, and three constraints, $|X| = 3$, $|Y| = 3$, and $X \cap Y = \emptyset$. If we consider each of the three constraints separately, they are all bound consistent. However, it is clear that there is no solution since X and Y are disjoint and, according to the cardinality constraint, they require 6

different elements in total. If we were able to take all constraints into account at once, the disjoint constraint would achieve stronger propagation. ◇

Azevedo [2] added a cardinality component to the subset-bound representation. It enables the set domain to take the cardinality information into account during propagation as well as to update the cardinality bounds. We call it the sbc-domain.

Definition 5 (sbc-domain)**.** A subset-bound+cardinality domain (sbc-domain) $sbc\langle r, p, \check{c}, \hat{c}\rangle$ consists of 4 parameters, the required set r, the possible set p, the cardinality lower bound $\check{c}$, and the cardinality upper bound $\hat{c}$. It represents the set of sets

$$sbc\langle r, p, \check{c}, \hat{c}\rangle \equiv \{s \mid r \subseteq s \subseteq p \wedge \check{c} \leq |s| \leq \hat{c}\}$$

Example 6 (sbc-domain)**.** The sbc-domain $sbc\langle\{3\}, \{1, 2, 3, 5\}, 2, 3\rangle$ represents the set { {1,3}, {2,3}, {3,5}, {1,2,3}, {1,3,5}, {2,3,5} }. Figure 2.9 illustrates the sbc-domain. The sets which doesn't satisfy the cardinality (those in gray) constraint are removed from the domain. ◇

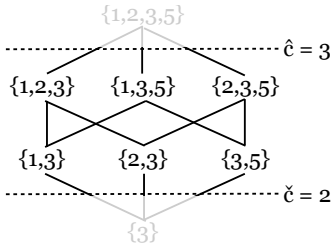

Figure 2.9: The lattice for $X \in sbc\langle\{3\}, \{1, 2, 3, 5\}, 2, 3\rangle$

The bound consistency definition for the sbc-domain builds on the top of the sb-domain, conditions for the cardinality component are added. They require that each cardinality bound is supported by some solutions.

Definition 6 (sbc-bound consistency)**.** A set constraint $\mathcal{C}(X_1, ..., X_m)$ (X_i are set variables using

the sbc-domain) is said to be sbc-bound consistent if and only if $\forall 1 \leq i \leq m$,

$$\exists x_1 \in d(X_1), ..., x_m \in d(X_m) \text{ s.t. } C(x_1, ..., x_m) \tag{2.10}$$

$$\wedge \quad r_{X_i} = \bigcap_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} x_i \tag{2.11}$$

$$\wedge \quad p_{X_i} = \bigcup_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} x_i \tag{2.12}$$

$$\wedge \quad \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \text{ s.t. } \left(|x_i| = c_{\check{X}_i} \wedge C(x_1, ..., x_m)\right) \tag{2.13}$$

$$\wedge \quad \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \text{ s.t. } \left(|x_i| = c_{\hat{X}_i} \wedge C(x_1, ..., x_m)\right) \tag{2.14}$$

where $d(X_i) = sbc\langle r_{X_i}, p_{X_i}, \check{c}_{X_i}, \hat{c}_{X_i} \rangle$ is the subset-bound+cardinality domain.

This definition imposes an interesting technical concern which is unseen in finite-domain variables. The membership information (the required and possible set) and the cardinality restriction not always work orthogonally. When a bound is updated, we may need to update other bounds to achieve bound consistency.

Example 7. Consider a sbc-domain variable $X \in sbc\langle\{1\}, \{1, 2, 3, 4\}, 1, 3\rangle$. Suppose the element 2 is added to the required set, it yields the domain $sbc\langle\{1, 2\}, \{1, 2, 3, 4\}, 1, 3\rangle$. This is not bound consistent since the cardinality lower bound $\check{c} = 1$ finds no support: none of the domain values has size 1, since the required set's cardinality is 2. The cardinality lower bound needed to be 2 in order to achieve bound consistency. Now, we have $X \in sbc\langle\{1, 2\}, \{1, 2, 3, 4\}, 2, 3\rangle$. $\diamond$

There has been a lot of work concerning the use of the cardinality component. Sadler and Gervet gave reduction rules for the pair-wise *atmost*1 global constraint[55], they showed that applying these rules effectively reduces the search space. Bessière, Hebrard, Hnich, and Walsh systematically evaluates the effect of adding the cardinality component to the subset-bound domain in global intersection constraints[6]. It was shown that many global constraints become intractable when the cardinality restriction presents. Van Hoeve and Sabharwal gave a bound-consistent propagator for the binary *atmost*1 constraint over sbc-variables[70]. In [75], Yip, Van Hentenryck, and Gervet evaluated the performance of sbc-domain variables on network deployment problems, a highly symmetrical problem which had been used to evaluate the effectiveness of different symmetry breaking techniques. The paper showed many global constraints become intractable when the cardinality is taken into account.

Domain reduction rules based on cardinality restriction are introduced to boost propagation.

The introduction of the cardinality component is a key milestone in the research on set variables. It dramatically diverges from the finite-domain variables. It encapsulates the cardinality constraint in the domain representation, enabling other constraints to take into account the cardinality information. Bound-consistent propagators are, by definition, required to take such information into consideration and hence stronger propagation is achieved. Bound updates may trigger domain-reduction operations on other bounds, which is not a common phenomenon in the finite-domain world. This further illustrates the essence of the use of set variables: It allows model to keep the problem structure and enables propagators to exploit its semantics.

2.5 Hybrid Domain

We have discussed a few representations whose primarily goal is to capture membership and cardinality information. But, they are unable to directly capture some domain information: ordering constraints. Ordering constraints are very common constraint for eliminating symmetry, which many set-CSPs exhibit.

We demonstrated four kinds of symmetry in the social golfer problem: positions within a group are interchangeable, groups within a week are interchangeable, weeks are interchangeable, and golfers are also interchangeable. The first symmetry type is eliminated by using set variables, the rest can be removed by symmetry-breaking constraints. Lines 2.5, 2.6, and 2.9 are ordering constraints that remove non-canonical solutions. One of the most common ordering constraints is the lexicographical ordering constraint on the 0/1-characteristic vector of the set variable.

Now, it comes to the question of whether or not the set variable domain representation is good enough to capture the propagation result of the ordering constraint. It appears that if the domain cannot capture the ordering information, some propagations may be hindered.

Example 8. Consider a set variable $X \in sbc\langle\{\}, \{1,2,3,4,5\}, 3, 3\rangle$, and two lexicographical ordering constraints $\{1,4,5\} \leq_{lex} X$ and $X \leq_{lex} \{2,3,4\}$. Suppose two constraints are not considered at the same time, the domain of X is bound consistent for both constraints, no propagation is achieved. However, when two constraints are considered at once, that possible solutions for X is $\{1,4,5\}$ and $\{2,3,4\}$, the element 4 belongs to all solutions hence should be included into the required set. $\diamond$

If the domain representation takes the lexicographical information into account, there is a chance of getting stronger propagation. Sadler and Gervet observed this and proposed to add a lexicographical component to the subset-bound+cardinality domain[56, 57]. It is called the *hybrid domain*.

Definition 7 (hybrid domain)**.** A hybrid domain $hybrid\langle r, p, \check{c}, \hat{c}, l, u\rangle$ adds a lexicographic component onto the sbc-domain, it consists of 6 parameters, the required set r, the possible set p, the cardinality lower bound $\check{c}$, the cardinality upper bound $\hat{c}$, the lexicographical lower bound l, and the lexicographical upper bound u. It represents the set of sets

$$hybrid\langle r, p, \check{c}, \hat{c}, l, u\rangle \equiv \{s \mid r \subseteq s \subseteq p \wedge \check{c} \leq |s| \leq \hat{c} \wedge l \leq_\theta s \leq_\theta\}$$

where θ is a total order. In [57], θ is defined as the lexicographical ordering in which the largest element is the most significant position.

2.6 Reduced Ordered Binary Decision Diagram Domain

The subset-bound domain, as well as all its variants, aims at obtaining a precise approximation of the set domain. However, the set domain, by definition, contains an exponential number of values, it is impossible for these bounds, which are polynomial in size, to achieve a precise domain in general.

Hawkings, Lagoon, and Stuckey presented a dramatically orthogonal approach for modeling the set domain which allows efficient domain propagation [33, 34]. They proposed to use *reduced ordered binary decision diagram*(ROBDD) to compactly represent set domains as well as set constraints. Under ROBDD, it is possible to attain an exact, and usually compact, domain representation. It allows the solver to get an exact view on the problem and achieve domain propagation. This is unprecedented approach in set variables. The key idea of the proposal is that the membership information of an element (i.e. $e \in s$) can be neatly encapsulated with a binary decision variable. A boolean formula, which is a conjunction of a set of binary variables, represents a set. A set domain is, accordingly, disjunction of a set of boolean formula. ROBDD is a data-structure for manipulating boolean formulas, its size is usually compact since the binary decision diagram is *reduced* as many of the decision variables can be combined.

We illustrate the basic idea of ROBDD using the example given in [34]. Suppose we have a set

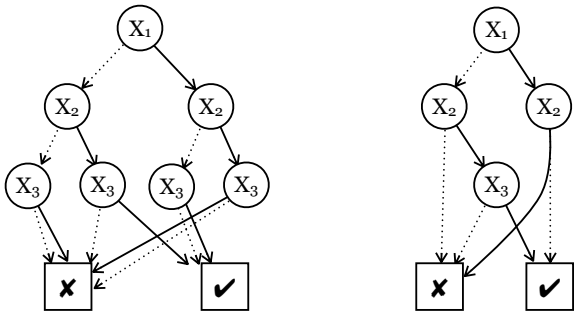

Figure 2.10: ROBDD: $X \in \{\{1\}, \{1,3\}, \{2,3\}\}$. Solid lines correspond to true, dotted lines false. Left: Original; Right: Reduced Ordered BDD.

variable $X \in \{\{1\}, \{1,3\}, \{2,3\}\}$ and would like to represent it using binary decision diagrams. We first discuss how to represent a set using a boolean formula and then the domain as a disjunction of a set of formulas. Similar to the 0/1-vector transformation of the sb-domain, we use a vector of binary variable to represent a set, i.e. $X_e = 1 \Leftrightarrow e \in X$. The set $\{1,3\}$ is, therefore, equivalent to the boolean formula $X_1 \wedge \neg X_2 \wedge X_3$. The set domain is a disjunction of all these formulas. Figure 2.10 (left) illustrates the set domain of X using boolean formulas. Solid lines correspond to $X_e = 1$, whilst dotted lines $X_e = 0$. Paths lead to the ✔ box are allowed domain values and those lead to ✘ are forbidden. Indeed, a lot of internal nodes of the diagram serves no function. For example, in the leftmost X_3, both of the outgoing arcs point to the ✘ box, indicating under that particular path, no matter what value X_3 is, the value is not allowed. The leftmost X_3 node can be eliminated and the dotted outgoing arc of the leftmost X_2 node can point directly to ✘. The *reduced* ordered binary decision diagram data structure provides a mechanism to *reduce* the size of the diagram by eliminating excessive and redundant nodes. Figure 2.10 (right) illustrates the ROBDD of the domain of X. This is the key of ROBDD: with a very compact representation, we know *exactly* which set is in the variable domain. Domain propagation is achievable.

In addition, the ROBDD approach not only allows exact domain representation, but also the flexibility of implementing constraints. ROBDDs also represent constraints. This is from the observation that most constraints can be transformed to boolean formulas. We get a propagator in ROBDD simply by specifying the boolean formula of the constraint based on the membership decision variables. It avoids to laboriously work of implementing constraint-specific propagators. More

	Subset-Bound (and its variants)	ROBDD
Propagation	Loose	Very Precise
Space	$O(n)$	Potentially Exponential
Efficiency	Fast	Potentially Slow
Convergence	Fast	Potentially Slow

Figure 2.11: Comparison over Different Set Domain Representations. (n is the universe size)

generally, primitive ROBDD-based constraints can join together and form a global constraint which allows stronger propagation.

The same paper proposed several amendments and variants to the ROBDD representation. Sometimes domain propagation may be too costly to find a solution and the diagram size may explode, the solver may resolve the problem by restricting its propagation strength to set bound reasoning. On the other hand, inspired by the cardinality and lexicographical components introduced in the subset-bound domains, it was also shown that these bound can be obtained from and propagated in the ROBDD domain trivially. Despite of having a potentially exponential storage size, the ROBDD approach gives very promising results on various standard benchmarks over previous work on the subset-bound domains as well as its variants.

Another interesting extension of the BDD technique is proposed by Gange, Stuckey, and Lagoon. They incorporate a BDD-based set solver with the learning abilities of SAT solvers[35, 24]. Domain reductions are performed by binary decision diagrams. After the BDD propagation, the corresponding clauses are generated and sent to the SAT solver for search and conflict analysis. The hybrid, BDD + SAT, approach gives even more promising results than the ROBDD original paper, and beats the state-of-the-art set solvers on three standard benchmarks.

2.7 Conclusion

A set variable domain may contain an exponential number of values. It is impossible to represent a set domain by explicitly enumerating all domain values. Much effort has been made on either approximating the set domain or making it as compact as possible. We presented the development of the domain representation of set variables over the past 20 years. The subset-bound representation is the first attempt and is a smooth and direct transition from the finite-domain world. A few additional

components has been proposed to strengthen it. It is available in most modern constraint solvers. On the other hand, an exact domain representation was proposed using binary decision diagrams. Moreover, this approach allows global constraints be implemented by specifying its boolean formula, which greatly simplifies the work as well as guarantees correctness. Despite of the potentially exponential space it uses, the approach outperforms the subset-bound domain across a various of benchmarks. Figure 2.11 gives a high level comparison of the two research directions. The subset-bound domain is fast and cheap but imprecise, whilst the ROBDD domain is heavy but accurate. It raises a research question of whether there exists something in between: that is not too heavy but reasonably accurate. We will address the solution in the next section.

Chapter 3

Length-Lex Domain

3.1 Overview

In the last chapter, we reviewed several domain representations. The most common domain representation is the subset-bound domain, whose primarily goal is to capture the membership constraint, i.e., $e \in X$. However, this representation has inherent difficulties in handling cardinality and lexicographic constraints, which are very common and important in modeling. The length-lex representation was proposed to tackle this problem in an orthogonal perspective[29]. While the subset-bound domain stresses on the membership information, the length-lex domain takes a dual perspective and encodes the cardinality and lexicographical information directly. The advantages of using the length-lex representation is fourfold:

1. it features a total ordering, which makes it possible to enforce bound consistency;
2. it directly captures cardinality and lexicographical information which are common constraints for modeling combinatorial problems;
3. it takes linear space;
4. it allows propagators to enforce bound consistency in polynomial time (assuming checking feasibility in the length-lex domain is tractable).

Enforcing bound consistency is essential to efficiently solve a problem. Bounds represent solutions to the constraint as, when the propagator decides that the bound belongs to no solution, a good representation should provide a mechanism for removing the bound.

The subset-bound domain has a notion of bound consistency too but in a weak sense. A bound in the subset-bound domain simply reflects the state of *an element*. It is weaker than the length-lex bound which is a *domain value*. The length-lex representation is proposed to remedy the problem of the weak bound-consistency definition in subset-bound domain. We will further illustrate the difference after formally introducing the length-lex domain.

3.2 The Length-Lex Domain

Notations For simplicity, we assume that sets take their values in a universe $U(n)$ of integers $\{1, \ldots, n\}$ equipped with traditional set operations. n denotes the size of a universe. Elements are denoted by letters e, f, possibly subscripted and modified as $\check{f}$, $\mid f$ and $\hat{f}$ to denote the minimum, mean and maximum value respectively. Sets are denoted by l, u, s, t, w, x, y, z. A subset s of $U(n)$ of cardinality c is called c-set and is denoted as $\{s_1, s_2, \ldots, s_c\}$ where $(s_1 < s_2 < \ldots < s_c)$. The notation $s_{i..j}$ is a shorthand for $\{s_i, s_{i+1}, \ldots, s_j\}$.

Length-Lex Representation The length-lex ordering $\preceq$, proposed in [29], totally orders sets first by cardinality and then lexicographically.

Definition 8 (Length-Lex Ordering)**.** The length-lex ordering is defined by

$$s \preceq t \textbf{ iff } s = \emptyset \vee |s| < |t| \ \vee |s| = |t| \ \wedge (s_1 < t_1 \vee s_1 = t_1 \wedge s \setminus \{s_1\} \preceq t \setminus \{t_1\})$$

Its strict version is defined by $s \prec t$ **iff** $s \preceq t \wedge s \neq t$.

Example 9 (Length-Lex Ordering)**.** Given $U(4) = \{1, \ldots, 4\}$, we have $\emptyset \prec \{1\} \prec \{2\} \prec \{3\} \prec \{4\} \prec \{1,2\} \prec \{1,3\} \prec \{1,4\} \prec \{2,3\} \prec \{2,4\} \prec \{3,4\} \prec \{1,2,3\} \prec \{1,2,4\} \prec \{1,3,4\} \prec \{2,3,4\} \prec \{1,2,3,4\}$. Figure 3.1 depicts the length-lex ordering. $\diamond$

The length-lex ordering is total. The ordering relation $\prec$ is defined between any pair of sets. One way to view this new ordering is that we map every sets to integer. That the empty set $\{\}$ is mapped

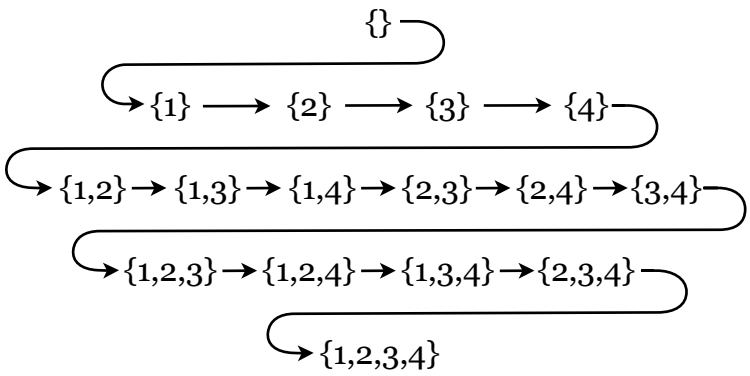

Figure 3.1: The Length-Lex Ordering for $U(4)$.

to 0, the set $\{1\}$ to 1, and on. It makes defining an *interval* of sets possible, which is laid down as the foundation for a domain representation that is capable to achieve bound consistency.

Definition 9 (Length-Lex Domain)**.** The length-lex domain (ll-domain) $ll\langle l, u, n\rangle$ consists of a lower bound l, an upper bound u, and a universe size n. It contains all sets (inclusively) in the universe $U(n)$ between l and u in the length-lex ordering.

$$ll\langle l, u, n\rangle \equiv \{s \subseteq U(n) \mid l \preceq s \preceq u\}$$

Sometimes we also call it a length-lex interval.

Example 10 (Length-Lex Domain)**.** The length-lex domain $ll\langle\{1,3\},\{1,2,4\},4\rangle$ represents the set of sets $\{\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\},\{1,2,3\},\{1,2,4\}\}$. ◇

Because the length-lex ordering defines a total order on sets, it is possible to enforce bound consistency of set constraints. Informally, bound consistency requires both bounds appear in a solution to the constraint, which corresponds to the traditional notion of bound consistency on finite-domain variable.

Definition 10 (ll-Bound Consistency)**.** A set constraint $C(X_1, ..., X_m)$ (X_i are ll-domain set variables) is said to be ll-bound consistent if and only if $\forall 1 \leq i \leq m$,

$$\begin{aligned} & \exists x_1 \in d(X_1), ..., x_m \in d(X_m) : C(x_1, ..., x_{i-1}, l_{X_i}, x_{i+1}, ..., x_m) \\ \wedge \quad & \exists x_1 \in d(X_1), ..., x_m \in d(X_m) : C(x_1, ..., x_{i-1}, u_{X_i}, x_{i+1}, ..., x_m) \end{aligned}$$

where $d(X_i) = ll\langle l_{X_i}, u_{X_i}, n_{X_i}\rangle$ denotes the domain of X_i.

Example 11 (ll-Bound Consistent)**.** Consider the unary constraint $C(X) \equiv 1 \in X$, the ll-domain $ll\langle\{1,3\},\{1,2,4\},4\rangle$ is bound consistent because both the lower bound $\{1,3\}$ and upper bound $\{1,2,4\}$ are solutions to the constraint. $\diamond$

Example 12 (*Not* ll-Bound Consistent)**.** Consider the unary constraint $C(X) \equiv 2 \in X$, the ll-domain $ll\langle\{1,3\},\{1,2,4\},4\rangle$ is *not* bound consistent. Since the lower bound $\{1,3\}$ is not a solution to the constraint. $\diamond$

Enforcing Bound Consistency To enforce bound consistency, we find the first successor of the lower bound (respectively, the first predecessor of the upper bound) which satisfies the constraint. Figure 3.2 illustrates enforcing bound consistency for a primitive unary constraint $C(X) \equiv 2 \in X$, the ll-domain in Example 12 is used. The lower bound $\{1,3\}$ is not a solution to the constraint C. To enforce bound consistency, we need to find a new bound that satisfies the constraint. Recall that every (sound) propagator should *not* remove any solution, since otherwise we are transforming the problem. Therefore, the goal here is to find the first successor of the lower bound $\{1,3\}$ which is a solution to the constraint. The easiest way would be to enumerate all successors, one at a time. The set $\{1,4\}$ is checked against the constraint, and it is not a solution. Then the set $\{2,3\}$ is checked, it is a solution. Hence, the lower bound of the domain is updated to $\{2,3\}$. Two sets, which are not solutions to the constraint, are removed. After enforcing bound consistency, the domain of X becomes $ll\langle\{2,3\},\{1,2,4\},4\rangle$.

Contrary to the subset-bound domain whose primarily goal is to capture the membership information, the length-lex domain takes a dual perspective and focuses on the cardinality and lexicographical information. As a result, not all subset-bound information is preserved in the length-lex domain representation.

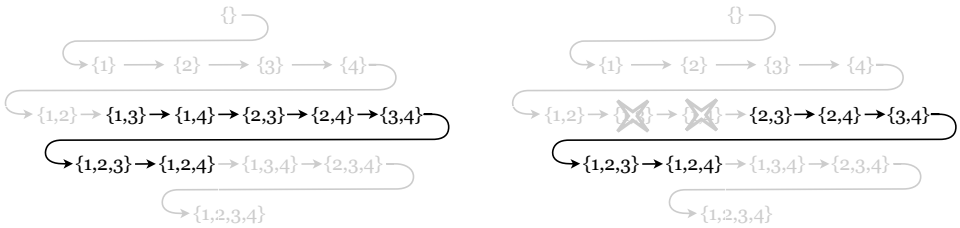

Figure 3.2: $C(X) \equiv 2 \in X$. Left: Original domain $ll\langle\{1,3\},\{1,2,4\},4\rangle$. Right: Domain after enforcing bound consistency $ll\langle\{2,3\},\{1,2,4\},4\rangle$

Example 13 (Length-Lex is an approximation)**.** Consider a length-lex set variable $X \in ll\langle\{1,3\},\{1,2,4\},4\rangle$, and the unary constraint $1 \in X$. It is bound consistent as both bounds are solution to the constraint. Nonetheless, three domains values in the middle of the interval, $\{2,3\},\{2,4\},\{3,4\}$, are not solutions to the constraint. However, under the definition of length-lex, there is no way to remove such inconsistent values. $\diamond$

Another key difference between the length-lex and subset-bound domain is that a bound in length-lex is itself a *solution* to the constraint, whilst a bound in the subset-bound domain only indicates whether *an element* belongs to some (or all) solution. The length-lex bound, in a loose sense, is much stronger than the bound in the subset-bound domain. We illustrate how important it is to have a domain representation whose bounds are solutions.

Example 14 (Why Bound Consistency is Important)**.** Consider a problem consists of a set variable whose domain contains all 3-set in $U(6)$, and two unary intersection constraint $C_1(X) \equiv |X \cap \{1,2,3\}| \leq 1$ and $C_2(X) \equiv |X \cap \{4,5,6\}| \leq 1$. Clearly, there is no solution since X can take at most one element from $\{1,2,3\}$, and one from $\{4,5,6\}$, however X has to take three elements from $\{1,..,6\}$. We demonstrate the difference in propagation between the length-lex and subset-bound+cardinality domain in Figure 3.3. For simplicity, and for the rest of this thesis, we always assume the subset-bound domain has a tight cardinality component. Initially, both domain representations are identical.

At step 1, C_1 is invoked. The lower bound of the ll-domain is not a solution and therefore removed from the domain. Its first feasible successor is $\{1,4,5\}$. On the other hand, consider the subset-bound domain, the set of all feasible domain values for C_1 are $\{1,4,5\},\{1,4,6\},\{1,5,6\}$,

Step	Length-Lex Domain	Subset-Bound Domain
	$X \in ll\langle\{1,2,3\},\{4,5,6\},6\rangle$	$X \in sbc\langle\{\},\{1,2,3,4,5,6\},3\rangle$
1	$C_1 \Downarrow$	$C_1 \Downarrow$
	$X \in ll\langle\mathbf{\{1,4,5\}},\{4,5,6\},6\rangle$	$X \in sbc\langle\{\},\{1,2,3,4,5,6\},3\rangle$
2	$C_2 \Downarrow$	$C_2 \Downarrow$
	$X \in ll\langle\mathbf{\{2,3,4\}},\mathbf{\{2,3,6\}},6\rangle$	$X \in sbc\langle\{\},\{1,2,3,4,5,6\},3\rangle$
3	$C_1 \Downarrow$	
	$\bot$	

$C_1(X) \equiv |X \cap \{1,2,3\}| \leq 1$ and $C_2(X) \equiv |X \cap \{4,5,6\}| \leq 1$.

Figure 3.3: Length-Lex Domain Versus Subset-Bound Domain.

$\{2,4,5\},\{2,4,6\},\{2,5,6\}$, $\{3,4,5\},\{3,4,6\}$, and $\{3,5,6\}$. None of the elements belongs to all solutions nor no solution. Each of them belongs to the possible set and none of them belongs to the required set. According to the bound consistency definition of the subset-bound domain, none of the bounds need to be updated. The subset-bound domain is unchanged. Despite the fact that only 9 of the $\binom{6}{3} = 20$ domain values are solution to the constraint, no propagation can be achieved. There is no way for the subset-domain to capture a *witness* to the constraint, which can be used for inference by other constraints. After step 1 in which the constraint C_1 is propagated, the length-lex domain is updated while the subset-bound domain remain unchanged.

At step 2, a similar constraint C_2 is propagated. The length-lex domain is again updated since both of its bound are not solutions. The subset-bound domain for the same reasons as before is not pruned.

Indeed, after step 2, all domain values of X starts with the prefix $\{2,3\}$ and all of them violates constraint C_1. The domain is emptied and the problem is therefore inconsistent. $\diamond$

This is the key difference between the length-lex and subset-bound domain. The length-lex representation features a total ordering which enables propagators to enforce bound consistency where bounds are solutions to the constraint. The bounds are witnesses and are captured in the domain representation. Domains are the primary mean of communication between constraints. By being

	Subset-Bound (and its variants)	**Length-Lex**	ROBDD
Propagation	Loose	**Strong**	Very Precise
Space	$O(n)$	$\mathbf{O(c)}$	Potentially Exponential
Efficiency	Fast	**?**	Potentially Slow
Convergence	Fast	**?**	Potentially Slow

Figure 3.4: Comparison over Different Set Domain Representations. (n is the universe size, c is the cardinality upper bound.)

able to capture witnesses as well as to prune some of the infeasible values (contrarily to infeasible elements), other constraints can take advantage of the reduced domain and achieve strong propagation. This gives the length-lex domain a huge advantage over the traditional subset-bound domain. Indeed, this leads to some very interesting behavior in the constraint propagation algorithm too. We will further discuss it in details in Chapter 7.

3.3 Conclusion

In this chapter, we introduced the length-lex representation introduced by Gervet and Van Hentenryck [29]. The newly proposed domain representation offers a few theoretical advantages over earlier techniques on set variables: it takes linear space, it achieves bound consistency, and it directly captures cardinality and lexicographical information. We gave an example where the length-lex domain detects inconsistency during constraint propagation while the subset-bound domain prunes nothing. It shows the importance of capturing a witness in the domain representation, which enables constraints to communicate effectively. Moreover, the same paper gave linear time propagators for some primitive unary constraints. The comparison is shown in Figure 3.4.

It is Where Everything Start The thesis is built on the foundation which Gervet and Van Hentenryck laid in [29]. A new representation is proposed, it is shown to offer some theoretical advantages, and very primitive operations are defined. The goal of this thesis is to support the following statement.

To show that length-lex is an effective domain representation for constraint programming over sets.

To support the claim, we attempt to answer all following questions:

1. Are there efficient bound-consistent propagators for other unary and binary constraints?
2. Are there generic ways of implementing bound-consistent propagators for length-lex variables?
3. Since the length-lex domain directly captures lexicographic information, how shall we utilize it with symmetry-breaking constraints?
4. Are there global constraints that exploit the length-lex domain?
5. Is it possible to integrate the length-lex domain with earlier work in constraint programming?
6. What is the empirical performance of the length-lex representation on standard benchmarks?

Chapter 4

Unary Length-Lex Constraints

4.1 Overview

In last chapter, we introduced the length-lex domain representation. Contrarily to the traditional subset-bound domain, it takes an orthogonal perspective and defines a total ordering which primarily captures the cardinality and lexicographic information. We showed that the newly proposed representation has several advantages over the traditional one, it enables stronger propagation and better communication between propagators. Gervet and Van Hentenryck gave specialized bound-consistent algorithms for a few primitive constraints and some domain reduction rules[29]. A few questions still remain open: Does there exist an efficient bound-consistent propagator in general? If so, from a software engineering perspective, is there a generic method to implement it? This chapter addresses these questions. We focus on unary propagators. Techniques generalizing to higher arity propagators are introduced in the next chapter.

To enforce bound consistency, the propagator has to make sure that both the lower and upper bounds of the length-lex domain belongs to a solution. Whenever a bound (say, the lower bound) is infeasible, violating the bound-consistency requirement, a bound-consistent propagator has to update the bound and meet the requirement. The propagator cannot remove any solution from the domain, since otherwise it is transforming the problem and potentially making a consistent problem inconsistent. It has to find the first successor of the lower bound which is a solution to the constraint,

and set this particular successor as the new lower bound. In this way, the propagator is sound and complete as no solution is removed and the bound is a solution to the constraint.

The Key is to Do It Quick A naive approach is to enumerate every successor of a bound and verify if it satisfies the constraint. Implementing such naive approach is trivial and generic, since all we require is to test the constraint. Since set domains usually contain an exponential number of values, naively enumerating everything is computationally expensive. Indeed, it can be avoided by taking into account the key observation that consecutive sets in the length-lex ordering usually share a lot of common properties. By exploiting these properties, a propagator can consider a chunk of sets at a time.

Briefly, the propagator partitions a length-lex interval into a (polynomial) number of chunks of sets. It evaluates one chunk at a time and *locates* the first consistent chunk. It then constructs the smallest feasible value within the chunk using a binary search. This method dramatically improves the propagator efficiency. In particular, a generic propagator implemented in this way takes $O(\alpha c \log n)$ time, where c is the cardinality of the upper bound, n is the universe size, and $O(\alpha)$ is the running time for the constraint-specific checker.

We first present a simple partition method based on the subset-bound domain and we argue that it is possible to use a subset-bound propagator as a black-box to implement a length-lex propagator. Afterwards, we introduce a generic framework for implementing a bound-consistent length-lex propagator. We propose a novel special class of length-lex intervals, PF-intervals, which enjoy some nice closure properties of the subset-bound domain and exploit the strength of length-lex. A generic efficient propagator is then given which only assumes a feasibility checking routine for a PF-interval. Of course, more specialized propagator can be implemented under the above framework, which we will briefly discuss at last.

4.2 Partition into Subset-Bound Lattices

In this section, we illustrate how to partition a length-lex interval into a set of subset-bound lattices from which we can implement length-lex propagators using their subset-bound counterparts as a blackbox.

A subset-bound+cardinality lattice records essentially the required and possible elements and the cardinality restriction. Figure 2.9 shows such a lattice. Some length-lex intervals can be transformed into a subset-bound lattice.

Example 15. A length-lex interval $ll\langle\{1,3,4\},\{1,5,6\},6,3\rangle$ represents the set of sets $\{\{1,3,4\},\{1,3,5\},\{1,3,6\},\{1,4,5\},\{1,4,6\},\{1,5,6\}\}$. The interval is equivalent to a subset-bound lattices $sb\langle\{1\},\{1,3,..,6\},3\rangle$. Notice that the required set is essentially the longest common prefix of all sets. ◇

Reference [13] points out that a length-lex interval can be partitioned into a linear number of sub-intervals which has an equivalent subset-bound lattices.

Example 16 (Partitioning Length-Lex Interval into a Set of Subset-Bound Lattices)**.** A length-lex interval $ll\langle\{1,2,6,7\},\{1,3,6,8\},8\rangle$ can be partitioned into some sub-intervals such that each sub-interval can be denoted as a subset-bound lattice. The decomposition is illustrated in Figure 4.1 and as follows:

$$\begin{aligned}
ll\langle\{\textit{1,2},6,7\},\{\textit{1,2},7,8\},8\rangle &= sbc\langle\{1,2\},\{1,2,6,7,8\},4\rangle \\
ll\langle\{\textit{1,3,4},5\},\{\textit{1,3,4},8\},8\rangle &= sbc\langle\{1,3,4\},\{1,3,4,5,6,7,8\},4\rangle \\
ll\langle\{\textit{1,3,5},6\},\{\textit{1,3,5},8\},8\rangle &= sbc\langle\{1,3,5\},\{1,3,5,6,7,8\},4\rangle \\
ll\langle\{\textit{1,3,6},7\},\{\textit{1,3,6},8\},8\rangle &= sbc\langle\{1,3,6\},\{1,3,6,7,8\},4\rangle
\end{aligned}$$

In the figure, the horizontal line depicts the length-lex interval. Each rectangle corresponds to a partition, which is also a subset-bound lattice. ◇

As a consequence, one can implement bound-consistent propagators for length-lex domains by using subset-bound propagators as a black-box. We give a sketch of this method. For instance, given a subset-bound propagator for the unary constraint $5 \in X$ which is a boolean feasibility checker, we can implement a length-lex propagator. We first decompose a length-lex interval into sub-intervals as demonstrated in Example 16. Then we transform each sub-interval into a subset-bound lattice, and invoke the subset-bound propagator for each lattice. The subset-bound propagator serves as a black-box. Consider the first lattice $sbc\langle\{1,2\},\{1,2,6,7,8\},3\rangle$, there is no solution since none

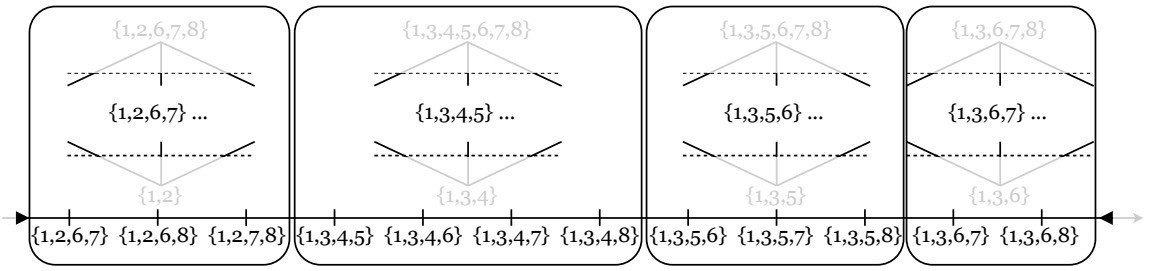

Figure 4.1: Decomposing a Length-Lex Interval into Subset-Bound Lattices

of sets in this lattice satisfy the constraint $5 \in X$. Then chuck of sets in the length-lex interval corresponding to the lattice can therefore be removed. Afterwards, we consider the second lattice $sbc\langle\{1,3,4\},\{1,3,4,5,6,7,8\},3\rangle$, and clearly there is a solution. Such set can be "constructed" by appending elements to the prefix. We enumerate from the smallest free element (elements in the possible set but *not* in the required set), add it to the required set and invoke the black-box. If the black-box indicates there is a solution, we can put it to the required set, otherwise, the element can be excluded.

This method invokes the subset-bound propagator at most $O(cn)$ times where c is the variable's cardinality and n is the size of universe. It demonstrates length-lex propagators can be implemented generically using subset-bound propagators (which is available in most modern constraint programming solver). Nonetheless, it is by no means an effective algorithm. The number of partitions formed is not a polynomial to the cardinality.

Theorem 1. There does not exists a subset-bound decomposition $X^1_{sbc},..,X^{O(c)}_{sbc}$ of the length-lex interval $X_{ll} = ll\langle l,u,n\rangle$ such that $X_{ll} \equiv \cup_{i=1}^{O(c)} X^i_{sbc}$ where $c = |u|$.

Proof. Consider $X_{ll} = ll\langle\{1,2,...,c\},\{n/2,n-c+2,...,n-1,n\},n\rangle$, the interval is equivalent all c-sets begins with elements in range $[1,..,n/2]$, which is

$$\big\{s \mid 1 \le s_1 \le n/2 \wedge s \subseteq U(n) \wedge |s| = c\big\}$$

It implies that some of the subset-bound lattices has be the set of all c-sets with smallest element belongs to a set. Such semantics is not captured by the sbc-domain. □

Despite of the drawback, this method highlights the beauty of decomposition of length-lex interval.

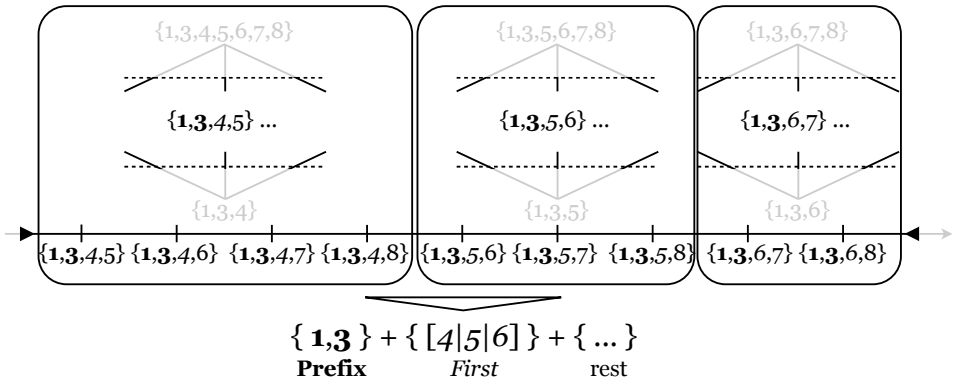

Figure 4.2: The Elegant PF-interval

4.3 Partition into PF-Intervals

This section refines the subset-bound decomposition, making it more efficient by taking advantage of the length-lex semantics. We propose the PF-interval. It is a special class of length-lex interval which enjoys some nice closure properties of a subset-bound lattice. The PF-interval offers two fundamental advantages. First, every length-lex interval can be decomposed into $O(c)$ PF-intervals. Second, reasoning on PF-intervals is as easy as subset-bound lattice, only a minor modification on the subset-bound black box is required. The PF-interval is the core component of many efficient length-lex operations, most bound-consistent propagators presented in this and the next chapter are based on it.

The key observation is that, in the subset-bound decomposition, consecutive subset-bound lattices look alike. Consider the last three lattices in Example 16, which are also shown in Figure 4.2, there are many similarities betweens sets in these lattices. Consider the bold elements $\{1, 3\}$, it is the common prefix of all sets. Immediately following the prefix is an element from the set $\{4, 5, 6\}$, which is in italics. Afterwards, we fill the remaining empty slots in the set by greater elements drawn from the universe.

We conceptualize this notion and define the PF-interval. A PF-interval in a special class of length-lex interval which denotes all c-sets that begin with the same prefix, immediately followed by one element f of a set F (the F-set), and the rest being filled by elements greater than f.

Definition 11 (PF-Interval)**.** Let P be a set and $\check{f}$, $\hat{f}$, n and c be integers. A PF-interval

$pf\langle P, \check{f}, \hat{f}, n, c\rangle$ satisfies

$$(\max(P) < \check{f}) \wedge (\check{f} \leq \hat{f}) \wedge (n - \hat{f} + 1 \geq c - |P|)$$

and denotes the set of sets

$$\Big\{ P \uplus \{f\} \uplus s \Big| \check{f} \leq f \leq \hat{f} \wedge s \subseteq \{f+1, \ldots, n\} \wedge |P \uplus \{f\} \uplus s| = c \Big\}.$$

Theorem 2. PF-interval is a special class of length-lex interval. We have,

$$pf\langle P, \check{f}, \hat{f}, n, c\rangle \equiv ll\langle P \uplus \{\check{f}, \check{f}+1, ..., \check{f}+c'-1\}, P \uplus \{\hat{f}\} \uplus \{n-c'+2, ..., n\}, n\rangle$$

where $c' = c - |P|$ is the cardinality after the prefix.

Example 17 (PF-interval)**.** PF-interval $pf\langle\{1,2\}, 4, 5, 8, 5\rangle$ denotes the set of sets $\big\{\{1,2,4,5,6\}$, $\{1,2,4,5,7\}$, $\{1,2,4,5,8\}$, $\{1,2,4,6,7\}$, $\{1,2,4,6,8\}$, $\{1,2,4,7,8\}$, $\{1,2,5,6,7\}$, $\{1,2,5,6,8\}$, and $\{1,2,5,7,8\}\big\}$.

Example 18 (PF-interval)**.** In Example 16, the union of last 3 lattices contains all 4-sets that begin with $\{1,3\}$, immediately followed by $4, 5$ or 6, and completed by elements in $\{4, .., 8\}$. Therefore, it can be expressed as a PF-interval $pf\langle\{1,3\}, 4, 6, 8, 4\rangle$. It is also shown in Figure 4.2.

The elegance structure of PF-interval enjoys three properties. First, it is very similar to a subset-bound lattices, making its inferences almost identical to subset-bound domains. Second, the F-set is the most significant element after the required prefix, which is the key to capture lexicographical information. Third, any length-lex interval can be decomposed into a linear number of PF-intervals.

However, not all length-lex intervals are PF-interval.

Example 19 (Counter-Example)**.** Consider the length-lex interval $ll\langle\{1,3,5,7\}, \{1,3,6,8\}, 8\rangle$. It cannot be captured by a PF-interval. Since it does not contain all sets with a third element in $\{5,6\}$, in particular, $\{1,3,5,6\}$ is not in the length-lex interval.

We now describe how a length-lex interval can be efficiently partitioned into $O(c)$ number of ordered PF-intervals.

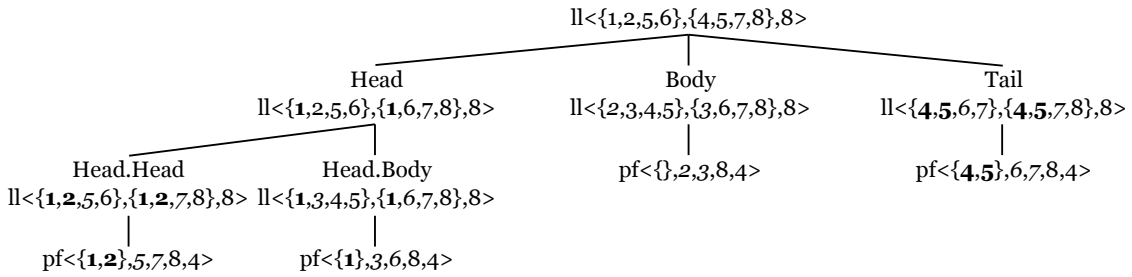

Figure 4.3: Decomposing Length-Lex Interval into PF-Intervals

4.4 The Decomposition

We present an algorithm that takes a length-lex interval and decomposes it and returns an ordered list of PF-intervals. We illustrate the basic intuition with an example.

Example 20 (Decomposition Intuition)**.** Suppose we want to partition a length-lex interval $ll\langle\{1,2,5,6\},\{4,5,7,8\},8\rangle$ into some PF-intervals. Obviously, we cannot use one PF-interval to represent it. We need to partition it into chunks such that some of which satisfy the PF-interval definition. Observe that the given interval contains all 4-sets beginning with either element 2 or 3, as its lower bound beings with element 1 and upper bound element 4. All these sets in the middle can group together and form a PF-interval, $pf\langle\{\},2,3,8,4\rangle$. We call it the *body*.

Now, we have two sub-intervals, the *head* which is before the body and the *tail* which is after. The head and tail are recursively decomposed as illustrated in figure 4.3.

Consider the head interval $ll\langle\{1,2,5,6\},\{1,6,7,8\},8\rangle$. It is not a PF-interval, hence we have to further decompose it. The interval contains a chunk which contains all 4-sets starting with the prefix $\{1\}$ and followed by element 3, 4, 5, or 6. It is a PF-interval $pf\langle\{1\},3,6,8,4\rangle$, or in other words *body of the head*. Now, in the head, all it remains is head of the head interval, $ll\langle\{1,2,5,6\},\{1,2,7,8\},8\rangle$. Indeed, in this case, it is a PF-interval. We reach the base case of the recursion in which the input length-lex interval is a PF-interval. Similarly, the tail is also a PF-interval, and the whole process ends.

After the recursive decomposition process, we partition a length-lex interval into 4 PF-intervals. ◇

The decomposition algorithm is a recursive process. It takes a length-lex interval $ll\langle l, u, n\rangle$ as input. At each stage, it factors out the main body, which is always a PF-interval, and recursively decompose the head and tail. The *head*, if exists, is a length-lex interval containing all c-sets that starts with element l_1, while the *tail* if exists, is also a length-lex interval containing all c-sets that starts with element u_1. The recursive decompositions of the head always produce empty tails, which is critical for the complexity and size of the decomposition. The head always has an upper bound which is the largest c-set begins with l_1 that yields an empty tail in the next stage.

More formally, the decomposition algorithm takes a length-lex interval $ll\langle l, u, n\rangle$ and returns a ordered partition of PF-intervals. Since the decomposition is recursive, the specification needs to include a prefix set P which is initially empty. The algorithm also receives the integer n to represent the universe and uses :: to denote the concatenation of two sequences and ϵ to denote an empty sequence.

Specification 1. Given universe $U(n)$, Algorithm $decompose(l, u, P, n)$ returns an ordered sequence $[X_{pf}^1, \cdots, X_{pf}^w]$ of PF-intervals satisfying

$$\bigcup_{i\in[1,..,w]} X_{pf}^i = ll\langle P \cup l, P \cup u, n\rangle$$

and $\forall i < j \in [1, .., w] : \forall s \in X_{pf}^i, t \in X_{pf}^j : s \prec t$.

Algorithm 1 $decompose(l, u, P, n)$

```
1: c ← |l|
2: H, B, T ← ϵ, ϵ, ϵ
3: h, t ← l_1, u_1
4: if h = t then
5:     return decomp(l_{2..c}, u_{2..c}, P ∪ {h}, n)
6: if l ≠ {l_1, l_1 + 1, .., l_1 + c − 1} then
7:     H ← decomp(l_{2..c}, {n − c + 2, .., n}, P ∪ {l_1}, n)
8:     h ← h + 1
9: if u ≠ {u_1, n − c + 2, .., n} then
10:    T ← decomp({u_1 + 1, .., u_1 + c − 1}, u_{2..c}, P ∪ {u_1}, n)
11:    t ← t − 1
12: if h ≤ t then
13:    B ← [pf⟨P, h, t, n, c + |P|⟩]
14: return H :: B :: T
```

The algorithm is depicted in Algorithm 1. Lines 4–5 factorize the common prefixes. Lines 6–8 create a head if necessary, i.e., if l is not minimal. Lines 9–11 creates a tail if u is not maximal. Line 12–13

create the body if necessary (e.g., $ll\langle\{1,2,5,6\},\{2,5,7,8\},8\rangle$ has no body) and Line 14 returns the partition. These two recursive calls in Lines 7 and 10 increase the size of the prefix. The first in Line 7 now has a maximal second argument compatible with the prefix, while the recursive call in Line 10 has a minimal first argument compatible with its prefix.

Example 21 (The Decomposition)**.** We illustrate the algorithm using Example 20. Given a length-lex interval $ll\langle\{1,2,5,6\},\{4,5,7,8\},8\rangle$, we want to decompose it into a number of PF-intervals. The lower bound is not minimal, hence it has to further decompose the *head* (Line 6–8). Similarly, the upper bound is not maximal, the *tail* has to be further decomposed too (Line 9–11). The *body* ($ll\langle\{2,3,4,5\},\{4,6,7,8\},8\rangle$) is always a PF-interval (Line 12–13). A recursive call is performed on the *head* H. The algorithm sets the prefix to $\{1\}$, obtaining a length-lex interval $ll\langle\{2,5,6\},\{6,7,8\},8\rangle$. Observe that $\{6,7,8\}$ is maximal, so subsequent recursive calls do not generate tails. Once again, we obtain two sub-intervals $ll\langle\{2,5,6\},\{2,7,8\},8\rangle$ and $ll\langle\{3,4,5\},\{6,7,8\},8\rangle$ which when added to the current prefix $\{1\}$ form H' and $B' = pf\langle\{1\},3,6,8,4\rangle$. Since all sets in the first interval begin with 2, the algorithm adds it to the prefix and continues recursively again. Since $ll\langle\{5,6\},\{7,8\},8\rangle$ in addition to prefix $\{1,2\}$ forms the PF-interval $pf\langle\{1,2\},5,7,8,4\rangle$ which is equal to H', there is no head and tail in this call and the algorithm concludes. As a result, $\langle\{1,2,5,6\},\{4,5,7,8\},8\rangle$ is partitioned into

$$\begin{aligned}
\text{Head.Head.Body:} \quad & ll\langle\{1,2,5,6\},\{1,2,7,8\},8\rangle && = pf\langle\{1,2\},5,7,8,4\rangle \\
\text{Head.Body:} \quad & ll\langle\{1,3,4,5\},\{1,6,7,8\},8\rangle && = pf\langle\{1\},3,6,8,4\rangle \\
\text{Body:} \quad & ll\langle\{2,3,4,5\},\{3,6,7,8\},8\rangle && = pf\langle\emptyset,2,3,8,4\rangle \\
\text{Tail.Body:} \quad & ll\langle\{4,5,6,7\},\{4,5,7,8\},8\rangle && = pf\langle\{4,5\},6,7,8,4\rangle
\end{aligned}$$

◇

Theorem 3. Algorithm 1 (*decompose*) partitions a length-lex interval of c-sets into $O(c)$ PF-intervals and takes $O(c^2)$ time.

Proof. After the first call of Algorithm 1, the head H is subsequently decomposed only into heads and bodies (no tails) and the tail is subsequently decomposed only in bodies and tails (no heads). Hence each subsequent call will only make one additional PF-interval and the depth of recursion

can be at most c since the prefix length is incremented in each recursive call. Hence the maximum number of calls and PF-intervals is $2c-1$, which is $O(c)$. For each of those calls, the comparisons in Lines 4 and 7 take $O(c)$ time and the total time complexity is $O(c^2)$. □

The decomposition algorithm partitions a length-lex interval into $O(c)$ PF-intervals. This allows length-lex propagators to consider one PF-interval at a time, leading to efficient filtering algorithm running in time of a function of c and independent of the universe size. It gives a foundation for algorithms presented in later sections.

4.5 Bound Consistency for Unary Constraints

This section introduces a generic algorithm for enforcing bound consistency on basic unary constraints. The algorithm only assumes a feasibility routine hs, which determines if a PF-interval is consistent, making the implementation of bound-consistent propagator effortless. Since the structure of PF-interval is to a great extent similar to a subset-bound lattice, the design of a feasibility routine only requires a slight modification of a naive subset-bound propagator.

We first give an overview of the implementation of a bound-consistent algorithm. We focus our attention on the algorithm for finding the new feasible lower bound, the one for upper bound is essentially equivalent. The algorithm has two phases: first it *locates* the smallest feasible PF-interval; then it *constructs* the smallest feasible value using a binary search.

A bound-consistency algorithm $bc\langle\mathcal{C}\rangle$ on unary constraint $\mathcal{C}$ takes a length-lex domain, and returns a bound-consistent length-lex domain (i.e. both bounds are solution of $\mathcal{C}$) if it is consistent, or return $\perp$ to indicate inconsistency.

Figure 4.4 illustrates the idea. Suppose it is given a length-lex set variable $X \in ll\langle\{1,2,7,8\},\{4,6,7,8\},8\rangle$ and a unary constraint $6 \in X$. It is not bound-consistent since the lower bound $\{1,2,7,8\}$ is not a solution to the constraint. We need to update the lower bound and make it bound consistent. The algorithm removes the first two sets from the domain, which are not solutions, and set the third feasible set $\{1,3,4,6\}$ as the new lower bound. The domain of X now becomes $ll\langle\{1,3,4,6\},\{4,6,7,8\},8\rangle$ and it is consistent.

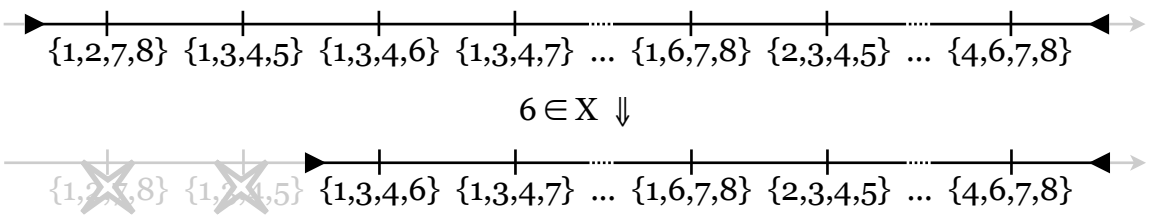

Figure 4.4: Enforcing Bound Consistency For Unary Constraint

Algorithm 2 $\mathrm{bc}\langle\mathcal{C}\rangle(X_{ll} = ll\langle l, u, n\rangle\rangle)$

```
1: l' ← succ⟨C⟩(X_ll)
2: u' ← pred⟨C⟩(X_ll)
3: if l' ≠ ⊥ then
4:     return ll⟨l', u', n⟩
5: else
6:     return ⊥
```

Specification 2 (unary bound-consistent propagator)**.** For a unary constraint $\mathcal{C}$ and a length-lex interval $X_{ll} = ll\langle l, u, n\rangle$, the function $bc\langle\mathcal{C}\rangle(X_{ll})$ returns the a bound-consistent domain $X'_{ll} = ll\langle l', u', n\rangle$ such that

$$\{s \in X_{ll} : \mathcal{C}(s)\} = \{s \in X'_{ll} : \mathcal{C}(s)\} \wedge \mathcal{C}(l') \wedge \mathcal{C}(u')$$

if $\exists s \in X_{ll} : \mathcal{C}(s)$, otherwise returns $\bot$ to indicate inconsistency.

The first criteria is the soundness condition which guarantees that no solution is removed, while the rest is the bound consistency requirement. Algorithm 2 implements Specification 2. It comprises two sub-routines, namely $succ\langle\mathcal{C}\rangle(X_{ll})$ and $pred\langle\mathcal{C}\rangle(X_{ll})$, each of them is responsible for finding the first feasible successor (and predecessor respectively) of the lower (upper) bound that is a solution to $\mathcal{C}$.

Specification 3 (succ)**.** For a unary constraint $\mathcal{C}$ and a length-lex interval X_{ll}, Algorithm $succ\langle\mathcal{C}\rangle(X_{ll})$ returns the smallest (w.r.t. length-lex ordering) supported set $x \in X_{ll}$ in the interval, i.e.

$$\min_{\preceq} x \in X_{ll} : \mathcal{C}(x)$$

if such set exists, or returns $\bot$ otherwise.

and the predecessor algorithm *pred* is similar:

Specification 4 (pred)**.** For a unary constraint $\mathcal{C}$ and a length-lex interval X_{ll}, Algorithm $pred\langle\mathcal{C}\rangle(X_{ll})$ returns the largest (w.r.t. length-lex ordering) supported set $x \in X_{ll}$ in the interval, i.e.

$$\max_{\preceq} x \in X_{ll} : \mathcal{C}(x)$$

if such set exists, and returns $\perp$ otherwise.

Example 22 (Enforcing Bound Consistency)**.** Consider a length-lex set variable $X \in ll\langle\{1,2,3\},\{3,5,6\},6\rangle$ and the unary constraint $\mathcal{R}_5(X) \equiv 5 \in X$. $\{1,2,3\}$ is not a solution and hence the input domain is not bound consistent. The algorithm $bc\langle R_5\rangle(X_{ll})$ returns a bound-consistent domain $ll\langle\{1,2,5\},\{3,5,6\},6\rangle$. $\diamond$

Example 23 (Detecting failure)**.** Consider a length-lex set variable $X \in ll\langle\{2,3,4\},\{3,5,6\},6\rangle$ and unary constraint $\mathcal{R}_1(X) \equiv 1 \in X$. There is no possible successor of $\{2,3,4\}$ of cardinality 3 that could contain element 1. There is no solution and hence $bc\langle\mathcal{R}_1\rangle(X_{ll})$ returns $\perp$ that indicates failure. $\diamond$

We focus our attention on the $succ\langle\mathcal{C}\rangle(X_{ll})$ since the predecessor algorithm operates in a symmetrical manner.

4.6 Generic Successor Algorithm for Length-Lex Interval

We are interested in finding the first feasible successor for an arbitrary unary constraint. The generic successor algorithm returns the smallest solution of the input interval if one exists, otherwise return $\perp$ to indicate inconsistency. The algorithm consists of two phases. In the first phase, it decomposes the given length-lex interval into a set of PF-intervals, and locate the first consistent PF-interval. Then in the second phase, it constructs the smallest feasible set in that PF-interval. We illustrate the first phase in this section and the second in the next.

The algorithm is generic and only assumes a feasibility routine hs which takes a length-lex interval and returns a boolean value indicating if the interval is consistent.

Specification 5 (Feasibility Routine)**.** Given a unary constraint $\mathcal{C}$ and a length-lex interval X_{ll},

Algorithm 3 $succ\langle\mathcal{C}\rangle(X_{ll} = ll\langle l, u, n\rangle)$

1: $[X^1_{pf}, .., X^w_{pf}] \leftarrow decompose(l, u, \emptyset, n)$ {*locate* phase}
2: $X_{pf} \leftarrow \emptyset$
3: **for** $X'_{pf} \leftarrow X^1_{pf}$ to X^w_{pf} **do**
4: **if** $hs\langle\mathcal{C}\rangle(X'_{pf})$ **then**
5: $X_{pf} \leftarrow X'_{pf}$
6: **break**
7: **if** $X_{pf} = \emptyset$ **then** {*construct* phase}
8: **return** $\perp$
9: **return** $succ\langle\mathcal{C}\rangle(X_{pf})$

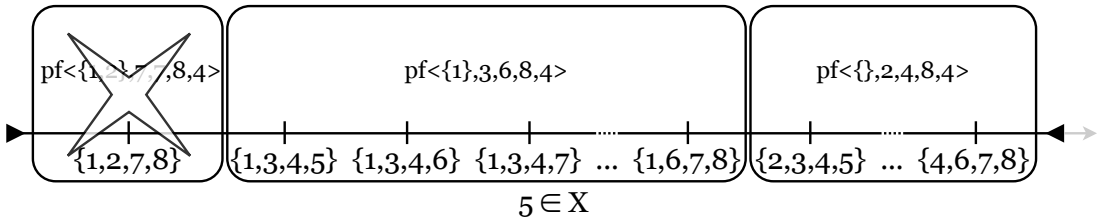

Figure 4.5: Generic Successor Algorithm for Unary Constraint

the feasibility routine returns whether or not the interval has solutions,

$$hs\langle\mathcal{C}\rangle(X_{ll}) \equiv \exists x \in X_{ll} : \mathcal{C}(x).$$

The above feasibility routine can be reduced to one which only decides a PF-interval, instead of a more general length-lex interval. It is due to the fact that every length-lex interval can be decomposed into a set of PF-intervals, the feasibility of a length-lex interval is equivalent to the disjunction of the feasibility of the set of PF-intervals decomposed from it:

$$hs\langle\mathcal{C}\rangle(X_{ll}) \equiv \bigvee_{1 \le i \le w} hs\langle\mathcal{C}\rangle(X^i_{pf})$$

where $X_{ll} = \biguplus_{1 \le i \le w} hs\langle\mathcal{C}\rangle(X^i_{pf})$.

Algorithm 3 (*succ*) is a generic routine for finding the successor that implements Specification 4. Figure 4.5 illustrates the idea. It proceeds in two steps: first, it *locates* the smallest PF-interval that contains a solution(Lines 1–6); second, it *constructs* the smallest solution within the PF-interval,

if any (Lines 7–9). More specifically, Line 1 calls the decomposition routine and obtains the set of PF-intervals. Lines 3–6 iterates over the PF-intervals and locate the first feasible one. Once the PF-interval is located (Lines 4–6), the algorithm invokes a more specialized construction routine, which will be introduced in the next section (Line 9). If the feasibility routine determines that none of the PF-intervals contains a solution, the algorithm returns $\perp$ (Lines 7–8).

Take the length-lex interval $ll\langle\{1,2,7,8\},\{4,6,7,8\},8\rangle$ shown in Figure 4.5 as an example. Suppose we have the unary constraint $5 \in X$. The generic successor algorithm decomposes it into three PF-intervals. They are checked against the feasibility routine one after another. The small one goes first. The first PF-interval, $pf\langle\{1,2\},7,7,8,4\rangle$, is infeasible. All sets within it are skipped. The algorithm then checks the second PF-interval, $pf\langle\{1\},3,6,8,4\rangle$. The feasibility routine returns true, and the algorithm passes this PF-interval to a more specialized algorithm which constructs the smallest feasible set.

Theorem 4. Suppose $hs\langle\mathcal{C}\rangle(X_{pf})$ takes $O(\alpha)$ and $succ\langle\mathcal{C}\rangle(X_{pf})$ takes $O(\beta)$. Algorithm 3 ($succ\langle\mathcal{C}\rangle(X_{ll})$) takes $O(c^2 + c\alpha + \beta)$ time.

Proof. Algorithm *decompose* takes $O(c^2)$. The size of the list of PF-intervals is $O(c)$, hence there are at most $O(c)$ calls to the feasibility routine. Lines 3–8 takes $O(c\alpha)$. Line 9 takes $O(\beta)$. Therefore, the algorithm takes $O(c^2 + c\alpha + \beta)$ time in total. □

4.7 Generic Successor Algorithm for PF-Interval

All it remains is to construct a smallest feasible set within the PF-interval. We give a more fine-grained generic successor algorithm. Likewise, the algorithm only depends on the feasibility routine that takes a PF-interval.

Bisecting a PF-Interval into two The goal is to construct a new lower bound. The algorithm starts with a prefix, appends element to the prefix one at a time, and repeats until a lower bound is constructed. To check whether an element can be appended to the prefix, we resort to the feasibility routine. We create a new interval based on the input interval, and fix the F-set to be a singleton set that only contains the appending element, and invoke the feasibility routine. If the routine returns a positive signal, it indicates that the sub-PF-interval, which contains all sets begin with the

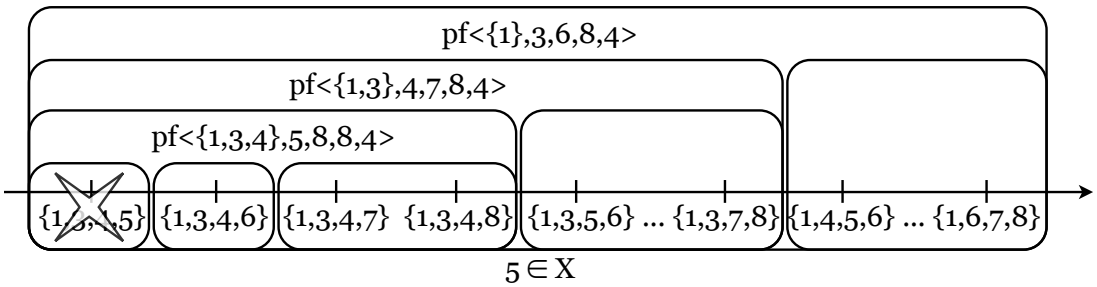

Figure 4.6: Generic Successor Algorithm for Unary Constraint

current prefix plus the appending element, contains a solution. Since the sub-PF-interval is also a PF-interval. We perform the same process for the next appending element. The algorithm iterates until the lower bound is constructed. Since the F-set determines the most significant element that come right after the prefix, picking the smallest element in the F-set guarantees the lower bound is constructed.

Figure 4.6 illustrates the idea. Consider the input PF-interval, $pf\langle\{1\}, 3, 6, 8, 4\rangle$, and a unary constraint $5 \in X$. At each level, we try to append the smallest element in the F-set to the prefix. If the feasibility routine returns true, indicating there is a solution, then we go down another level. Otherwise, the next element in the F-set is tried. In this case, there is a solution, hence we consider a sub-PF-interval, $pf\langle\{1, 3\}, 4, 7, 8, 4\rangle$, in which the element 3 become part of the prefix. We know that the smallest bound begins with $\{1, 3\}$. The algorithm attempts to append the element 4 in the next step, and it succeeds. We go down yet another level, and consider a sub-sub-PF-interval, $pf\langle\{1, 3, 4\}, 5, 8, 8, 4\rangle$. All it remains is to enumerate all element in the F-set to complete the lower bound. Element 5 is first tried and failed, since in such case the lower bound doesn't satisfy the constraint $5 \in X$. The next element in the F-set is considered. The set $\{1, 3, 4, 6\}$ is a solution to the constraint, and new lower bound is found.

The Generic Successor Algorithm Algorithm 4($succ\langle\mathcal{C}\rangle(X_{pf})$) implements Specification 4 over PF-intervals. The routine takes a PF-interval as input. It first checks if the PF-interval is consistent, it returns $\perp$ when we are sure that there is no solution. The routine constructs the new bound as follows. First, it assigns the prefix(Line 3). Then, it iterates over the remaining positions to complete the set (Lines 4–8). At each iteration, we find the minimum F-set element that belongs to a solution

Algorithm 4 $succ\langle\mathcal{C}\rangle(X_{pf} = pf\langle P, \check{f}, \hat{f}, n, c\rangle)$

```
1: if not hs⟨C⟩(X_pf) then
2:     return ⊥
3: s_{1..|P|} ← P
4: l, h ← f̌, f̂
5: for i ← |P| + 1 to c do
6:     s_i ← min_f{l ≤ f ≤ h | hs⟨C⟩(pf⟨s_{1..i−1}, f, n − (c − i), n, c⟩)}
7:     l, h ← s_i + 1, n − (c − i) + 1
8: return s
```

by querying the feasibility routine $hs\langle\mathcal{C}\rangle$. The element is appended to the prefix. The algorithm constructs a new F-set for the next iteration.

By observing that the F-set is always a range, we can perform a binary search on the F-set instead of an explicit enumeration, thus reduce the number of feasibility checks in each position from $O(n)$ to $O(\log n)$. Line 6 defines a binary search when the F-set is not singleton.
We trace algorithm 4 using the following example.

Example 24. Let a unary constraint be $\mathcal{R}_6(X) \equiv 6 \in X$, and a PF-interval $X_{pf} = pf\langle\{1\}, 3, 6, 8, 4\rangle$. Line 1 first checks if the PF-interval contains a solution calling the feasibility routine. It does since 6 is in the F-set. We first assign the prefix $\{1\}$ to s. Then, we need to iterate over the remaining positions (lines 5–7). The F-set is a range from 3 to 6. The algorithm query the feasibility routine to find the smallest F-set element that belongs to a solution. We get element 3 since $hs\langle\mathcal{D}\rangle(pf\langle\{1\}, 3, 3, 8, 4\rangle)$ returns true. We append 3 to s (line 6). We have to prepare for the next position. Observe that $pf\langle\{1\}, 3, 3, 8, 4\rangle = pf\langle\{1, 3\}, 4, 7, 8, 4\rangle$, hence, we can update the F-set accordingly for the next iteration (line 7). In each iteration, one element is appended to the prefix. And finally, the algorithm returns the smallest supported set $\{1, 3, 4, 6\}$. ◇

Complexity Analysis Suppose the feasibility routine for a PF-interval takes $O(\alpha)$ time. It is called both in Algorithm 3 and Algorithm 4.

Lemma 1. Algorithm 4 ($succ\langle\mathcal{C}\rangle(X_{pf})$) takes $O(\alpha c \log n)$.

Proof. Line 1 takes $O(c)$. Lines 3–5 is a for-loop that iterates at most $O(c)$ times. Line 4 is a binary search on the $F - set$ whose maximum cardinality is n, hence there are at most $O(\log n)$ calls to the feasibility routine that takes $O(\alpha)$. Therefore $O(\alpha c \log n)$ in total. □

Algorithm 5 $hs\langle\mathcal{R}_e\rangle(X_{pf} = pf\langle P, \check{f}, \hat{f}, n, c\rangle)$

1: **if** $e \in P$ **then**
2: **return** *true*
3: **if** $\check{f} \leq e \leq \hat{f}$ **then**
4: **return** *true*
5: **if** $c > |P| + 1$ and $\check{f} \leq e \leq n$ **then**
6: **return** *true*
7: **return** *false*

By Theorem 4, and setting $O(\beta) = O(\alpha c \log n)$, the generic algorithm enforces bound consistency on unary constraint in $O(\alpha c \log n)$ time.

4.8 Feasibility Routine for $e \in X$ for PF-Interval

The generic successor algorithm only assumes a constraint specific feasibility routine for PF-interval $hs\langle\mathcal{C}\rangle(X_{pf})$. In this section, we illustrate implementing the feasibility routine for the unary membership constraint $\mathcal{R}_e(X) \equiv (e \in X)$. We hope to provide insight to the design of feasibility routine for other constraints.

Algorithm 5 ($hs\langle\mathcal{R}_e\rangle(X_{pf})$) is the feasibility routine for unary membership constraint that determines whether a set containing e can be found within a given PF-interval. It considers three main cases for a satisfiable set to exist: 1) The element e already belongs to the required prefix (Lines 1–2); 2) the element is a possible first-element (Lines 3–4); 3) the interval cardinality is at least 2 more than the prefix size, and the element is in the remaining possible elements (Lines 5–6). If none of the three cases is possible there is no feasible set.

Theorem 5. $hs\langle\mathcal{R}_e\rangle(X_{pf})$(Algorithm 5) takes $O(c)$.

Therefore, by setting $O(\alpha) = O(c)$, a bound-consistent algorithm that only assumes a feasibility routine runs in $O(c^2 \log n)$ time.

4.9 Conclusion

In this section, we illustrated a generic algorithm for enforcing bound consistency for unary constraint. The algorithm only depends on a boolean feasibility routine which takes a PF-interval as

input. The main idea for the algorithm is that any length-lex interval can be decomposed into $O(c)$ of PF-intervals which have a lot of similarities with a subset-bound lattice. Deciding whether a PF-interval has a solution that is straight forward and hence we can easily locate the smallest feasible PF-interval using a binary search.

The algorithm runs in $O(\alpha c \log n)$ time, where $O(\alpha)$ is the running time for the constraint-specific feasibility routine. This implies that, in a loose sense, when there is a subset-bound black box available, we can implement a bound-consistent length-lex propagator by adding a factor of $O(c \log n)$. Of course, the algorithm can be optimized. Specialized algorithms, which take into the account of the constraint semantics as well as consecutive PF-intervals, can be used in both phases. The total running-time for many basic unary constraints are, indeed, $O(c)$. Readers may refer to Chapter B.1 for detailed descriptions.

Chapter 5

Binary Length-Lex Constraints

5.1 Overview

In last chapter, we illustrated how to implement an efficient, yet generic, bound-consistent propagator for unary length-lex constraint. The key idea of the generic propagator is that length-lex intervals can be decomposed into a linear number of PF-intervals, which captures some nice closure properties of the subset-bound lattice, making inference easy, and allowing the propagator to consider a chunk of domain values at a time. The generic algorithm runs in $O(\alpha c \log n)$ time where $O(\alpha)$ is the time required by the feasibility routine for a PF-interval.

In this chapter, we generalize the generic algorithm into a fixed-arity constraint. We restrict our attention to binary constraints $C(X, Y)$. Algorithms for higher-arity constraints are similar.

Things are Easy When the Other is a PF-Interval We first consider a special case. Suppose that we want to find a new lower bound for variable X and the domain of Y can be represented with *a single PF-interval*. We can view the problem as a *unary constraint* since one of the parameter for the feasibility routine is fixed. In general, we decompose Y into a set of PF-intervals, and deal with each of them one at a time. This adds a factor of $O(c)$ (since it is the number of PF-intervals in the decomposition) to the unary generic algorithm. The binary generic algorithm runs in $O(\alpha c^2 \log n)$.

The chapter is organized as follows. We first give an high level idea of the bound-consistent algorithm. Then we illustrate the generic algorithm for finding the lower bound for length-lex

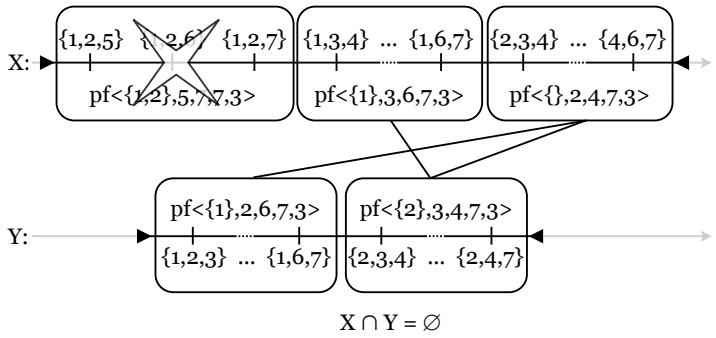

Figure 5.1: Enforcing Bound Consistency for Binary Disjoint Constraint. Solid Lines Indicates Feasible PF-interval Pairs.

intervals. Lastly, we present the feasibility routine for binary disjoint constraint. Other intersection constraints can be obtained in a similar way.

5.2 Bound Consistency for Binary Constraints

We present the generic bound-consistent algorithm for binary constraints. We focus on the successor algorithms since its predecessor counterpart is symmetrical. The algorithm relies on a feasibility routine $hs\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ which determines if the given PF-intervals are feasible.

Example 25. We illustrate the high-level idea using Figure 5.1. Given two length-lex intervals $X \in ll\langle\{1,2,5\},\{4,6,7\},7\rangle$, $Y \in ll\langle\{1,2,3\},\{2,4,7\},7\rangle$, and a binary constraint $X \cap Y = \emptyset$, we want to find a new feasible lower bound for X. Both the length-lex intervals X and Y are decomposed into a set of PF-intervals. First, we locate the smallest feasible PF-interval in X. The task is achieved by checking against every PF-interval in Y. Consider the first PF-interval, $pf\langle\{1,2\},5,7,7,3\rangle$, it is not feasible since both PF-intervals in Y are not compatible with it. The whole chuck of sets with this particular PF-interval are discarded. Then we consider the second chunk, $pf\langle\{1\},3,6,7,3\rangle$, it has support from the PF-interval, $pf\langle\{2\},3,4,7,3\rangle$, indicated by the solid line. Now, we have located the PF-interval, the task remains is to construct the smallest feasible set within the interval. The construction process is similar to that of unary constraint. ◇

It is important to mention that in general, a PF-interval may have support from multiple PF-intervals of the other variable. For example, the third PF-interval in X has support from both PF-intervals

Algorithm 6 $bc\langle\mathcal{C}\rangle(X_{ll} = ll\langle l_X, u_X, n_X\rangle, Y_{ll} = ll\langle l_Y, u_Y, n_Y\rangle)$

1: $l'_X, u'_X \leftarrow succ_X\langle\mathcal{C}\rangle(X_{ll}, Y_{ll}), pred_X\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$
2: $l'_Y, u'_Y \leftarrow succ_Y\langle\mathcal{C}\rangle(X_{ll}, Y_{ll}), pred_Y\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$
3: **if** $l'_X \neq \bot$ **then**
4: **return** $ll\langle l'_X, u'_X, n_X\rangle, ll\langle l'_Y, u'_Y, n_Y\rangle$
5: **else**
6: **return** $\bot$

in Y. Different interval gives different bounds, hence, the algorithm has to take this into account and invoke the construction routine for each PF-interval from the other side.

Formally, the bound-consistent algorithm $bc\langle\mathcal{C}\rangle$ on binary constraint $\mathcal{C}$ takes two length-lex domains, and returns two bound-consistent length-lex domain with regard to $\mathcal{C}$ or returns $\bot$ to indicate inconsistency.

Specification 6 (binary bound-consistent algorithm)**.** For a binary constraint $\mathcal{C}$ and two length-lex intervals $X_{ll} = ll\langle l_X, u_X, n_X\rangle$ and $Y_{ll} = ll\langle l_Y, u_Y, n_Y\rangle$. The function $bc\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$ returns two bound-consistent domain $X_{ll} = ll\langle l'_X, u'_X, n_X\rangle$ and $Y_{ll} = ll\langle l'_Y, u'_Y, n_Y\rangle$ such that

$$\{(x,y)|x \in X_{ll}, y \in Y_{ll} : \mathcal{C}(x,y)\} = \{(x,y)|x \in X'_{ll}, y \in Y'_{ll} : \mathcal{C}(x,y)\}$$
$$\wedge\ \exists y \in Y'_{ll}\ :\ \mathcal{C}(l'_X, y)\ \wedge\ \exists y \in Y'_{ll}\ :\ \mathcal{C}(u'_X, y)$$
$$\wedge\ \exists x \in X'_{ll}\ :\ \mathcal{C}(x, l'_Y)\ \wedge\ \exists x \in X'_{ll}\ :\ \mathcal{C}(x, u'_Y)$$

The first criteria guarantees soundness that no solution is removed, while the others are the definition of bound consistency. Algorithm 6 ($\text{bc}\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$) implements Specification 6. It invokes four different routines, each responsible for finding a supported bound. Line 3 is a domain consistency check, if l'_X is $\bot$ it indicates no value in X_{ll} finds a support from Y_{ll} and therefore it has no solution. Otherwise when l'_X is *not* $\bot$, there exists support from $Y_l l$ hence the algorithm can return bound-consistent domains accordingly.

We give the specification for $succ_X\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$, the routine the returns the smallest set in X_{ll} that has a support from Y_{ll}. The specification for $succ_Y$, $pred_X$ and $pred_Y$ are essentially equivalent.

Specification 7. For a binary constraint $\mathcal{C}$ and two length-lex interval X_{ll} and Y_{ll}. Algorithm

$succ_X\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$ returns the smallest supported set $x \in X_{ll}$, i.e.

$$\min_{\preceq} x \in X_{ll} : \exists y \in Y_{ll}, \mathcal{C}(x, y)$$

if such set exists, or returns $\perp$ otherwise.

We illustrate the generic algorithm with the binary disjoint constraint ($\mathcal{D}(X, Y) \equiv X \cap Y = \emptyset$). We show that it is possible to enforce bound consistency in $O(c^3 \log n)$ time.

5.3 Generic Successor Algorithm for Length-Lex Interval

The generic successor algorithm for binary constraint is similar to the one for unary constraint The *succ* routine only assumes a feasibility routine that decides if there is any solution among two intervals. It first partitions the length-lex interval X_{ll} into some PF-intervals, and *locates* the smallest PF-interval that contains a solution. Then, it *constructs* the smallest feasible set in the PF-interval. To simplify the implementation of the feasibility routine, the algorithm partitions Y_{ll} into PF-intervals, so that the algorithm only requires a feasibility routine that takes two PF-intervals as input. Therefore, the algorithm has to check against every PF-interval in Y_{ll}, which incurs an extra loop.

Algorithm 7 ($succ_X\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$) implements Specification 7 for length-lex intervals. In the locate phase, it first partitions the length-lex interval X_{ll}, Y_{ll} into PF-intervals(Lines 1–2). Then, tries for each PF-interval in X_{ll}, it determines if it is supported by a PF-interval from Y_{ll} partition (Lines 4–8). Once a PF-interval in X_{ll} finds a support from Y_{ll}, the new bound can be constructed within this PF-interval, hence it terminates and proceeds to the construct phase (Lines 7–8). If no support is founded, it returns $\perp$ to indicate inconsistency (Lines 9–10). Otherwise, it invokes another *succ* algorithm that is specialized for PF-interval and returns the lower bound (Line 11). Notice that since X_{pf} can be supported by multiple PF-intervals from Y_{pf}, and each of them possibly support different bound, hence the algorithm has to iterates over all of them and return the smallest supported bound.

We illustrate using the binary disjoint constraint $\mathcal{D}(X, Y) \equiv (X \cap Y = \emptyset)$.

Example 26 ($succ\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$)**.** Consider the binary disjoint constraint over the length-lex intervals $X_{ll} = ll\langle\{1, 2, 5\}, \{4, 6, 7\}, 7\rangle$, $Y_{ll} = ll\langle\{1, 2, 3\}, \{2, 4, 7\}, 7\rangle$. The decompose algorithm yields the

Algorithm 7 $succ_X\langle\mathcal{C}\rangle(X_{ll} = ll\langle l_X, u_X, n_X\rangle, Y_{ll} = ll\langle l_Y, u_Y, n_Y\rangle)$

```
1: [X^1_pf, .., X^i_pf] ← decompose(l_X, u_X, ∅, n_X) {locate phase}
2: [Y^1_pf, .., Y^j_pf] ← decompose(l_Y, l_Y, ∅, n_X)
3: X_pf ← ∅
4: for X'_pf ← X^1_pf to X^i_pf do
5:    for Y'_pf ← Y^1_pf to Y^j_pf do
6:       if hs⟨C⟩(X'_pf, Y'_pf) then
7:          X_pf ← X'_pf
8:          break
9: if X_pf = ∅ then {construct phase}
10:   return ⊥
11: return min_{Y'_pf ∈ [Y^1_pf, ..., Y^j_pf]}(succ_X⟨C⟩(X_pf, Y'_pf))
```

sequences

$$X^1_{pf} = pf\langle\{1,2\}, 5, 7, 7, 3\rangle \tag{5.1}$$

$$X^2_{pf} = pf\langle\{1\}, 3, 6, 7, 3\rangle \tag{5.2}$$

$$X^3_{pf} = pf\langle\{\}, 2, 4, 7, 3\rangle \tag{5.3}$$

and

$$Y^1_{pf} = pf\langle\{\}, 1, 1, 7, 3\rangle \tag{5.4}$$

$$Y^2_{pf} = pf\langle\{2\}, 3, 4, 7, 3\rangle \tag{5.5}$$

In Lines 4–8, the algorithm iterates over all possible choices of X sequentially, and locate the smallest PF-interval that finds a support from Y. The algorithm first considers X^1_{pf}, it is not supported by any intervals in Y_{ll} since it contains elements 1 and 2. Then, it considers X^2_{pf}. The feasibility routine returns *true* for Y^2_{pf}, and it breaks the locate loop and proceed to the construct phase(Lines 6–8). The algorithm then calls another successor algorithm dedicated for PF-interval, and will return $\{1, 3, 4\}$ as the new lower bound. ◇

Theorem 6. Suppose $hs\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$ takes $O(\alpha)$ and $succ\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$ takes $O(\beta)$. Algorithm 7 $(succ\langle\mathcal{C}\rangle(X_{ll}, Y_{ll}))$ takes $O(c^2\alpha + c\beta)$ time.

Proof. By Theorem 3, Lines 1–2 take $O(c^2)$ and the number of PF-intervals in the partitions is $O(c)$.

There will be at most $O(c^2)$ calls to $hs\langle\mathcal{C}\rangle$ in Line 6, hence Lines 4–8 takes $O(c^2\alpha)$. Lines 9–10 takes $O(1)$. Line 11 invokes Algorithm 8 for $O(c)$ times and therefore it takes $O(c^2\alpha + c\beta)$. □

5.4 Generic Successor Algorithm for PF-Intervals

We discuss the generic algorithm that construct the lower bound in a PF-interval. The algorithm takes two PF-intervals as input, and greedily picks the smallest feasible element to fill up all its position, and return the smallest supported set. Algorithm 8 implements Specification 7. Indeed, it is almost equivalent to the same routine for unary constraints.

Algorithm 8 $succ\langle\mathcal{C}\rangle(X_{pf} = pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle, Y_{pf})$

```
1: if not hs⟨C⟩(X_pf, Y_pf) then
2:     return ⊥
3: s_{1..|P_X|} ← P_X
4: l, h = f̌_X, f̂_X
5: for i = |P_X| + 1 to c_X do
6:     s_i ← min_f{ l ≤ f ≤ h | hs⟨C⟩(pf⟨s_{1..i−1}, f, f, n_X, c_X⟩, Y_pf) }
7:     l ← s_i + 1
8:     h ← n_X − c_X + i + 1
9: return s
```

Complexity Analysis We first analysis the running time for the successor algorithm for PF-intervals. Consider that $hs\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ takes time $O(\alpha)$.

Lemma 2. Algorithm 8 ($succ\langle\mathcal{C}\rangle(X_{pf} = pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle, Y_{pf})$) takes $O(\alpha c_X \log n_X)$ time.

Proof. Line 3 takes $O(c_X)$. The for-loop in lines 5–8 iterates at most from 1 to c_X. In each iteration, the search in line 6 uses a binary search, the maximum range of $[l, h]$ is n_X, giving a time complexity of $O(\alpha c_x \log n_X)$. □

For simplicity, we denote $c = \max(c_X, c_Y)$ and $n = \max(n_X, n_Y)$. Applying Theorem 6, by setting $O(\alpha) = O(c)$ and $O(\beta) = O(\alpha c \log n)$, enforcing bound consistency on binary disjoint constraint takes $O(\alpha c^2 \log n)$ time.

Algorithm 9 $hs\langle\mathcal{D}\rangle(X_{pf} = pf\langle P_X, \check{f}_X, \hat{f}_X, n_X, c_X\rangle, Y_f = f\langle\check{f}_Y, \hat{f}_Y, n_Y, c_Y\rangle)$

Require: $n_X = n_Y$
1: $F_X, V_X \leftarrow \{\check{f}_X, ..., \hat{f}_X\}, \{\check{f}_X, ..., n_X\}$
2: $F_Y, V_Y \leftarrow \{\check{f}_Y, ..., \hat{f}_Y\}, \{\check{f}_Y, ..., n_Y\}$
3: $flag \leftarrow$ **true**
4: **if** $\check{f}_Y \geq \check{f}_X$ **then**
5: $\quad flag \leftarrow flag \wedge (|P_X \cup V_X| \geq c_X + c_Y)$
6: $\quad flag \leftarrow flag \wedge (|P_X \cup V_X| > c_X)$
7: $\quad flag \leftarrow flag \wedge (|F_X \cup F_Y| \geq 2)$
8: $\quad flag \leftarrow flag \wedge (|V_Y| = c_Y \Rightarrow \check{f}_X < \check{f}_Y)$
9: **else**
10: $\quad flag \leftarrow flag \wedge (|P_X \cup V_X \cup V_Y| \geq c_X + c_Y)$
11: $\quad flag \leftarrow flag \wedge (|V_Y \setminus P_X| > c_Y)$
12: $\quad F'_Y = F_Y \setminus P_X$
13: $\quad flag \leftarrow flag \wedge (F'_Y \neq \emptyset)$
14: $\quad flag \leftarrow flag \wedge (|F_X \cup F'_Y| \geq 2)$
15: $\quad flag \leftarrow flag \wedge (|P_X \cup V_X| = c_X \Rightarrow \exists f \in F'_Y : f < \check{f}_X)$
16: **return** $flag$

5.5 Feasibility Routine for $X \cap Y = \emptyset$ for PF-Intervals

The generic algorithm enforces bound consistency and only assumes one simple constraint specific feasibility routine. Such routine takes two PF-intervals and return a boolean value indicating whether or not there exists a solution. This section illustrates the feasibility routine for binary disjoint constraint. It gives the basic idea for other binary fixed intersection cardinality constraints (e.g. $|X \cap Y| \leq k$). We assume both PF-intervals have the same universe (i.e. $n_X = n_Y$). We first tackle a more specialized case, that the prefix of one PF-interval is an empty set. Then we show it easily generalize to the complete algorithm.

Algorithm 9 implements the feasibility routine for binary disjoint constraint that takes one PF-interval X_{pf} and one F-interval Y_f (a special case of PF-interval where the prefix is empty, i.e. $f\langle\check{f}, \hat{f}, n, c\rangle \equiv pf\langle\emptyset, \check{f}, \hat{f}, n, c\rangle$).

The basic idea behind the feasibility routine is: it determines whether or not it is possible to construct a pair of sets (x, y) where $x \in X_{pf}$ and $y \in Y_f$ such that x and y are disjoint. Recall sets in a PF-interval are constructed from three different components: 1) the prefix, which is fixed, 2) the element immediately follows the prefix which is drawn from the F-set, and 3) other greater elements used to fulfill the cardinality restriction. For disjoint constraint, we want to construct two

disjoint sets and each of which is constructed in the above way. In particular, y does not take any element from the prefix of X_{pf}, x and y take different F-set elements, and x and y each complete themselves with different elements.

The actual reasoning in Algorithm 9 exploits the property that the possible set of a PF-interval is always a range that starts from the smallest element in the F-set and ends with the universe size. Under this property the union of possible sets of X_{pf} and Y_f is always equals to at least one of them. Lines 4–8 cover the case where X_{pf} has a larger possible set and Lines 10–15 are the case where the one of Y_f is larger. The algorithm maintains a boolean value $flag$ to indicates the feasibility (line 3).

Within each case the conditions relative to the three components that constitute a set within a PF-interval are checked: Lines 5,11 reason about the size of the possible set to ensure there are enough elements to construct both x and y. Lines 6–8, 11–15 reason about the existence of a F-set element for each set. The size of union of both F-set should be at least 2, and the other variable should not be too huge and leave no space for the F-set element.

We illustrate the case analysis with the following example:

Example 27 ($hs\langle\mathcal{D}\rangle(X_{pf}, Y_f)$)**.** It is a case analysis.

- Suppose $X_{pf} = pf\langle\{3\}, 4, 4, 8, 3\rangle, Y_f = f\langle 3, 4, 8, 3\rangle$, there is no solution since Y cannot take 3 as F-set element (part of x's prefix, leaving only 4 for both sets to share as first element. Condition line 14 of Algorithm 9 is violated.

- Suppose $X_{pf} = pf\langle\{2\}, 6, 6, 8, 4\rangle, Y_f = f\langle 6, 8, 8, 1\rangle$, there is no solution since the cardinality restriction of X is so tight that it needs to take every elements in the possible set (i.e. $\{6, 7, 8\}$), leaving no F-set element for Y. The condition in line 6 is violated.

- Suppose $X_{pf} = pf\langle\{1\}, 3, 4, 8, 4\rangle, Y_f = f\langle 3, 6, 8, 4\rangle$, there is no solution since X, Y have to pick 3, 4 different elements from the possible set($\{3, ..., 8\}$) respectively, there are not enough rooms. The condition in line 5 is violated.

- Suppose $X_{pf} = pf\langle\{2, 4\}, 5, 5, 8, 4\rangle, Y_f = f\langle 4, 6, 8, 2\rangle$, there exists solutions since X can pick 5 as its F-set element, Y can pick 6, and the possible is large enough to fulfill the cardinality requirement.

Algorithm 10 $hs\langle\mathcal{D}\rangle(X_{pf} = pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle, Y_{pf})$

Require: $n_X = n_Y$

1: **if** $P_X \cap P_Y \neq \emptyset$ **then**
2: **return** **false**
3: **else if** $P_Y = \emptyset$ **then**
4: **return** $hs\langle\mathcal{D}\rangle(pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle f\langle\check{f_Y}, \hat{f_Y}, n_Y, c_Y\rangle)$
5: **else if** $P_X = \emptyset$ **then**
6: **return** $hs\langle\mathcal{D}\rangle(pf\langle P_Y, \check{f_Y}, \hat{f_Y}, n_Y, c_Y\rangle f\langle\check{f_X}, \hat{f_X}, n_X, c_X\rangle)$
7: **else if** $\max P_X \geq \max P_Y$ **then**
8: $P'_X \leftarrow \{e \in P_X | e > \max P_Y\}$
9: **return** $hs\langle\mathcal{D}\rangle(pf\langle P'_X, \check{f_X}, \hat{f_X}, n_X, c_X - |P_X| + |P'_X|\rangle, f\langle\check{f_Y}, \hat{f_Y}, n_Y, c_Y - |P_Y|)$
10: **else**
11: $P'_Y \leftarrow \{e \in P_Y | e > \max P_X\}$
12: **return** $hs\langle\mathcal{D}\rangle(pf\langle P'_Y, \check{f_Y}, \hat{f_Y}, n_Y, c_Y - |P_Y| + |P'_Y|\rangle, f\langle\check{f_X}, \hat{f_X}, n_X, c_X - |P_X|)$

$\diamond$

Algorithm 10 ($hs\langle\mathcal{D}\rangle(X_{pf}, Y_{pf})$) implements the feasibility routine for binary disjoint constraint that takes two PF-intervals. This is the only routine required by the generic algorithm. This routine essentially factors the prefixes out and reduces to a simpler case which is handled by Algorithm 9 ($hs\langle\mathcal{D}\rangle(X_{pf}, Y_f)$).

The algorithm bases on one key observation: the largest element of two prefixes, if they are both non-empty, are different, and since the prefix, by definition, precedes all other elements, hence we can crop the smaller prefix. Lines 1–2 ensure prefixes are disjoint. Lines 3–6 deal with the simple cases, where at least one of the prefixes is empty. Notice that since the disjoint constraint is symmetrical, we can invoke the feasibility routine by inverse the argument order. Lines 7–12 are case that applies the key observation. As two prefixes are disjoint, we can infer that any element smaller than this maximum element is either in one of the prefixes or can not be taken by the other set. Thus we can crop both prefixes by all elements smaller than this maximum value and restrict our feasibility routine call in a way similar to Lines 2 and 4. For instance, suppose we have $X_{pf} = \langle\{1, 4\}, 6, 7, 8, 3\rangle$, $Y_{pf} = \langle\{3\}, 4, 6, 8, 3\rangle$, 3 is the smaller maximum element in P_Y and P_X, thus any element smaller than 3 will have no impact on the truth value of the feasibility routine and can be factorized. We crop the prefix of X_{pf}, factorize the prefix of Y_{pf}, revise the cardinalities of the sets considered, and apply the reasoning to a simpler feasibility routine, Algorithm 9. In this particular example, we would call Algorithm 9 with the following arguments: $X_{pf} = \langle\{4\}, 6, 7, 8, 2\rangle$ and $Y_{pf} = \langle\{\}, 4, 6, 8, 2\rangle$.

Implementation and Complexity The key factor for an efficient implementation lies in the use of the property of ranges. F_X, V_X, F_Y, V_Y are logical representations of range, they are not explicit enumeration of elements. Since most of the operations on these ranges are union (except Lines 12–14, which require extra bookkeeping), we only need to compute the minimum and maximum values that take $O(1)$ time. Line 12, we do not explicitly construct F'_Y, we just mark the removed elements. There are at most $O(c_X)$ of them, since $|P_X| \leq c_X$. Lines 13–14 can be computed in $O(c_X)$ time.

Lemma 3. Algorithm 9($hs\langle\mathcal{D}\rangle(X_{pf}, Y_f)$) takes $O(c_X)$.

Proof. Lines 1–2 are logical representations since ranges are not explicitly created. Each line takes $O(1)$. Lines 5,6,10,11 involve the prefix, whose length is at most $O(c_X)$. With proper bookkeeping, each line can be done in $O(c_X)$ time. Line 12 creates a range with at most $O(c_X)$ "holes" introduced by the prefix P_X, we can simply mark the "holes" instead of explicitly creating the list. And hence Lines 13–15 take $O(c_X)$ time. The overall time is therefore $O(c_X)$. □

Theorem 7. Algorithm 10 ($hs\langle\mathcal{D}\rangle(X_{pf}, Y_{pf})$) takes $O(c)$ where $c = \max(c_X, c_Y)$.

Proof. In the worst case, we need to factorize the prefixes of X_{pf} and Y_{pf} once, which costs $O(c)$. And Algorithm 9 will be invoked at most once. Hence the total runtime is $O(c)$. □

5.6 Evaluation

Given the generic algorithms presented in this section. We are able to solve the social golfer problem. The goal is to give a comparison between the subset-bound+cardinality and the length-lex domain. We will use the same set of instances thru this paper. The reader will see that, the more advanced modeling technique and propagation component we add to the model, the more efficient it is to solve the problem. This evaluation is by no mean comprehensive. For a detailed comparison between the length-lex domain and earlier attempts on several different benchmarks, please refer to Chapter 11.2.

To give a fair comparison, we implemented all our algorithms on the COMET system. Our experiments are run on a Core2Duo 2.4GHz laptop with 4GB of memory. The symbol × indicates a timeout of 1800 seconds.

```
1 int g=3;
2 int s=3;
3 int w=3;
4 int p = g*s;
5 range Groups = 1..g;
6 range Weeks = 1..w;
7 range Players = 1..p;
8 Solver<CP> cp();
9 LengthLexVar<CP> llx[Weeks,Groups](cp,p,s);
10
11 solve<cp>{
12   forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj){
13     cp.post( disjoint(llx[wi,gi],llx[wi,gj]) );
14     cp.post( llx[wi,gi] <= llx[wi,gj] );
15   }
16
17   forall (wi in Weeks, wj in Weeks, gi in Groups, gj in Groups : wi < wj)
18     cp.post( atmost(llx[wi,gi],llx[wj,gj],1) );
19
20   forall (wi in Weeks, wj in Weeks : wi < wj)
21     cp.post( llx[wi,1] <= llx[wj,1] );
22 }
```

Figure 5.2: COMET Model for Social Golfer Problem using Length-Lex Set Variable

The Length-Lex Model We first give the model using length-lex variables. Figure 5.2 illustrates the model of Figure 2.6 in the COMET language. Lines 1–8 initialize the model and define the parameters used in this model. Line 9 declares the length-lex variables `llx` used in this model. The initial domain is all the s-set from the universe $U(p)$, which corresponds to all possible groups. The disjoint constraint which guarantees groups do not overlap in a week is shown in Line 13. Groups within a week are interchangeable, such symmetry is eliminated by Line 14. Line 18 restricts that each pair of players are not allowed to play more than once. Line 21 breaks the symmetry between weeks.

Figure 5.3 is the search procedure for the social golfer problem. Groups of the first week and the first group of the second week is fixed (Lines 3, 5). Variables are labeled in a week-wise fashion. Within each week, the variable with the shortest prefix is selected first (ties are broken by smaller group index first), and the choice consist in inserting the smallest element first and to exclude it on backtracking. This guarantees the golfers evenly distributed among groups. Lines 6–12 illustrate this simple and elegant search strategy in the COMET language.

The Subset-Bound Model The model for subset-bound is indeed very similar. Figure 5.4 gives the model and the search proceduce. In COMET, a subset-bound variable also takes the cardinality

```
using{
  forall (pi in 1..g*s)
    cp.post(NaiveLLRequires<CP>(llx[1,(pi-1)/s+1],pi));
  forall (si in 1..s)
    cp.post(NaiveLLRequires<CP>(llx[2,1],(si-1)*s+1));
  forall (wi in 2..w)
    while (or(gi in 1..g)(!llx[wi,gi].bound()))
      selectMin(gi in 1..g)(llx[wi,gi].getPrefixLength(), gi){
        int pi = llx[wi,gi].getFirstPossible();
        try<cp> cp.post(NaiveLLRequires<CP>(llx[wi,gi],pi));
        | cp.post(NaiveLLExcludes<CP>(llx[wi,gi],pi));
      }
}
```

Figure 5.3: COMET Search Procedure for Social Golfer Problem using Length-Lex Variables

Domain	Subset-Bound		Length-Lex	
(g,s,w)	Time	Fails	Time	Fails
(3,3,3)	**0.01**	**0**	**0.01**	**0**
(4,3,5)	19.99	289948	**17.87**	**103462**
(4,4,6)	96.55	1241016	**28.21**	**40180**
(5,3,6)	**61.58**	605967	64.3	**177220**
(5,3,7)	x	x	x	x
(5,4,5)	7.38	76013	**2.16**	**3979**
(5,4,6)	x	x	x	x
(5,5,4)	69.93	866005	**1.36**	**311**

Table 5.1: Social Golfer Problem: Subset-Bound Domain vs Length-Lex Domain

information into account. That said, it always maintain a bound-consistent state as a sbc-domain. And for the rest of this thesis, unless otherwise specified, all subset-bound variables use sbc-domain. The initialization phase is identical to that of the length-lex domain hence skipped. The `atmost1` constraint is a bound-consistent propagator for subset-bound variable. Symmetries among groups and weeks are eliminated by enforcing a lexicographical ordering in the variables' 0/1-characteristic vector.

Evaluation Table 5.1 shows the difference in time and size of the search tree. Fastest time and smallest number of fails are bolded. Apparently, the social golfer we gave in the introduction is way too easy for us. Length-lex is generally faster than subset-bound, and has dramatically less fail nodes. For instance $(5, 5, 4)$, it reduces the size of the search tree by more than 50 times. Instances denoted by x cannot be solved within the time limit of 1800 seconds. This shows that, the length-lex set variable, even with the most naive and basic model, performs better than the subset-bound

```
Solver<CP> cp();
var<CP>{set{int}} sbx[Weeks,Groups](cp,Players,s..s);
solve<cp>{
  forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj){
    cp.post( disjoint(sbx[wi,gi],sbx[wi,gj]) );
    cp.post( lexleq( all(pi in Players)(sbx[wi,gj].getRequired(pi)),
                     all(pi in Players)(sbx[wi,gi].getRequired(pi)) ) );
  }

  forall (wi in Weeks, wj in Weeks, gi in Groups, gj in Groups : wi < wj)
    cp.post( atmost1(sbx[wi,gi],sbx[wj,gj]) );

  forall (wi in Weeks, wj in Weeks : wi < wj)
    cp.post( lexleq( all(pi in Players)(sbx[wj,1].getRequired(pi)),
                     all(pi in Players)(sbx[wi,1].getRequired(pi)) ) );
}using{
  forall (pi in 1..g*s) cp.post(requiresValue(sbx[1,(pi-1)/s+1],pi));
  forall (si in 1..s) cp.post(requiresValue(sbx[2,1],(si-1)*s+1));
  forall (wi in 2..w)
    while (or(gi in 1..g)(!sbx[wi,gi].bound()))
      selectMin(gi in 1..g) ( sbx[wi,gi].getRequiredSet().getSize(), gi ){
        selectMin(pi in 1..g*s: !sbx[wi,gi].isExcluded(pi) && !sbx[wi,gi].isRequired(pi) )(pi)
          try<cp> cp.post(requiresValue(sbx[wi,gi],pi));
          | cp.post(excludesValue(sbx[wi,gi],pi));
    }
}
```

Figure 5.4: COMET Model and Search Procedure for Social Golfer Problem using Subset-Bound Set Variable

variable.

5.7 Conclusion

In this section, we presented a generic algorithm for enforcing bound consistency for binary constraint. The idea is to partition the other variable to a set of PF-intervals, and to construct a new bound for each PF-interval. Only a feasibility routine for PF-intervals is assumed. The algorithm runs in $O(\alpha c^2 \log n)$ time where $O(\alpha)$ is the running time for the feasibility routine. In particular, the generic binary disjoint constraint take $O(c^3 \log n)$ time. Of course, there exists more efficient algorithm by implementing specialized locate and construct algorithm, in which case the binary disjoint constraint, as well as other intersection constraints, takes $O(c^2)$ time. Please refer to the appendix for details. Figure **??** illustrates the key difference between domains.

Moreover, we give a preliminary evaluation of the length-lex representation using the social golfer problem. We show that it dramatically reduces the size of the search tree. And it is generally faster

	Subset-Bound (and its variants)	**Length-Lex**	ROBDD
Propagation	Loose	**Strong**	Very Precise
Space	$O(n)$	$\mathbf{O(c)}$	Potentially Exponential
Efficiency	Fast	$\mathbf{O(poly(c))}$	Potentially Slow
Convergence	Fast	**?**	Potentially Slow

Figure 5.5: Comparison over Different Set Domain Representations. (n is the universe size, c is the cardinality upper bound.)

than the subset-bound representation.

Chapter 6

Symmetry Breaking with Length-Lex Variables

6.1 Overview

The length-lex domain representation uses a total ordering which allows the domain representation to directly capture the ordering information. This makes length-lex a perfect vehicle for breaking symmetries. In last chapter, we break symmetries by posting ordering constraints among variables.

This section pushes the idea even further. We discuss some advanced symmetry-breaking techniques which greatly improve the propagation strength of the length-lex domain. Three techniques are proposed,

1. a generic algorithm for pushing length-lex ordering propagator into an arbitrary binary constraint;
2. domain reduction rules for the global alldisjoint constraints and a chain of length-lex symmetry-breaking propagators;
3. breaking value symmetry with dual modeling techniques.

We will show that, with all these techniques available, we are able to solve much larger instances of the social golfer problem.

6.2 Pushing Length-Lex Ordering into Binary Constraints

Combining propagators give at least as strong propagation as its decomposition. In the constraint programming community, a lot of research has been devoted to combining propagators. The global all-different constraint is one of the most prominent examples. Enforcing global arc consistency (GAC) takes polynomial time and is much stronger than the pairwise decomposition of inequality constraints. Indeed, in general, many global constraints are more effective than their decomposition[37, 29, 41]. In particular, [37] proposed an algorithm to enforce lexicographic ordering and sum constraint for two vectors of variables simultaneously. [41] proposed an generic algorithm to push the lexicographic order into a global constraint by invoking $O(m)$ calls to the domain-consistent global constraint propagator, m being the number of variables.

When we consider combining propagators, three questions arise. Is the combined propagator efficient? Is it easy to implement? Is it effective?

We demonstrate an efficient and generic method of pushing a length-lex ordering constraint into an arbitrary binary symmetric constraint C. The algorithm only assumes a feasibility routine of C and adds an $O(\log n)$ overhead to the original generic binary bound-consistent algorithm, yielding a total of $O(\alpha c^2 \log^2 n)$, where $O(\alpha)$ is the running time for the feasibility routine. In addition, we demonstrate how to implement a feasibility routine for the combined propagator, and we show the performance gains on the social golfer problem. This is in contrast to other representations, since some earlier attempts of pushing the symmetry-breaking constraint into other constraint did not improve propagation much.

We give two examples showing the advantages of pushing symmetry-breaking constraints into other constraint for length-lex variables. The first example shows we get more pruning than propagating its decomposition. The second example shows that it is hard for the subset-bound representation to propagate this class of constraints well due it is semantics.

To Combine Or Not To Combine

Example 28. We demonstrate it with a simple example shown in Figure 6.1. We compare the effect between propagating a binary disjoint constraint and a length-lex ordering constraint separately and propagating a binary symmetry-breaking disjoint constraint which considers both at the same time.

Decomposition: $\mathcal{D}$, $\preceq$	Combined: $\mathcal{D}_{\preceq}$
$X \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$ $Y \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$	$X \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$ $Y \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$
$\mathcal{D} \Downarrow$	$\mathcal{D}_{\preceq} \Downarrow$
$X \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$ $Y \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$	$X \in ll\langle\{1,2,3\},\{\mathbf{2},6,7\},7\rangle$ $Y \in ll\langle\mathbf{\{2,3,4\}},\{5,6,7\},7\rangle$
$\preceq \Downarrow$	
$X \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$ $Y \in ll\langle\{1,2,3\},\{5,6,7\},7\rangle$	

$\mathcal{D}(X,Y) \equiv X \cap Y = \emptyset$, $\preceq (X,Y) \equiv X \preceq Y$, and $\mathcal{D}_{\preceq}(X,Y) \equiv \mathcal{D}(X,Y) \wedge \preceq (X,Y)$.

Figure 6.1: Combined Symmetry-Breaking Propagators vs Its Decomposition.

We begin with two length-lex variables having the same initial domain. Left is the decomposition, and right is the combined propagator.

In the decomposition, nothing is pruned for the disjoint constraint since the lower bound of each domain is supported by the upper bound of the other one. For the order constraint, there is no propagation as well since the lower bound of Y is not smaller than the lower bound of X nor the upper bound of X is larger than the upper bound of Y.

However, for the combined propagator $\mathcal{D}_{\preceq}$, two bounds are updated. First consider the upper bound of X. X cannot take any sets begin with the element 3 since due to the ordering constraint, Y has to take sets begin at least with 3. Due to the disjoint constraint, X and Y need to take altogether 6 different elements in the set $\{3, ..., 7\}$, which is impossible. It implies that X cannot begin with any element greater than 2. The upper bound of X is therefore $\{2, 6, 7\}$. The lower bound of Y can be obtained under a similar reasoning. After propagating the combined propagator, X can only begin with either element 1 or 2, and Y cannot begin with element 1. $\diamond$

Subset-Bound vs Length-Lex

Example 29. Figure 6.2 gives a comparison between the two approximated representations of set variables. It demonstrates the key difference between the two approximation approaches. There is no propagation occurred in the subset-bound domain, while the upper bound of the length-lex

Subset-Bound	Length-Lex
$X \in sbc\langle\{\}, \{1, ..., 7\}, 3\rangle$	$X \in ll\langle\{1, 2, 3\}, \{5, 6, 7\}, 7\rangle$
$Y \in sbc\langle\{\}, \{2, ..., 7\}, 3\rangle$	$Y \in ll\langle\{2, 3, 4\}, \{5, 6, 7\}, 7\rangle$
$\mathcal{D}_{\preceq} \Downarrow$	$\mathcal{D}_{\preceq} \Downarrow$
$X \in sbc\langle\{\}, \{1, ..., 7\}, 3\rangle$	$X \in ll\langle\{1, 2, 3\}, \{\mathbf{2}, 6, 7\}, 7\rangle$
$Y \in sbc\langle\{\}, \{2, ..., 7\}, 3\rangle$	$Y \in ll\langle\{2, 3, 4\}, \{5, 6, 7\}, 7\rangle$

$\mathcal{D}(X, Y) \equiv X \cap Y = \emptyset$, $\preceq (X, Y) \equiv X \preceq Y$, and $\mathcal{D}_{\preceq}(X, Y) \equiv \mathcal{D}(X, Y) \wedge \preceq (X, Y)$.

Figure 6.2: Combining Propagator : Subset-Bound vs Length-Lex.

representation is updated.

From the combined constraint, we know that X only begin with element 1 or 2, but not any other. There is no way to capture this kind of information using the subset-bound domain, since it only capture whether or not an element belongs to the solution. On the other hand, the length-lex representation is able to capture this since the two bound essentially denotes the range for the most signification element.

Here we don't claim that the combined propagator is better than the decomposition, nor length-lex is better than subset-bound. All we want to give is just a little ground on which we see a chance of exploiting the semantics and strength of length-lex. ◇

In the following, we show how to implement the combined propagator generically.

6.2.1 Overview

We present a generic bound-consistent algorithm that pushes the length-lex ordering constraint into a symmetrical arbitrary binary constraints [1], only assuming a feasibility routine for the binary constraint over two PF-intervals. That said, once we have implemented a generic binary constraint, we get the combined propagator for free. The generic algorithm bases on the observation that total ordering is transitive, once we deduce that a set s is less than any set in some interval, the condition holds for all sets $t < s$. The algorithm enforces bound consistency in $O(\alpha c^2 \log^n)$ time, which is a $O(\log n)$ overhead to the generic binary constraint.

[1] The restriction to symmetric constraint is natural, since otherwise the symmetry would already be broken by the constraint itself.

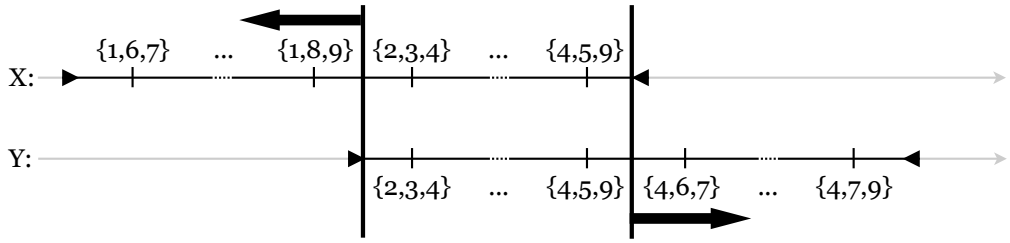

Figure 6.3: Slicing Length-Lex Intervals into 3 Parts

The algorithm bases on two key observations. First, if the greatest set in X_{ll} is smaller than the smallest set in Y, the length-lex ordering constraint is entailed and we can simply apply $bc\langle\mathcal{C}\rangle$. Second, when two PF-intervals are identical ($X_{pf} = Y_{pf}$) and C is symmetric, $hs\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ holds implies $hs\langle\mathcal{C}_{\preceq}\rangle(X_{pf}, Y_{pf})$ also holds. As a consequence of the first property, the algorithm starts by slicing the input length-lex intervals into several pieces, which we first illustrate on an example before defining it formally.

Example 30 (Slicing Two PF-Intervals)**.** Consider the length-lex intervals $X \in ll\langle\{1,6,7\},\{4,5,9\},9\rangle$ and $Y \in ll\langle\{2,3,4\},\{4,7,9\},9\rangle$. The two intervals can be slices into three sub-intervals, as shown in Figure 6.3. The first sub-interval is $ll\langle\{1,6,7\},\{1,8,9\},9\rangle$, all sets in this sub-interval are smaller than any in the domain of Y. Similarly, the third sub-interval is $ll\langle\{4,6,7\},\{4,7,9\},9\rangle$, all sets in this sub-interval are greater than any in the domain of X. Suppose we can find a new bound within either of these sub-intervals, it can be sure that the ordering constraint is satisfied. ◇

Now we formally specify the slicing algorithm.

Specification 8 (3Slices)**.** Given two length-lex intervals X_{ll} and $Y_{ll} = ll\langle l_Y, u_Y, n_Y\rangle$. The function $3Slices(X_{ll}, Y_{ll})$ slices the domain of X with respect to that of Y and returns the three intervals $\check{X}_{ll}, \dot{X}_{ll}$, and $\hat{X}_{ll}$ such that

$$\check{X}_{ll} \equiv \{x \in X_{ll} | x \prec l_Y\} \text{ and } \dot{X}_{ll} \equiv \{x \in X_{ll} | x \in Y_{ll}\} \text{ and } \hat{X}_{ll} \equiv \{x \in X_{ll} | u_Y \prec x\}.$$

Figure 6.4 shows the slices. It only remains to take care of the sub-interval in the middle. What is interesting is that the sub-interval for X and the one for Y are identical. They have the same

$X_{ll} : ll\langle\{1,6,7\},\{4,5,8\},9\rangle$	$Y_{ll} : ll\langle\{2,3,4\},\{4,7,9\},9\rangle$

$\check{X}_{ll} : ll\langle\{1,6,7\},\{1,8,9\},9\rangle$	$\check{Y}_{ll} : \emptyset$
$\dot{X}_{ll} : ll\langle\{2,3,4\},\{4,5,9\},9\rangle$	$\dot{Y}_{ll} : ll\langle\{2,3,4\},\{4,5,9\},9\rangle$
$\hat{X}_{ll} : \emptyset$	$\hat{Y}_{ll} : ll\langle\{4,6,7\},\{4,7,9\},9\rangle$

$\check{X}^1_{pf} : pf\langle\{1\},6,8,9,3\rangle$	:
$\dot{X}^1_{pf} : pf\langle\{\},2,3,9,3\rangle$	$\dot{Y}^1_{pf} : pf\langle\{\},2,3,9,3\rangle$
$\dot{X}^2_{pf} : pf\langle\{4,5\},6,9,9,3\rangle$	$\dot{Y}^2_{pf} : pf\langle\{4,5\},6,9,9,3\rangle$
:	$\hat{Y}^1_{pf} : pf\langle\{4\},6,7,9,3\rangle$

Figure 6.4: Original (Top), Slicing (Middle) and Slicing with PF-Decomposition (Bottom) of Length-Lex Domains.

PF-interval decomposition.

Lemma 4. Given two length-lex intervals X_{ll} and Y_{ll}. $3Slices(X_{ll}, Y_{ll}) = \check{X}_{ll}, \dot{X}_{ll}, \hat{X}_{ll}$ and $3Slices(Y_{ll}, X_{ll}) = \check{Y}_{ll}, \dot{Y}_{ll}, \hat{Y}_{ll}$, we have $\dot{X}_{ll} = \dot{Y}_{ll}$.

Proof. $\dot{X}_{ll} \equiv \{s \in X_{ll} | s \in Y_{ll}\} \equiv \{s \in Y_{ll} | s \in X_{ll}\} \equiv \dot{Y}_{ll}$. □

We can reuse the generic algorithm for binary constraints. Moreover, as the constraint is symmetric, the feasibility routine of the combined constraint reduces to the original one. The new bound can be constructed using a binary search.

Definition 12 (Symmetric Constraint)**.** A binary constraint C over two set variables X and Y is *symmetric* if and only if $C(X, Y) \Leftrightarrow C(Y, X)$.

Lemma 5. If a binary constraint C is symmetric, $hs\langle C\rangle(X_{ll}, X_{ll}) = hs\langle C_{\preceq}\rangle(X_{ll}, X_{ll})$.

Proof. Suppose $C(s, t)$ holds for $s, t \in X_{ll}$. By symmetry, $C(t, s)$ holds and $s \preceq t \vee t \preceq s$ also holds. □

6.2.2 Generic Successor Algorithm for Length-Lex Intervals

We present the generic successor algorithm for finding the new feasible lower bound for Y in $C_{\preceq}$. The main idea of the algorithm is shown in Figure 6.6. Two sub-intervals, the one overlaps with X_{ll} and the one doesn't, are considered. The one, we call it $\dot{Y}_{ll}$, which overlaps with X_{ll} is first considered, since sets in this interval is smaller.. We need to pay special attention since they are

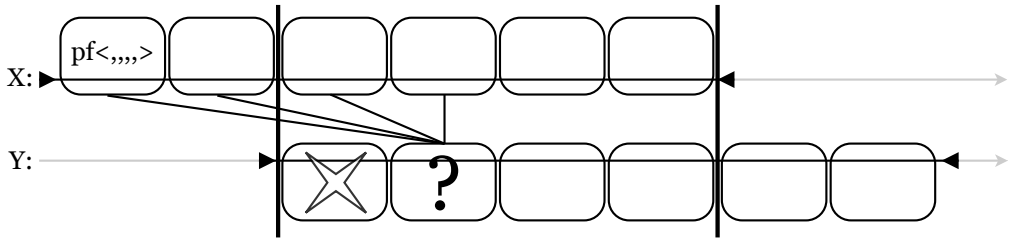

Figure 6.5: Generic Successor Algorithm for $C_{\preceq}$. Solid Lines between PF-intervals Illustrates Feasible Pair Regarding the Ordering Constraint.

overlapping. The algorithm first decomposes $\dot{Y}_{ll}$ into a list of PF-intervals. They are shown as boxes in figure 6.6. We consider one PF-interval at a time, checking it against all PF-intervals in X which is smaller or equals to it. In the figure, for instance, it is determined that the leftmost PF-interval is infeasible. The algorithm then consider the second one, denoted by the symbol ?. Supports come from four PF-intervals which are not greater than itself. We apply the same idea used in the generic binary algorithm, a bound is constructed against each PF-interval on the other side, and the smallest become the bound.

For the first three starting from the left, it is guaranteed that the ordering constraint is satisfied by our slicing construction. Checking feasibility and bound construction reduce to the original binary propagator. Only special care needed to be take for the identical PF-interval. We show that the checking is reducible to the original one. The remaining problem is the construction, which will be discussed into details in the next sub-section.

Algorithm 11 finds the new lower bound for Y. It locates the smallest PF-interval and the construct the bound within it.

Algorithm 11 first slices the length-lex intervals (Lines 1–2) and decomposes the results into PF-intervals (Lines 3–5). $\hat{X}_{ll}$ and $\check{Y}_{ll}$ are not considered as they violate the ordering constraint. Notice that since $\dot{X}_{ll}$ and $\dot{Y}_{ll}$ are identical, their PF-interval decomposition are also identical (e.g. $\dot{X}^i_{pf} \equiv \dot{Y}^i_{pf}$). The algorithm *locates* the smallest PF-interval that contains the new upper bound (Lines 7–11). The process is similar to $bc\langle\mathcal{C}\rangle$, except that it must pay some attention to the ordering constraint. The intuition is captured in Figure 6.6.

The loop starts with the smallest PF-interval in $\dot{Y}_{ll}$ (Line 6) since we want to locate the smallest successor. When considering PF-intervals $\dot{Y}^i_{pf}$, we only consider the PF-intervals in Y_{ll} no larger

Algorithm 11 $succ_Y\langle\mathcal{C}_{\preceq}\rangle(X_{ll} = ll\langle l_X, u_X, n\rangle, Y_{ll} = ll\langle l_Y, u_Y, n\rangle)$

1: $\check{X}_{ll}, \dot{X}_{ll}, \hat{X}_{ll} \leftarrow 3Slices(X_{ll}, Y_{ll})$ {*locate* phase}
2: $\check{Y}_{ll}, \dot{Y}_{ll}, \hat{Y}_{ll} \leftarrow 3Slices(Y_{ll}, X_{ll})$
3: $[\check{X}^1_{pf}, .., \check{X}^{\check{o}}_{pf}] \leftarrow decomp(\check{X}_{ll})$
4: $[\dot{X}^1_{pf}, .., \dot{X}^{\dot{o}}_{pf}] \leftarrow decomp(\dot{X}_{ll})$
5: $[\dot{Y}^1_{pf}, .., \dot{Y}^{\dot{o}}_{pf}] \leftarrow decomp(\dot{Y}_{ll})$
6: $Y_{pf} \leftarrow \emptyset$
7: **for** $i \leftarrow 1$ to $\dot{o}$ **do**
8: $\quad\mathcal{X}' \leftarrow [\check{X}^1_{pf}, .., \check{X}^{\hat{o}}_{pf}, \dot{X}^1, ..., \dot{X}^{i-1}_{pf}]$
9: $\quad$**if** $hs\langle\mathcal{C}_{\preceq}\rangle(\dot{Y}^i_{pf}, \dot{Y}^i_{pf})$ **or** $\bigvee_{X_{pf}\in\mathcal{X}'} hs\langle\mathcal{C}\rangle(\dot{X}^i_{pf}, \dot{Y}^i_{pf})$ **then**
10: $\quad\quad Y_{pf}, \mathcal{X} \leftarrow \dot{Y}^i_{pf}, \mathcal{X}'$
11: $\quad\quad$**break**
12: **if** $Y_{pf} = \emptyset$ **then** {*construct* phase}
13: $\quad$**return** $succ_Y\langle\mathcal{C}\rangle(\check{X}_{ll}, \dot{Y}_{ll} \uplus \hat{Y}_{ll})$
14: **return** $\min(succ_Y\langle\mathcal{C}_{\preceq}\rangle(Y_{pf}, Y_{pf}), \min_{X_{pf}\in\mathcal{X}}(succ_Y\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})))$

than $\dot{Y}^i_{pf}$. Two cases must be distinguished. First, there is exactly one PF-interval in Y_{ll} identical to $\dot{X}^i_{pf}$ and the algorithm must call $hs\langle\mathcal{C}_{\preceq}\rangle$ to take into account the ordering constraint (first condition in line 9 and the vertical line in Figure **??** (Right)). Second, the remain PF-intervals are greater than $\dot{Y}^i_{pf}$ (line 8) and the algorithm simply calls $hs\langle\mathcal{C}\rangle$ (second condition in Line 9 and the diagonal linese in Figure **??** (Left)). If the condition in line 9 holds, the current PF-interval $\dot{Y}^i_{pf}$ has support, the algorithm then breaks the loop and proceeds to *construct* a bound within the PF-interval (Lines 10–11). The algorithm returns the smallest set that has a support, notice that it has to distinguish between two cases as in the *locate* phase(Line 13). If no PF-interval in $\dot{X}_{ll}$ contain a new upper bound, we will try $\hat{Y}_{ll}$. As all sets in $\hat{Y}_{ll}$ are larger than X, we use the simple successor algorithm $succ_Y\langle\mathcal{C}\rangle$ (Lines 12–13).

Example 31. We use Figure 6.4. Consider a binary disjoint constraint $X \cap Y = \emptyset$. We want to push the length-lex ordering constraint ($X \preceq Y$) into it. $\dot{Y}^1_{pf}$ is the first consider interval since it is the smallest. It is compared against $\check{X}^1_{pf}$ and $\dot{X}^1_{pf}$ (it is unnecessary to compare $\dot{X}^2_{pf}$ since it violates the ordering constraint). The feasibility routine reveals that both PF-intervals in X has a support for some values in $\dot{Y}^1_{pf}$, it remains to construct the smallest supported value in this PF-interval. As up to this point the algorithm has no idea which PF-interval in X gives a support for the smallest successor in Y, the algorithm has to construct all of them and return the smallest one (Line 14). $\diamond$

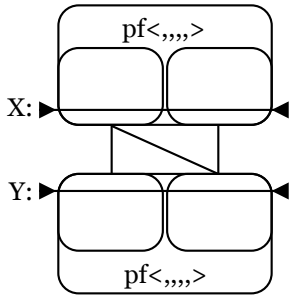

Figure 6.6: Generic Successor Algorithm for PF-Intervals for $C_{\preceq}$

6.2.3 Generic Successor Algorithm for PF-Intervals

It remains to show how to find the successor when two PF-intervals are identical. The key idea is illustrated in Figure 6.6. A PF-interval is cut into two halves until it becomes a singleton. We first try to construct a new bound from the left half. It can only be supported by the left half from X, which is identical. All we need is to recursively calling the algorithm itself with half of the PF-interval. If we get a bound, we win. In the other case, where the left PF-interval is not a solution, we construct a new lower bound from the right half. Supports come from either left or right. The algorithm needs to try both, and return the minimum. For the left one, it reduces to the original binary algorithm since left is smaller than right. For the right one, a recursive call is invoked.

Algorithm 12 constructs the lower bound by performing a binary search in the F-set$\dot{\text{I}}$t partitions the F-set into two halves (Line 9) and checks if there is a solution in the lower half (Line 11 and the right vertical line in the figure). Notice that by Lemma 5, we can simply call $hs\langle\mathcal{C}\rangle$. This lower half needs to be compared only with the lower half $\hat{X}_{pf}$ of X_{pf}, since the upper half $\check{X}_{pf}$ violates the ordering constraint. If there is a solution (Line 10), the algorithm is called recursively within $\check{Y}_{pf}$ (Line 11). Otherwise, the algorithm tries the upper half $\hat{Y}_{pf}$. Now there are two possible choices ($\check{X}_{pf}$ and $\hat{X}_{pf}$) and we do not know which one gives a smaller bound. The algorithm tries both (Lines 13–14) and returns the smallest (Line 15). Once the F-set becomes a singleton, it is inserted in the prefix and the algorithm considers the next position (Lines 3–8).

Example 32. Following Example 31. Consider $succ_Y\langle\mathcal{C}_{\preceq}\rangle(\dot{X}^1_{pf}, \dot{Y}^1_{pf})$, where $\dot{X}^1_{pf} = \dot{Y}^1_{pf} =$

Algorithm 12 $succ_Y\langle\mathcal{C}_{\preceq}\rangle(X_{pf} = pf\langle P, \check{f}, \hat{f}, n, c\rangle, Y_{pf} = pf\langle P, \check{f}, \hat{f}, n, c\rangle)$

Require: $X_{pf} == Y_{pf}$
1: **if** not $hs\langle\mathcal{C}_{\preceq}\rangle(X_{pf}, Y_{pf})$ **then**
2: **return** $\perp$
3: **if** $\check{f} = \hat{f}$ **then**
4: **if** $|P| = c - 1$ **then**
5: **return** $P \uplus \{\check{f}\}$
6: **else**
7: $P', \check{f}', \hat{f}' \leftarrow P \uplus \{\check{f}\}, \check{f} + 1, n - c - |P|$
8: **return** $pred_X\langle\mathcal{C}_{\preceq}\rangle(pf\langle P', \check{f}', \hat{f}', n, c\rangle, pf\langle P', \check{f}', \hat{f}', n, c\rangle)$
9: $\dot{f} \leftarrow (\check{f} + \hat{f})/2$
10: **if** $hs\langle\mathcal{C}\rangle(pf\langle P, \check{f}, \dot{f}, n, c\rangle, pf\langle P, \check{f}, \dot{f}, n, c\rangle)$ **then**
11: **return** $succ_X\langle\mathcal{C}_{\preceq}\rangle(pf\langle P, \check{f}, \dot{f}, n, c\rangle, pf\langle P, \check{f}, \dot{f}, n, c\rangle)$
12: **else**
13: $s_0 = succ_X\langle\mathcal{C}\rangle(pf\langle P, \check{f}, \dot{f}, n, c\rangle, pf\langle P, \dot{f} + 1, \hat{f}, n, c\rangle)$
14: $s_1 = succ_X\langle\mathcal{C}_{\preceq}\rangle(pf\langle P, \dot{f} + 1, \hat{f}, n, c\rangle, pf\langle P, \dot{f} + 1, \hat{f}, n, c\rangle))$
15: **return** $\min(s_0, s_1)$

$pf\langle\{\}, 2, 3, 9, 3\rangle$. The algorithm performs a binary search in the F-set (Lines 9–15), it first checks whether there is a solution in the lower half $pf\langle\{\}, 2, 2, 9, 3$. Clearly, there is no solution for the disjoint constraint as the F-set is a singleton. It implies the new lower bound is in $pf\langle\{\}, 3, 3, 9, 3\rangle$. Its support can come from either $pf\langle\{\}, 2, 2, 9, 3\rangle$ or $pf\langle\{\}, 3, 3, 9, 3\rangle$, hence we need to get both and return the minimum (Lines 13–15). ◇

Complexity Analysis

Lemma 6. Suppose $hs\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ takes $O(\alpha)$. Algorithm 12 ($succ_Y\langle\mathcal{C}_{\preceq}\rangle(X_{pf}, Y_{pf})$) takes $O(\alpha c^2 \log^2 n)$.

Proof. The running time of Algorithm 12 is affected by the number of unfixed positions (maximum is c), the number of possible choices r in the F-set (i.e., $\hat{f} - \check{f} + 1$) and the universe size n. Denote the computation time of the algorithm by $T_n(r, c)$. $T_n(r, c)$ can be expressed by the following recurrence relation:

$$T_n(r,c) = \begin{cases} O(\alpha) + \max(T_n(r/2, c), T_n(r/2, c) + O(\alpha c \log n)) & \text{if } r > 1 \\ T_n(n, c-1) & \text{if } r = 1 \wedge c > 1 \\ O(1) & \text{otherwise} \end{cases}$$

Solving the recurrence relation gives $T_n(r,c) = O(\alpha c^2 \log^2 n)$. □

Theorem 8. Assume $hs\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ takes time $O(\alpha)$. Then Algorithm 12 ($succ_Y\langle\mathcal{C}_{\preceq}\rangle(X_{ll}, Y_{ll})$) takes $O(\alpha c^2 \log^2 n)$.

Proof. Locating the first supported X_{pf} takes $O(\alpha c^2)$ time. Once the algorithm locates the X_{pf}, it constructs a predecessor against every possible Y_{pf}. There are at most $O(c)$ possible choices and $O(c)$ of them require only the simple predecessor algorithm $pred_X\langle\mathcal{C}\rangle$ that takes $O(\alpha c \log n)$ [36] . At most one of them must be taken care specially by Algorithm 12 and takes $O(\alpha c^2 \log^2 n)$. Hence, the total run time is $O(\alpha c^2 \log^2 n)$. □

6.2.4 Feasibility Routine for PF-Intervals for Binary Symmetry-Breaking Disjoint Constraint

The above generic algorithm pushes the length-lex ordering constraint into arbitrary binary symmetric constraints. Specialized algorithms can of course be designed for specific constraints. For instance, there exists a specialized algorithm for binary symmetry-breaking disjoint constraint with $O(1)$ overhead to the binary disjoint constraint. It only based on one key observation: by disjointness, the smallest element of two sets are distinct, the ordering constraint can be handled only by considering the smallest elements. We will illustrate it using the feasibility routine. The successor and predecessor routine are essentially the same.

Example 33. Algorithm 13 implements the feasibility routine for binary symmetry-breaking disjoint constraint. We illustrate it with following examples:

- Suppose $X_{pf} = pf\langle\{1,3\}, 5, 6, 8, 3\rangle$, $Y_{pf} = pf\langle\{2\}, 4, 5, 8, 3\rangle$. The prefixes of sets of both intervals are fixed, all sets in X_{pf} begin with $\{1,3\}$ while all sets in Y_{pf} begin with $\{2\}$. The length-lex ordering constraint is entailed. We can simply call the feasibility routine for binary disjoint constraint. (Line 12)

- Suppose $X_{pf} = pf\langle\{\}, 1, 4, 8, 3\rangle$, $Y_{pf} = pf\langle\{3\}, 4, 5, 8, 3\rangle$. The smallest element of any set in Y_{pf} is 3. By the ordering constraint, the smallest element of any set in X_{pf} can be at most 3. Moreover, disjointness prohibits sets in X_{pf} from taking 3. Hence, the smallest element of

Algorithm 13 $hs\langle\mathcal{D}_{\preceq}\rangle(X_{pf} = pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle, Y_{pf} = pf\langle P_Y, \check{f_Y}, \hat{f_Y}, n_Y, c_Y\rangle)$

Require: $n_X == n_Y$

1: **if** $c_X < c_X$ **then**
2: **return** $hs\langle\mathcal{D}\rangle(X_{pf}, Y_{pf})$
3: **else if** $c_X > c_Y$ **then**
4: **return** $false$
5: **if** $P_X = \emptyset \wedge P_Y = \emptyset$ **then**
6: $\check{f_Y} = \max(\check{f_X}, \check{f_Y})$
7: $\hat{f_X} = \min(\hat{f_Y}, \hat{f_Y})$
8: **else if** $P_X = \emptyset$ **then**
9: $\hat{f_X} = \min(\hat{f_X}, P_{Y0} - 1)$
10: **else if** $P_Y = \emptyset$ **then**
11: $\check{f_Y} = \max(\check{f_Y}, P_{X0} + 1)$
12: **else if** $P_{X0} \geq P_{Y0}$ **then**
13: **return** $false$
14: **return** $(\check{f_X} \leq \hat{f_X} \wedge (\check{f_Y} \leq \hat{f_Y}) \wedge hs\langle\mathcal{D}\rangle(pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle, pf\langle P_Y, \check{f_Y}, \hat{f_Y}, n_Y, c_Y\rangle)$

any set in X_{pf} can only be 1 or 2. We modify X_{pf} to $pf\langle\{\}, 1, 2, 8, 3\rangle$ and now the ordering constraint is entailed. We call the feasibility routine for binary disjoint constraint. (Line 9)

- Suppose $X_{pf} = pf\langle\{\}, 2, 4, 8, 3\rangle$, $Y_{pf} = pf\langle\{\}, 1, 5, 8, 3\rangle$. The smallest element of sets in X_{pf} must not be greater than those in Y_{pf}, hence $\check{f_Y}$ must be at least 2. We get modify Y_{pf} to $pf\langle\{\}, 2, 5, 8, 3\rangle$, and we can pass it to the binary disjoint feasibility routine. (Lines 5–7)

Theorem 9. Algorithm 13 ($hs\langle\mathcal{D}_{\preceq}\rangle(X_{pf}, Y_{pf})$) takes $O(\max(c_X, c_Y))$ time.

Proof. Lines 2, 14 invoke the feasibility routine of the binary disjoint constraint and take $O(\max(c_X, c_Y))$, the rest of lines takes $O(1)$. Hence, it is $O(\max(c_X, c_Y))$ in total. □

It is possible to implement more efficient bound-consistent algorithms for various symmetry-breaking intersection constraints too. The basic idea is discussed in Chapter B.2.

6.2.5 Evaluation

We evaluate the impact of using the combined propagator. We compare the performance between three models. The subset-bound model, the original length-lex model, and the length-lex pushing model, in which the combined propagator is used and the length-lex ordering constraint is pushed into binary constraints. In particular, consider Lines 13 and 14:

Domain	Subset-Bound		Length-Lex		Length-Lex	
Pushing ($\mathcal{D}_{\preceq}$, $atmost1_{\preceq}$)					✔	
(g,s,w)	Time	Fails	Time	Fails	Time	Fails
(3,3,3)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(4,3,5)	19.99	289948	17.87	103462	**6.52**	**36334**
(4,4,6)	96.55	1241016	28.21	40180	**0.01**	**0**
(5,3,6)	61.58	605967	64.3	177220	**24.51**	**67804**
(5,3,7)	x	x	x	x	x	x
(5,4,5)	7.38	76013	2.16	3979	**1.13**	**2309**
(5,4,6)	x	x	x	x	x	x
(5,5,4)	69.93	866005	1.36	311	**0.19**	**175**
(7,7,4)	x	x	x	x	**94.38**	**1739**

Table 6.1: Social Golfer Problem: Pushing the Length-Lex Ordering into Binary Constraints

```
13    cp.post( disjoint(llx[wi,gi],llx[wi,gj]) );
14    cp.post( llx[wi,gi] <= llx[wi,gj] );
```

They can be combined into one propagator which achieves stronger propagation:

```
    cp.post( disjointLe(llx[wi,gi],llx[wi,gj]) );
```

Similarly, Line 21 is replaced by:

```
    cp.post( atmostLe(llx[wi,1],llx[wj,1],1) );
```

This illustrates the flexibility of constraint programming. Constraints can be added and removed from the model without the need to consider its impact on other constraints. We replace three lines by two and introduce a new model, namely Length-Lex + Pushing.

Figure 6.1 presents the results. We use all instances in last section and a few instances that was unsolveable before. The model Pushing is the best model. It not only greatly reduces the number of fails when comparing with the original model, the time spent is reduced dramatically as well. It detects failure for instance $(4, 4, 6)$, which has no solution, without the need of searching. Inferences among constraints is sufficient to detect the inconsistency. Pushing symmetry-breaking constraints into binary constraint yields a significantly stronger propagation over its decomposition. The combined propagator doesn't incur overhead in practice as well. Consider the *fails-time* ratio, which is roughly the number of node visited per seconds, the ratio of the original model and Pushing model are roughly the same.

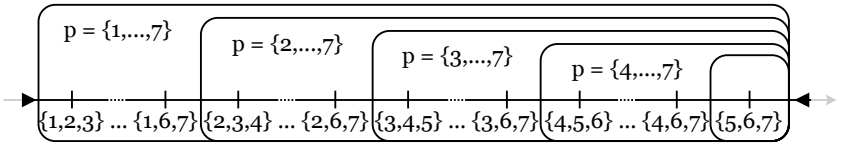

Figure 6.7: How The Most Significant Set Element Determines the Possible Elements.

This shows the great strength of pushing lexicographical-ordering constraint into binary constraints. Almost no overhead in computation time is incurred, while the model enjoys a great improvement in propagation strength. Moreover, the generic pushing algorithm we give in this section makes implement a combined propagator effortless.

6.3 Global Filter for Symmetry-Breaking AllDisjoint

Previous section showed that pushing symmetry-breaking constraints into other constraints are effective. This section discusses a primal filter for the combination of a global *alldisjoint* constraint and a chain of symmetry-breaking constraints.

Definition 13 (Symmetry-Breaking AllDisjoint)**.** $alldisjoint_{\preceq}(X_1, ..., X_m) \equiv alldisjoint(X_1, ..., X_m) \wedge \left(\bigwedge_{i<j} X_i \preceq X_j \right)$.

Intuition The key observation is that the most significant element of a variable X_i (i.e., the smallest value in X_i) determines an upper bound of the possible elements that can be taken by subsequent variables $X_j, \forall j > i$. Since the global *alldisjoint* constraint imposes that all variables take different elements, the total number of elements taken by all variables is known. If an element in X_i is such that there are not enough elements left for the variables $X_{i+1}, \ldots, X_m$, then it must be the case that X_i contains a smaller element.

Figure 6.7 depicts the idea and, in particular, the effect of the symmetry-breaking constraints on the possible values that variables can take. Consider a domain which contains all sets of cardinality 3 drawn from 1..7 and ordered lexicographically. The rectangles show how the most significant element determines the set of possible elements p for a set variables. If the most significant element of a variable X_i is 2, then its possible set is $\{2, ..., 7\}$. Moreover, if there is a lexicographic constraint

between X_i and subsequent variables X_j $(j > i)$, then the set of possible elements for these subsequent variables is of cardinality at most 6, since their most significant element have to be at least 2.

Example 34. Consider a CSP with 3 length-lex variables X_1, X_2, X_3 of cardinality 3, taking their elements from a universe $U(9) = \{1..9\}$, and a constraint $alldisjoint_{\preceq}(X_1, X_2, X_3)$. Assume that $X_1 \in ll\langle\{1,7,8\},\{1,7,9\},9\rangle$, $X_2 \in ll\langle\{2,3,4\},\{7,8,9\},9\rangle$, and $X_3 \in ll\langle\{3,4,5\},\{7,8,9\},9\rangle$. The smallest element of X_2 cannot be 6, since this would leave only elements in $\{6,7,8,9\}$ for filling X_2 and X_3 which need 6 distinct elements in total. It can be seen that the smallest element of X_2 can at most be 4, i.e., $X_2 \in ll\langle\{2,3,4\},\{4,8,9\},9\rangle$.

More propagation is possible when the required elements of earlier variables are considered. X_1 is taking elements 1 and 7, making it impossible for either X_2 or X_3 to take element 7. Suppose X_2 takes 4 as its most significant element, X_2 and X_3 pick elements from the set $\{4,5,6,8,9\}$, whose size is insufficient to fulfill the cardinality requirement. Hence, X_2 cannot start with element 4. $\diamond$

A Reduction Rule We present the primal filter for the symmetry-breaking *alldisjoint*. For simplicity, all set variables are assumed to be of cardinality c.

Rule 1 (Symmetry-Breaking AllDisjoint: Upper Bound)**.**

$$\frac{1 \leq i \leq m \wedge \bigwedge_{i \leq j \leq m}(|X_j| = c) \wedge f = \max\{e | av_e(i) \geq (m-i+1)c\}}{alldisjoint_{\preceq}(X_1,...,X_m) \longmapsto \min(X_{m-i}) \leq f \wedge alldisjoint_{\preceq}(X_1,...,X_m)}$$

where $av_e(i) = (n-e+1) - \sum_{j<i} |\{e' \in req(X_j) \mid e' \geq e\}|$, and $req(X_j)$ returns a set of required element in the domain of variable X_j.

The function $av_e(i)$ returns an upper-bound on the number of elements X_i,...,X_m can take, assuming that X_i starts with element e. If the upper-bound is less than the total cardinality requirement (i.e., $(m-i+1)c$), then the constraint is infeasible. The rule finds the largest element f such that the condition holds and imposes a constraint on the most significant element of X_i accordingly.

The primal filter is independent of the variable representation: it simply posts a constraint on the smallest element of variable X_i. If the length-lex representation for set variables is used, this update is particularly effective, since it directly updates the upper bound of the length-lex interval.

Obviously, Rule 1 does not enforce bound consistency.

Lemma 7 (Incompleteness of Rule 1)**.** Enforcing bound consistency on $alldisjoint_{\preceq}(X_1, ..., X_m)$ is strictly stronger than applying Rule 1.

Proof. Consider variables X_1, X_2, and X_3 of cardinality 3 and drawing elements from $U(9)$, and a global constraint $alldisjoint_{\preceq}(X_1, X_2, X_3)$. Suppose $X_1 \in ll\langle\{1,4,5\},\{1,5,9\},9\rangle$, $X_2 \in ll\langle\{2,4,5\},\{2,5,9\},9\rangle$, and $X_3 \in ll\langle\{3,4,5\},\{3,5,9\},9\rangle$. X_1, X_2, and X_3 has to take an element in $\{4,5\}$. By pigeonhole principle, there is no solution. However, the propagation rule cannot pruning anything. □

We discussed symmetry-breaking binary disjoint constraint $\mathcal{D}_{\preceq}$ earlier. It and Rule 1 offer different perspective for pruning.

Lemma 8 (Rule 1 and $\mathcal{D}_{\preceq}$ are incomparable)**.** Rule 1 and a chain of binary constraints $\mathcal{D}_{\preceq}(X_i, X_j)$ are incomparable.

Proof. We compare the propagation result after Rule 1 on $[X_1, X_2, X_3]$, and a set of binary constraints: $\mathcal{D}_{\preceq}(X_i, X_j)\ \forall 1 \leq i < j \leq 3$.

	X_1	X_2	X_3
Initial	$ll\langle\{1,7,8\},\{1,7,9\},9\rangle$	$ll\langle\{2,3,4\},\{7,8,9\},9\rangle$	$ll\langle\{3,4,5\},\{7,8,9\},9\rangle$
Rule 1	$ll\langle\{1,7,8\},\{1,7,9\},9\rangle$	$ll\langle\{2,3,4\},$ **{3,8,9}**$,9\rangle$	$ll\langle\{3,4,5\},\{6,8,9\},9\rangle$
$\mathcal{D}_{\preceq}$	$ll\langle\{1,7,8\},\{1,7,9\},9\rangle$	$ll\langle\{2,3,4\},\{4,6,9\},9\rangle$,	$ll\langle\{3,4,5\},$ **{5,6,9}**$,9\rangle$

□

6.3.1 Evaluation

We evaluate the impact of the reduction rule for symmetry-breaking all-disjoint constraint. The filtering rule is indeed independent of the underlying domain representation. Hence, the evaluation considers both subset-bound and length-lex domain. We post the global constraint among the first group variables in all weeks. All first group variables contain player 1 and there is a $atmost(1)_{\preceq}$ binary constraint among them. It is possible to introduce an auxiliary variable `aux[wi]`, which removes the first player, for each first group variable `llx[wi,1]`, and post a symmetry-breaking all-disjoint constraint $alldisjoint_{\preceq}$ over them. The following constraint is posted.

Domain	Subset-Bound		Subset-Bound		Length-Lex		Length-Lex	
Pushing ($\mathcal{D}_{\preceq}$, $atmost1_{\preceq}$)					✔		✔	
Global SymBreak (Rule 1)			✔				✔	
(g,s,w)	Time	Fails	Time	Fails	Time	Fails	Time	Fails
(3,3,3)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(4,3,5)	24.77	289948	20.05	192448	6.52	36334	**3.88**	**17842**
(4,4,6)	96.55	1241016	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(5,3,6)	61.58	605967	55.84	445333	24.51	67804	**9.73**	**24502**
(5,3,7)	x	x	599.06	4121349	x	x	**132.43**	**194723**
(5,4,5)	7.38	76013	8.56	75593	1.13	2309	**0.83**	**1537**
(5,4,6)	x	x	x	x	x	x	x	x
(5,5,4)	69.93	866005	93.67	866005	**0.19**	**175**	**0.19**	**175**
(6,3,7)	x	x	504.01	3072559	x	x	**53.47**	**82005**
(6,4,5)	**0.1**	692	0.12	692	**0.1**	**89**	**0.1**	**89**
(6,5,5)	x	x	x	x	x	x	**1537.17**	**1206172**
(7,7,4)	x	x	x	x	94.38	**1739**	**93.32**	**1739**

Table 6.2: Social Golfer Problem: Primal Filter for Global Symmetry-Breaking AllDisjoint Constraint

```
cp.post( symbreak_alldisjoint(all(wi in 1..w) aux[wi],p) );
```

Table 6.2 presents the result. We compare its performance with the original model as well as the subset-bound model. The global symmetry-breaking alldisjoint filter is useful for both subset-bound and length-lex model. It is able to solve some reachable instances in the original model, i.e. $(5, 3, 7)$, $(6, 3, 7)$. For example, for $(5, 3, 6)$, the number of fails is cut by more than half in the length-lex model and 25% in the subset-bound model. The primal filter does not always improve propagation though, especially when the instance is not too tight, i.e. player 1 doesn't have to play with many other players. It doesn't reduce the number of fails in instances $(5, 5, 4)$, $(6, 4, 5)$, and $(7, 7, 4)$. However, the time spent to solve those instances are not increased. The filter incurs a negligible overhead to the system. It is also interesting to point out that in instance $(4, 4, 6)$, the subset-bound model becomes trivial after adding the global filter. On the other hand, the instance is already trivial for the length-lex model using binary symmetry-breaking constraints. This suggests that the technique of pushing ordering constraint into binary constraint enables to solver to gain some global perspective of the problem.

6.4 Dual Modeling for Length-Lex Set Variables

6.4.1 Overview

Most set-CSPs exhibit symmetries; among set variables, values or both. Breaking either symmetry on its own can be done by ordering the variables in the constraint model (variable symmetries) or in the dual model (value symmetries). A more complex problem lies in breaking both forms of symmetries simultaneously while guaranteeing that for a given symmetry class a solution can still be found. A priori, it is unclear whether enforcing the length-lex ordering on both variables and values will still leave some solutions in each symmetry class. We show in this section that imposing a *double length-lex ordering* on a fully interchangeable set-CSP does not eliminate all solutions in each symmetry class and thus can be safely applied. Intuitively, a *fully interchangeable* set-CSP is a set-CSP in which both the variables and the values are fully interchangeable. This section addresses the theoretical and practical issue of breaking both forms of symmetries simultaneously for such CSPs.

In 0/1 matrix formulation, these symmetries are successfully broken by imposing a lexicographic ordering on both rows and columns. It is guaranteed that some solutions of each symmetry class remains after this process. Since the length-lex ordering provides a total ordering on its sets, it also provides an ideal vehicle to break symmetries and we would like to use a similar technique with length-lex variables. Variable symmetries can be broken by imposing an ordering on the set variables. Now if the values are also interchangeable, we can consider the dual problem and impose an ordering on the dual variables. Our approach is conducted in two steps: 1) break forms of value symmetries using dual modeling and ordering constraints, 2) extend the approach to tackle fully interchangeable set-CSPs.

6.4.2 Breaking Value Symmetry

To ease the presentation, we use a slightly different notation than the one we used in the thesis. We revisit the definition of CSPs as in [18], whereby all the constraints are abstracted by a Boolean function that takes an assignment and returns *true* if they are satisfied simultaneously. We will use the notation P for primal set variables and Q for dual set variables.

Definition 14 (Set-CSP)**.** A set-CSP is a pair $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is the universe (the set of all possible values) for these variables. An (primal) assignment $\gamma : \mathcal{X} \to \mathbb{P}(\mathcal{D})$ maps variables to sets. $C : (\mathcal{X} \to \mathbb{P}(\mathcal{D})) \to \mathit{bool}$ is a constraint that specifies which assignments are solutions (i.e. $C(\gamma) = \mathit{true}$).

A common form of value symmetry is value-interchangeability, meaning that the values taken by the variables do not matter. What actually matter is which variables take the same value. Under our set-CSP definition, value interchangeability corresponds to modifying the output of an assignment. To simplify the notation, we define a set mapping function that applies to each set element to return a set value.

Definition 15 (Mapping Function)**.** Given a function $f : U(n) \to U(n)$, the set mapping function ϕ_f takes a ground set s, applies f to each of its elements and returns the image set: $\phi_f(s) \equiv \{f(e) | e \in s\}$.

Example 35. Consider the set-CSP with two variables $P_1, P_2 \in ll\langle\{1\}, \{2\}, 2\rangle$ with constraint $P_1 \neq P_2$. The possible set values are interchangeable ($\{1\}$ and $\{2\}$). This can be expressed with the bijective function $f(j) = 3 - j$. An assignment γ maps the variables to sets, for instance we have $\gamma(P_1) = \{1\}, \gamma(P_2) = \{2\}$. A symmetric solution can be obtained by applying $f(j)$ to each element in $\gamma(P_i)$. ϕ_f is defined by: (i.e. $\phi_f(\gamma(P_1)) = \{2\}$ and $\phi_f(\gamma(P_2)) = \{1\}$). ◇

Example 36. The social golfer problem also exhibits value symmetry. Given any solution, we attain another solution by permuting players. Figure 2.5 is symmetric to Figure 2.1 by permuting players 3, 6, and 9. ◇

We now formally define value-interchangeable CSP based on the set-CSP definition above. Its solution is preserved by permuting any subset of set values.

Definition 16 (Fully Value-Interchangeable Set-CSP)**.** A Set-CSP is fully value-interchangeable if and only if for a solution γ, any bijection $\tau : D \to D$, the assignment $\gamma' = \phi_\tau \circ \gamma$ is also a solution.

We can break value symmetry by enforcing an ordering constraint among variables in the dual model in which the role of variables and values are interchanged. The idea is similar to the 0/1-matrix model, where row-interchangeability in a model is equivalent to column-interchangeability in its transpose. Figure 6.8 illustrates the idea. Notice that the analysis is independent of the domain representation.

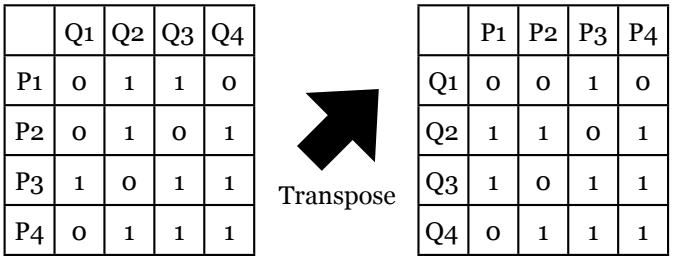

Figure 6.8: Reformulating Set-CSPs as a 0/1 Matrix

First we neglect all the cells notated by $P_1, ..., Q_4$. Consider a CSP which exhibits row symmetry, we obtain a symmetric solution by swapping assignment of any pairs of rows. The matrix on the left represents a solution to the CSP in which every cell represents a variable. There are altogether 16 cells. We obtain another matrix, the one on the right, by transposing it. Row symmetry in the original matrix becomes column symmetry in the transposed matrix.

Suppose P_1 is a set variable taking element i if the cell in column Q_i is 1. In other words, Q_i represents element in the universe. $P_1 = \{2, 3\}$ in matrix. We are assigning values to variables. Row symmetry corresponds to variable interchangeability. In the transposed matrix, all the variables become values. The symmetry class become value interchangeability. We may apply the methods for breaking variable symmetry to deal with value symmetry. All we need is to interchange the role of variables and values. A *dual* set of variables is introduced. Figure 6.9 visualizes this idea.

Formally, given a set-CSP, we can remodel it using a 0/1-matrix model. Suppose there are m set variables $P_1, ..., P_m$ and n values $\{1, ..., n\}$. We can construct a $m \times n$ matrix (denoted as $Z_{i,j} \in \{0, 1\}$ for $i \in \{1..m\}, j \in \{1..n\}$) is constructed. $Z_{i,j} = 1$ if and only if $j \in P_i$, and $Z_{i,j} = 0$ if otherwise. Every row $[Z_{i,1}, ..., Z_{i,n}]$ is indeed the characteristic vector of X_i, variable symmetry corresponds to row symmetry. Similarly, value symmetry corresponds to column symmetry. Every column $[Z_{1,j}, ..., Z_{m,j}]$ is the characteristic vector of the dual variable. When we take the transpose of the matrix, row and column symmetry interchanged. Column symmetry can now be tackled as a row symmetry, equivalently, value symmetry can be tackled as dual variable symmetry.

Given a fully value-interchangeable set-CSP, we can amend the model as following to eliminate all symmetric solutions caused by value-interchangeability.

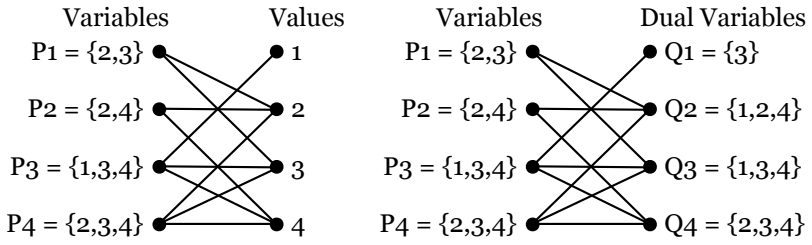

Figure 6.9: Dual Modeling in Sets

Definition 17 (Length-Lex Ordered Dual Set-CSP)**.** Let $\langle P_M, N, C\rangle$ be a CSP where $P_M = \{P_1, ..., P_m\}$ are (primal) set variables and $N = \{1, ..., n\}$. Its length-lex ordered dual version is defined as $\langle P_M \uplus Q_N, N \uplus M, C'\rangle$ where $Q_N = \{Q_1, ..., Q_n\}$ are the dual set variables, $M = \{1, ..., m\}$ and

$$C' \equiv C \wedge (Q_1 \preceq \ldots \preceq Q_n) \wedge \bigwedge_{i \in N, j \in M} (j \in P_i \Leftrightarrow i \in Q_j)$$

Theorem 10. Given a fully value-interchangeable CSP, its length-lex ordered dual version eliminates all but one solutions in each symmetry class.

Proof. Consider the 0/1-matrix model constructed from the set model, ordering constraints among dual variables is equivalent to enforcing length-lex ordering on 0/1 characteristic vector among columns. As length-lex is a total order, this leaves only one solution in each symmetric class. □

Example 37. Following example 35, in the length-lex ordered dual version,we have the dual set variables $Q_1, Q_2 \in ll\langle\{\}, \{1, 2\}, 2\rangle$. We add the symmetry breaking constraint $Q_1 \preceq Q_2$ and the channeling constraint $\bigwedge_{i \in \{1,2\}, j \in \{1,2\}} (j \in P_i \Leftrightarrow i \in Q_j)$. The impact of the symmetry breaking constraints is the elimination of the solution $\gamma(P_1) = \{2\}$ and $\gamma(P_2) = \{1\}$ because $\gamma(Q_1) = \{2\} \not\preceq \gamma(Q_2) = \{1\}$. ◇

6.4.3 Breaking Variable and Value Symmetry

We now address the issue with breaking both value and variable symmetries simultaneously. It leads to a question of in a problem that exhibits both variable and value symmetry, whether or not we can post both variable and value symmetry breaking constraint on the same model. We begin with formalizing the idea of set-CSP and fully-interchangeability. These concepts facilitate our proofs.

A fully interchangeable set-CSP is that all variables (values) are interchangeable. Permuting the assignment of variables and values preserve a solution.

Definition 18 (Fully Interchangeable Set-CSP)**.** A set CSP is fully interchangeable if and only if when γ is a solution, for any bijective $\sigma : V \to V$ and $\tau : D \to D$, and a mapping function $\phi_f(s) = \{f(e)|e \in s\}$, assignment $\gamma' = \phi_\tau \circ \gamma \circ \sigma$ is also a solution.

Given a fully interchangeable set-CSP, we would like to break such interchangeability by posting length-lex ordering constraint on both primal and dual variable. We formally define this as,

Definition 19 (Double Length-Lex Ordered Primal/Dual Set-CSP)**.** Let $\langle P_M, N, C\rangle$ be a CSP where $P_M = \{P_1, ..., P_m\}$ are (primal) set variables and $N = \{1, ..., n\}$. Its double length-lex primal/dual version is defined as $\langle P_M \uplus Q_N, N \uplus M, C'\rangle$ where $Q_N = \{Q_1, ..., Q_n\}$ are the dual set variables, $M = \{1, ..., m\}$ and

$$C' \equiv C \wedge (P_1 \preceq \ldots \preceq P_m) \wedge (Q_1 \preceq \ldots \preceq Q_n) \wedge \bigwedge_{i \in N, j \in M} (j \in P_i \Leftrightarrow i \in Q_j)$$

However, it is unclear about the soundness of such method. It may completely wipe out all solution of some symmetry class. In the remaining section, we prove that such problem doesn't exists. We prove by reducing our set model to 0/1-matrix model.

A similar problem is encountered in the double lex method for matrix model, it tackles row and column symmetry by enforcing lex-ordering constraints among rows and columns. The lex-ordering constraints, however, cannot be posted arbitrarily. In particular, if we enforce lex-ordering constraint among the rows but anti-lex-ordering constraint among the columns, all solutions of some symmetric class will be wiped out by the symmetry breaking constraints. We illustrate this with the following example.

Example 38. Suppose we have a two by two matrix of 0/1-variables, $X_{r,c}$, with constraints $\sum_{1 \le r' \le 2} X_{r',c} = 1, \forall 1 \le c \le 2$ and $\sum_{1 \le c' \le 2} X_{r,c'} = 1, \forall 1 \le r \le 2$. Clearly, both rows and columns are interchangeable. There are two solutions: If we enforce double lex-ordering or double anti-lex-ordering constraint on the matrix model, we will get either one of the above solutions. However, if we enforce lex-ordering on rows and anti-lex-ordering on columns, or vice verse, none of

1	0
0	1

0	1
1	0

	Original	0/1 of Original	Padded	0/1 of Padded
P_1	$\{2,3\}$	$\{0,1,1,0\}$	$\{-4,-3,2,3\}$	$\{1,1,0,0,0,0,1,1,0\}$
P_2	$\{2,4\}$	$\{0,1,0,1\}$	$\{-4,-3,2,4\}$	$\{1,1,0,0,0,0,1,0,1\}$
P_3	$\{1,3,4\}$	$\{1,0,1,1\}$	$\{-4,1,3,4\}$	$\{1,0,0,0,0,1,0,1,1\}$
P_4	$\{2,3,4\}$	$\{0,1,1,1\}$	$\{-4,2,3,4\}$	$\{1,0,0,0,0,0,1,1,1\}$
Q_1	$\{3\}$	$\{0,0,1,0\}$	$\{-4,-3,-2,3\}$	$\{1,1,1,0,0,0,0,1,0\}$
Q_2	$\{1,2,4\}$	$\{1,1,0,1\}$	$\{-4,1,2,4\}$	$\{1,0,0,0,0,1,1,0,1\}$
Q_3	$\{1,3,4\}$	$\{1,0,1,1\}$	$\{-4,1,3,4\}$	$\{1,0,0,0,0,1,0,1,1\}$
Q_4	$\{2,3,4\}$	$\{0,1,1,1\}$	$\{-4,2,3,4\}$	$\{1,0,0,0,0,0,1,1,1\}$

Figure 6.10: Preserving the length-lex ordering by padding dummy elements

these solutions satisfies these constraints. In other words, all solutions of the same symmetric class are wiped out. ◇

Reference [17] shows that in matrix formulation, there is always some way to break both row and column symmetry while preserving some solutions in every symmetric class.

Variable and value symmetry for length-lex set variables can be tackled in a similar fashion. We reduce to the 0/1 matrix model and enforce ordering constraints on both rows and columns. But instead of enforcing lexicographical ordering constraints, we enforce length-lex ordering. The analysis of the double-lex method cannot be applied directly our model. The subtlety can be resolved by transforming the sets, making the length-lex ordering identical to lex-ordering. The key observation is that, when sets are of the same cardinality, their length-lex order is equivalent to the lexicographic order. We reduce the length-lex ordering to lexicographic ordering by padding some dummy elements to the front and make all sets the same size.

Example 39. The upper half of Figure 6.10 illustrates difference between length-lex and lex ordering for a universe $U(4)$ and four sets P_1, P_2, P_3, P_4. The first column shows the sets in length-lex order. These original sets are not in lex-order $\leq_{lex}$ as $P_2 >_{lex} P_3$. The 0/1 characteristic function (second column) is not in anti-lex-order $\geq_{lex}$ either since $P_3 <_{lex} P_4$. By padding dummy elements (third column), the sets are in both length-lex-order and lex-order and their 0/1 characteristic functions are in anti-lex order.

The lower half of Figure 6.10 illustrates the dual sets Q_1, Q_2, Q_3, Q_4 (i.e. $j \in P_i \Leftrightarrow i \in Q_j$). The

	-4	-3	-2	-1	0	Q_1	Q_2	Q_3	Q_4
-4	1	1	1	1	1	1	1	1	1
-3	1	1	1	1	1	1	0	0	0
-2	1	1	1	1	1	1	0	0	0
-1	1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	0	0	0	0
P_1	1	1	0	0	0	0	1	1	0
P_2	1	1	0	0	0	0	1	0	1
P_3	1	0	0	0	0	1	0	1	1
P_4	1	0	0	0	0	0	1	1	1

Figure 6.11: The 0/1 matrix

original dual sets are not in lex nor anti-lex order. By padding elements, all dual sets are in both length-lex-order and lex-order (third and forth column). $\diamond$

The formal definition of padding is as follows.

Definition 20 (Padding)**.** We abuse the notation of a universe to allow negative value element, such that $U'(n) = \{-n, ..., -1, 0, 1, ..., n\}$. $pad_n : \mathbb{P}(U(n)) \to \mathbb{P}(U'(n))$ maps a set to a n-set padded by dummy elements. Formally, given $s \subseteq U(n)$ and $c = |s|$, $pad_n(s) \equiv \{-n, ..., -(c+1), s_1, ..., s_c\}$.

Lemma 9. Suppose $s, t \subseteq U(n)$. $s \preceq t \Leftrightarrow pad_n(s) \leq_{lex} pad_n(t)$.

Proof. Trivial when $|s| = |t|$. When $|s| < |t|$, denote $s' = pad_n(s), t' = pad_n(t)$, observe that $s'_{1..n-|t|} = t'_{1..n-|t|}$ as they are dummy elements. $s'_{n-|t|+1}$ is a dummy negative element, whilst $t'_{n-|t|+1} = t_1 > 0$, Hence $s' \leq_{lex} t'$. $\square$

Figure 6.11 illustrates the key idea of the proof. We use Example 39. It is constructed by padding dummy elements to sets. The lower right part is the original sub-matrix, where the primal sets $\{P_i\}$ and dual sets $\{Q_j\}$ are in length-lex order. Rows P_1, P_2, P_3, P_4 correspond to the padded sets on Figure 6.10, so as columns Q_1, Q_2, Q_3, Q_4. The upper left corner are dummy cells filled by 1s to ensure that they always remain in the upper left corner in the anti-lex ordering. Both rows and columns are in anti-lex ordering, while in the lower left sub-matrix, rows and columns are in length-lex ordering. Following is the formal proof,

Theorem 11. Given a fully interchangeable CSP, its double length-lex primal/dual version does not eliminate all solutions in each symmetry class.

Proof. Outline of the proof: we show that any solution γ of the CSP $\langle P_M, N, C\rangle$, there exists a solution γ_o in $\langle P_M \uplus Q_N, N \uplus M, C'\rangle$ such that there exists γ and γ_o are symmetric, formally:

$$\exists \sigma : P_M \to P_M, \tau : N \to N, \text{ s.t. } \forall P_i \in P_M, \tau(\gamma(\sigma(P_i))) = \gamma_o(P_i)$$

In other words, we want to show that there exists a symmetrical solution that satisfies the double length-lex ordering constraint. We apply the double anti-lex analysis for 0/1 matrix.

Consider a solution γ, transform it to $\gamma' = pad_\rho\gamma$ where $\rho = \max(n, m)$. We construct a 0/1 matrix $Z_{-\rho..\rho,-\rho..\rho}$ where $j \in \gamma'(P_i) \Leftrightarrow Z_{i,j} = 1$ and fill the upper left dummy cells with 1 (i.e. $\forall i \leq 0, j \leq 0, Z_{i,j} = 1$). Notice that the lower right sub-matrix $Z_{1..m,1..n}$ correspond to solution γ. To construct a solution that satisfies double length-lex ordering, we instead enforce double anti-lex ordering on matrix Z. Lemma 9 implies that double anti-lex on Z guarantees the solution corresponds the lower right sub-matrix is in double length-lex order. Moreover, enforcing ordering constraint between a pair of rows (or columns) can be regarded as swapping the rows (or columns) upon violation. Swapping of rows (or columns), in turn, corresponds to modifying the variable mapping σ (or value mapping τ). Hence, from a solution γ, we can construct a matrix Z by choosing the right mapping function σ and τ, such that it is in double anti-lex order, and Z has a sub-matrix corresponds to a solution γ_o which is in double length-lex order. □

6.4.4 Evaluation

We evaluate the effectiveness of the dual modeling method for breaking value symmetry. Once again, the social golfer problem is used. We compare the dual modeling method for both subset-bound and length-lex set variables, and the original length-lex model. Notice that the social golfer problem presented here does not directly apply the dual set variables, since we can exploit the problem structure by a bit further since we know that a player, which corresponds to the dual variable, plays at exactly one group in every week. We apply the model introduced by Barnier and Brisset [3] (see Figure A.3), the dual variable is a set of vectors, each vector represents the group player p belongs to in a week. Both models have the same set of solutions. But the dual vector model achieves more propagation.

Figure 6.12 presents the length-lex model in the COMET language. The initialization and search

```
1 LengthLexVar<CP> llx[Weeks,Groups](cp,p,s);
2 var<CP>{set{int}} sbx[Weeks,Groups](cp,Players,s..s);
3 var<CP>{int} y[Players,Weeks](cp,Groups);
4 var<CP>{set{int}} aux[Weeks](cp,Players,s..s);
5
6 solve<cp>{
7   forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj)
8     cp.post( disjointLe(llx[wi,gi],llx[wi,gj]) );
9
10  forall (wi in Weeks, wj in Weeks, gi in Groups, gj in Groups : wi < wj)
11    cp.post( atmost(llx[wi,gi],llx[wj,gj],1) );
12
13  forall (wi in Weeks, wj in Weeks : wi < wj)
14    cp.post( atmostLe(llx[wi,1],llx[wj,1],1) );
15
16  forall (wi in Weeks)
17    cp.post( removeMin(llx[wi,1],aux[wi]) );
18  cp.post( symbreak_alldisjoint(all(wi in 1..w) aux[wi],p) );
19
20  forall(wi in Weeks, gi in Groups)
21    cp.post( channel(llx[wi,gi],sbx[wi,gi]) );
22
23  forall (wi in DualWeeks)
24    cp.post( dualChannel(all(gi in Groups)sbx[wi,gi], all(pi in Players)y[pi,wi]) );
25
26  forall (pi in Players, pj in Players : pi < pj)
27    cp.post( lexleq(all(wi in Weeks)y[pi,wi],all(wi in Weeks)y[pj,wi]) );
28 }
```

Figure 6.12: COMET Model for Social Golfer Problem using Dual Modeling

part are skipped since they are equivalent to previous models. Subset-bound variables `sbx` are introduced as auxiliary variables for channeling with the dual variables. In the model, Lines 6–18 are identical to the previous model. Lines 20–21 are the channeling constraint between the two representations, to guarantee that the values they are taking agree, i.e. `llx[wi,gi] == sbx[wi,gi]`. Lines 23–24 connect the primal and dual variable, and Lines 26–27 are the symmetry-breaking constraints that eliminate the interchangeability among players.

Table 6.3 presents the results. Instances used in the previous evaluation as well as some larger instances. Four models are evaluated: the original primal subset-bound and length-lex model which uses symmetry-breaking constraint for eliminating variable symmetry, and the dual model for both representations which add dual constraints for breaking value symmetry. The goal of the evaluation is two-fold. First, to see how many value symmetry are left and not pruned by the original model. Second, to see the difference between the subset-bound and length-lex representation.

The dual-length-lex model solves all the instances efficiently while the dual-subset-bound model solves all but one instances. Breaking value symmetry allows us to solve a much larger instance with

Domain	Subset-Bound		Subset-Bound		Length-Lex		Length-Lex	
Pushing ($\mathcal{D}_{\preceq}$, $atmost1_{\preceq}$)					✔		✔	
Global SymBreak (Rule 1)	✔		✔		✔		✔	
Dual Model (Ref. [3])			✔				✔	
(g,s,w)	Time	Fails	Time	Fails	Time	Fails	Time	Fails
(3,3,3)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(4,3,5)	20.05	192448	0.11	968	3.88	17842	**0.04**	**136**
(4,4,6)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(5,3,6)	55.84	445333	4.21	29791	9.73	24502	**2.58**	**5508**
(5,3,7)	599.06	4121349	25.63	157285	132.43	194723	**13.08**	**16361**
(5,4,5)	8.56	75593	**0.24**	1594	0.83	1537	0.25	**428**
(5,4,6)	x	x	337.35	2013980	x	x	**198.16**	**259616**
(5,5,4)	93.67	866005	0.33	2320	0.19	**175**	**0.15**	**175**
(6,3,7)	504.01	3072559	0.22	935	53.47	82005	**0.19**	**219**
(6,4,5)	0.12	692	0.11	473	**0.1**	89	**0.1**	**83**
(6,5,5)	x	x	**36.49**	195986	1537.17	1206172	45.76	**34375**
(7,7,4)	x	x	x	x	93.32	**1739**	**50.96**	**1739**

Table 6.3: Social Golfer Problem: Breaking Value Symmetry with Dual Modeling Method.

dramatically small number of nodes. For example, for instance $(5, 3, 6)$, the size of the search tree reduces by 5 times when the dual model is added to the original length-lex model. More over, both dual models solve more instances than the original.

The length-lex dual model is generally faster than the subset-bound model. However, it is interesting to see that the length-lex model has dramatically less fail nodes than the subset-bound model. For example, for instance $(6, 5, 5)$, where both motels spend roughly the same time to solve the problem, the subset-bound model visits 5.7 times more nodes that the length-lex model.

Table 6.4 gives a close look to the data. The length-lex model reduces the size of the search tree at least by a few times (shown in the first column). The difference between fails count become more apparent when the problem gets harder. The second and third column gives the approximated number of search nodes per second of the two models. Clearly, the length-lex model spends a lot more than in doing inference than the subset-bound. The last column gives the ratio between the search nodes per second. It suggests that if we are able to speed up the inference process for the length-lex model, we can beat the subset-bound representation.

g,s,w	Subset-Bound Fails / Length-Lex Fails	Subset-Bound Fails / Subset-Bound Time	Length-Lex Fails / Length-Lex Time	Subset-Bound Ratio / Length-Lex Ratio
(3,3,3)	∞	0	0	∞
(4,3,5)	7.12	9132.08	3578.95	2.55
(4,4,6)	∞	0	0	∞
(5,3,6)	5.41	7072.89	2139.03	3.31
(5,3,7)	9.61	6137.47	1251.03	4.91
(5,4,5)	3.72	6614.11	1712	3.86
(5,4,6)	7.76	5969.95	1310.16	4.56
(5,5,4)	13.26	7138.46	1166.67	6.12
(6,3,7)	4.27	4230.77	1164.89	3.63
(6,4,5)	5.7	4300	855.67	5.03
(6,5,5)	5.7	5371.39	751.23	7.15
(7,7,4)	x	x	34.13	n/a

Table 6.4: Social Golfer Problem: Fails-to-Time Ratio of Subset-Bound and Length-Lex.

	Subset-Bound (and its variants)	**Length-Lex**	ROBDD
Propagation	Loose	**Strong**	Very Precise
Space	$O(n)$	$\mathbf{O(c)}$	Potentially Exponential
Efficiency	Fast	$\mathbf{O(poly(c))}$	Potentially Slow
Convergence	Fast	Slow	Potentially Slow

Figure 6.13: Comparison over Different Set Domain Representations. (n is the universe size, c is the cardinality upper bound.)

6.5 Conclusion

The length-lex domain representation offers a total ordering which makes it a good vehicle for symmetry breaking. In this chapter, we presented three different methods for incorporating symmetry-breaking techniques using the length-lex set variables.

Binary Symmetry-Breaking Propagators The first method gives an generic and efficient algorithm for combining an arbitrary binary constraint and a length-lex ordering constraint. The algorithm incurs a minimal overhead to the running time of the decomposition. We showed that, both theoretically and practically, combining these propagators achieves a much stronger propagation than its decomposition. We can solve larger instances with the use of the combined propagator.

Global Primal Symmetry-Breaking Filter The second method introduces a global constraint for the combination of a global alldisjoint constraint and a chain of length-lex symmetry-breaking constraints. It pushes the idea of combining binary propagators even further, and gives a more global perspective for pruning infeasible values. The key idea is that the most significant element, which is constrained by the symmetry-breaking constraints, determines the size of the possible set. That said, a large most significant element implies a small possible set. This gives us a huge opportunity for propagation.

Dual Modeling for Breaking Value Symmetry The third method adopts the double-lex method used commonly in matrix model for breaking value symmetry. Since a set-CSP can be trivially transformed into a matrix model, in which variable and value symmetry corresponds to row and column symmetry. Therefore, we are able to break value symmetries in set-CSPs by introducing a set of dual variables, which interchanges the role of variables and values. However, a problem arises since we may not be able to post variable and value symmetry-breaking constraints under the same model, and there is a chance that all solutions in a symmetry class get wiped out. This is caused by the fact that different symmetry-breaking constraints try to preserve different canonical solutions. Nonetheless, we proved the soundness property of our method, that it is safe to post both symmetry-breaking constraints on variable and values, by an elegant reduction to the 0/1-matrix model.

We empirically evaluate the performance of our proposal, and show that these methods are robust and advance the state-of-the-art solution to the problem.

Chapter 7

Exponential Length-Lex Propagators

7.1 Overview

This chapter proposes something outrageous. Since most CSPs are NP-complete, CP uses filtering algorithms and constraint propagation to reduce the variable domains and hence the search tree to explore. The hope is that the reduction in the search space is sufficient to solve problems of interest in reasonable time. In general, researchers have focused on designing polynomial-time algorithms for filtering, leaving the potentially exponential behavior in the search component. There are exceptions of course, and we will review some of them later, but researchers overwhelmingly focus on polynomial-time filtering algorithms, sometimes at the expenses of enforcing arc or bound consistency. This chapter takes the other road and argues that exponential filtering algorithms and constraint propagation may be highly beneficial in practice. It is motivated by the fact that reasonable exponential behavior in the filtering algorithm may produce significant reduction of search space and can therefore be cost-effective. Moreover, such a reasonable exponential behavior has beneficial effects on constraint propagation allowing further reduction of the search space, an observation made by Bessiere and Régin in [7] where they solve CSPs on the fly to achieve arc consistency on a global constraint. In particular, we show that the length-lex propagators takes exponential time to converge in general. It, nonetheless, reduces the search space by orders of magnitude when comparing with the classical subset-bound domain. Finally, since the overall approach is exponential

in the worst case, it may be preferable to shift some of the exponential behavior from the largely agnostic search to the filtering component where we can exploit the semantics of the constraints and locality.

This chapter evaluates the idea of exponential propagation in the context of CSPs over length-lex set variables. In this representation, filtering algorithms for many elementary constraints typically take polynomial-time. But the constraint-propagation algorithm may take exponential time to converge to a fixpoint [58].[1] There is thus an abundance of negative theoretical results on richer set representations. Yet, on a wide variety of standard benchmarks, the richer representations bring orders of magnitude improvements compared to the subset-bound domain or traditional encodings in terms of finite-domain variables[34, 73].

In the previous chapter, we examine the fail-time ratio between the subset-bound and length-lex representation. It appears that length-lex is spending more time on constraint propagation per node. Indeed, it is caused by the exponential behavior, which will be proved in this section. The goal of this section is thus to explore whether it is beneficial to boost constraint propagation over set variables even further. Its main contributions are twofold. First, it proves the $W[1]$-Hardness and NP-completeness of unary intersection constraints for length-lex domains, which also generalizes to subset-bound domains. Second, we propose exponential filtering algorithms for these intersection constraints and show that they bring another order of magnitude improvement in efficiency compared to existing approaches.

The rest of this chapter is organized as follows. Section 7.2 discusses intractability issues for set variables. Section 7.3 contrasts the theoretical results with an experimental evaluation of the various domains. Section 7.4 proposes an exponential-time propagator for a $W[1]$-hard unary intersection. Section 7.5 evaluates the effectiveness and efficiency of the proposed constraint. Section 7.6 discusses some of the related work.

[1] The fact that constraint-propagation algorithms may take exponential time to converge is not specific to set variables. It appears for instance in numerical continuous CSPs (e.g., [47]) and the propagation of cumulative constraints [50].

7.2 Theoretical Results on Intersection Constraints

We now present a number of theoretical results, which shed light on the behavior and complexity of filtering algorithms and constraint propagation on set domains.

The constraint propagation algorithm iterates over all constraints and invoke one at a time. The process repeats until a fixpoint in which no constraint is able to further reduce the domain. The iteration can be in any order. However, the order of invoking constraints has a huge impact on the efficiency in reaching the fixpoint. We start a pathological example in which a bad order of constraint propagation make severely dampen the solver.

A Simple but Annoying Example Consider a CSP with one length-lex set variable and 6 unary intersection constraints shown in Figure 7.1. We focus our attention on the lower bound. On the left is the pathological case, it is the worst thing we want to see. The propagators are scheduled in a way that every time only one value, the lower bound itself, is removed from the domain. It takes a long time to reach the fixpoint since a set domain potentially contains an exponential number of sets. On the right, is the best case we can ever get, the constraint propagation algorithm reaches the fixpoint in a single step.

In general, the length-lex domain suffers from the problem of reaching a fixpoint. Since it contains an exponential number of domain values, and each propagator in worst case removes only one value from the domain. In worst case it takes exponential number of iterations until it reaches the fixpoint. The question is whether or not we can find the holy grail, which give us a good sequence of propagating constraints. However, find the minimum number of steps to reach the fixpoint is intractable.

The Hardness Proof In the following, we use $bc_\theta\langle\mathcal{C}\rangle$ to denote a bound-consistenct propagator and $hs_\theta\langle\mathcal{C}\rangle$ to denote a feasibility routine for constraint $\mathcal{C}$ on a θ-domain. Recall that the feasibility routine hs returns a boolean value indicating if there is a solution: $hs_\theta\langle\mathcal{C}\rangle(X) \equiv \exists s \in d(X), \mathcal{C}(s)$.

The feasibility routine is the basic component of a length-lex bound-consistent propagator. Indeed, Chapters 4 and 5 introduced generic propagators for unary and binary constraint which only relies on a feasibility routine: once such a routine is available, bounds can be found using a binary

Step	A Pathological Case	The Holy Grail
	$X \in ll\langle\{1,2,3\},\bullet,7\rangle$	$X \in ll\langle\{1,2,3\},\bullet,7\rangle$
1	$C_1 \Downarrow$	$C_6 \Downarrow$
	$X \in ll\langle\{1,2,\mathbf{4}\},\bullet,7\rangle$	$X \in ll\langle\mathbf{\{1,4,5\}},\bullet,7\rangle$
2	$C_2 \Downarrow$	
	$X \in ll\langle\{1,2,\mathbf{5}\},\bullet,7\rangle$	
3	$C_3 \Downarrow$	
	$X \in ll\langle\{1,2,\mathbf{6}\},\bullet,7\rangle$	
4	$C_1 \Downarrow$	
	$X \in ll\langle\{1,2,\mathbf{7}\},\bullet,7\rangle$	
5	$C_2 \Downarrow$	
	$X \in ll\langle\{1,\mathbf{3,4}\},\bullet,7\rangle$	
6	$C_4 \Downarrow$	
	$X \in ll\langle\{1,3,\mathbf{5}\},\bullet,7\rangle$	
7	$C_5 \Downarrow$	
	$X \in ll\langle\{1,3,\mathbf{6}\},\bullet,7\rangle$	
8	$C_6 \Downarrow$	
	$X \in ll\langle\{1,\mathbf{4,5}\},\bullet,7\rangle$	

$C_1(X) \equiv |X \cap \{1,3,6\}| = 1$, $C_2(X) \equiv |X \cap \{2,4,6\}| = 1$, $C_3(X) \equiv |X \cap \{2,5,7\}| = 1$,
$C_4(X) \equiv |X \cap \{3,4,7\}| = 1$, $C_5(X) \equiv |X \cap \{3,4,6\}| = 1$, and $C_6(X) \equiv |X \cap \{1,2,3\}| = 1$.

Figure 7.1: Effect on Propagation Order

search. More importantly, only a polynomial number (to the number of elements in the universe) of feasibility checks is required. Suppose the checker takes $O(\alpha)$ time, a binary generic propagator makes $O(c^2 \log n)$ calls to the checker, making a total time complexity of $O(\alpha c^2 \log n)$. *In other words, if the feasibility checker runs in polynomial time, the bound-consistent propagator also runs in polynomial time. Moreover, since bound consistency in the length-lex domain determines feasibility, the propagator is thus at least as hard as the feasibility checker. Checking the feasibility is the core component of a propagator. Hence we focus our discussion of intractability in feasibility checkers.*

The first result we mentioned is well-known but quite interesting and concerns the subset-bound domain.

Theorem 12. $hs_{sbc}\langle|X_i \cap X_j| \leq 1, \forall i < j\rangle$ is NP-hard. [5]

We consider a special case of this constraint in which all but one variables are bounded. We show that, even in this simple unary case, enforcing bound consistency on both the sbc-domain and the ll-domain is fixed-parameter intractable.

Definition 21 (atmost1)**.** $atmost1(\{s_1,..,s_m\},X) \equiv |X \cap s_i| \leq 1,\ \forall 1 \leq i \leq m$

Theorem 13. $hs_{sbc}\langle atmost1(\{s_1,..,s_m\},X)\rangle$ is NP-hard.

Proof. Reduction from k-Independent Set. Instance: Graph $G = (V,E)$ and a positive integer $k \leq |V|$. Question: Does G contains an independent set of size k, i.e. a k-subset V' of V such that no two vertices in V' join by an edge in E.

We construct an instance of CSP with one sbc-variable X and one constraint $atmost1(\{s_1,..,s_m\},X)$. Intuitively, X corresponds to a independent set and each set s_i corresponds to the neighborhood of vertex i and itself. Hence, X can take at most 1 element from each set corresponds to the restriction that no two vertices in the independent set join by an edge.

Formally, for every $i \in V$, $s_i = \{i\} \cup adj(i)$ (where $adj(i)$ denotes the neighborhood of vertex i), and $X \in sbc\langle\emptyset,V,k,k\rangle$. The CSP has a solution if and only if G has a independent set of size k. $\Rightarrow$ Given a k independent set V', we can construct a solution by setting $X = V'$ since every element in X actually corresponds to a vertex. When X takes an element i, since the size of intersection is at most 1, it cannot take any other element from set s_i (i.e. $adj(i)$), the definition of independent set guarantees this. $\Leftarrow$ Given a consistent assignment of X, it is a independent set since any edge corresponds to taking two element from the same set which violates the $atmost1$ constraint. □

Reference [4] discusses a class of fixed-parameter tractable propagators which run in polynomial time when some of the parameters are fixed. The class is called fixed-parameter tractable (FPT) and its time complexity is bounded by $O(f(k)n^{O(1)})$ where $f(k)$ is an arbitrary function only depends on the parameter k. When k is fixed, $f(k)$ becomes a constant leaving the remaining $n^{O(1)}$ a polynomial. On the other hand, there is a class of propagators which is not FPT, meaning that even when the parameters are fixed, the feasibility routine still takes exponential time. Unary intersection constraints fall into this category.

Corollary 1. $hs_{sbc}\langle atmost1(\{s_1,..,s_m\},X)\rangle$ is $W[1]$-hard.

Proof. k-Independent Set is a $W[1]$-Complete problem.[15] □

Corollary 2. $hs_{ll}\langle atmost1(\{s_1,..,s_m\},X)\rangle$ is $W[1]$-hard.

Proof. For any sbc-domain that contains only all k-sets of some universe, there exists an equivalent ll-domain $sbc\langle\emptyset,V,k,k\rangle \equiv ll\langle\triangle_k,\bigtriangledown_k,|V|\rangle$ with $\triangle_k = \min_{\preceq}\{s \mid s \subseteq V \wedge |s| = k\}$ and $\bigtriangledown_k = \max_{\preceq}\{s \mid s \subseteq V \wedge |s| = k\}$. □

This result has an interesting corollary. Consider the propagation of a set of unary constraints of the form $|X \cap s_i| \leq 1$ $(1 \leq i \leq n)$. These constraints enjoy a polynomial-time bound-consistency algorithm in the length-lex domain. By definition of bound consistency, constraint propagation terminates in a failure or in a state where the bounds of the variable are solutions. Hence, by Corollary 2, constraint propagation cannot run in time $O(f(k)n^{O(1)})$ in the worst case.

Corollary 3. The propagation algorithm for a collection of $bc_{ll}\langle|X \cap s_i| \leq 1\rangle$ over X cannot run in time $O(f(k)n^{O(1)})$ in the worst case unless $FPT = W[1]$.

Similar results hold for other intersection constraints.

Definition 22 (exact1)**.** $exact1(\{s_1, .., s_m\}, X) \equiv |X \cap s_i| = 1,\ \forall 1 \leq i \leq m.$

Theorem 14. $hs_{sbc}\langle exact1(\{s_1, .., s_m\}, X)\rangle$ is NP-hard.

Proof. Reduction from 1-in-3 SAT. Instance: Set of n variables and m clauses, where each clauses consists of exactly three literals and each literal is either a variable or its negation. Question: Does there exist a truth assignment to variables such that each clause has exactly one true literal?

Given a instance of 1-in-3 SAT, we construct a CSP with a $exact1$ constraint. A set variable X associated with a sbc-domain $sbc\langle\emptyset, \{1, -1, .., n, -n\}, n, n\rangle$ corresponds to a truth assignment. $i \in X$ means variable i is true and vice versa. There are two types of sets. Set $s_i = \{i, -i\}$ $(1 \leq i \leq n)$ ensures a variable can either be true or false. Set $t_j = \{p, -q, r\}$ corresponds to a clause $(x_p \vee \neg x_q \vee x_r)$ guarantees that exactly one of its literal is true. Hence, we post the constraint $exact1\langle\{s_1, .., s_n, t_1, .., t_m\}, X\rangle$. Clearly, the input instance has feasible assignment if and only if the CSP has a solution. □

Definition 23 (atleast1)**.** $atleast1(\{s_1, .., s_m\}, X) \equiv |X \cap s_i| \geq 1,\ \forall 1 \leq i \leq m$

Theorem 15. $hs_{sbc}\langle atleast1(\{s_1, .., s_m\}, X)\rangle$ is NP-hard.

Proof. The hardness proof is essentially equivalent to that of Theorem 14 by changing the input instance to 3SAT. □

	Subset-Bound			Length-Lex		
(g,s,w)	Time	Fails	Ratio	Time	Fails	Ratio
(4,3,5)	0.11	968	**9132.08**	**0.04**	**136**	3578.95
(5,3,7)	25.63	157285	**6137.47**	**13.08**	**16361**	1251.03
(5,4,6)	337.35	2013980	**5969.95**	**198.16**	**259616**	1310.16
(5,5,4)	0.33	2320	**7138.46**	**0.15**	**175**	1166.67
(6,5,5)	**36.49**	195986	**5371.39**	45.76	**34375**	751.23
(8,5,6)	92.3	308195	**3339.02**	**65.76**	**20302**	308.74

Table 7.1: Social Golfer Problem: The Empirical Data Suggests the Length-Lex is Better.

7.3 Seemingly Contradicting Results Between Theory and Practice

As mentioned earlier, the potentially exponential behavior of constraint propagation was pointed out in [58] for knapsack constraints and similar results exist for continuous constraints and edge-finding algorithms for cumulative constraints. What is somewhat surprising here is the simplicity of the constraint involved, which are simple unary intersection constraints. *This abundance of negative theoretical results may lead researchers to conclude that the sbc-domain and, even more so, the ll-domain are unworthy of any consideration. Experimental results in Table 7.1 however clearly indicate otherwise.*

The length-lex model is significantly faster than the subset-bound model (except instance $(6, 5, 5)$ where they are competivie). The number of fail nodes for subset-bound is much higher. Consider the fails-to-time ratio, which is roughly the average number of constraint propagation algorithm completed per second, the constraint propagation for subset-bound is significantly faster than that of length-lex. It is mainly due to the fact that length-lex potentially takes exponential time in reaching the fixpoint in worst case. Comparing the fails-to-time ratio between two model, one may suggest that length-lex is a terrible representation.

Embrace the Complexity Indeed, we should view the problem from another perspective. The constraint propagation algorithm for the length-lex domain is tackling a much harder problem than that of the subset-bound domain. Figure 7.1 essentially illustrates a 3-SAT problem, which is intractable. The length-lex domain solves a NP-hard problem in the constraint propagation algorithm;

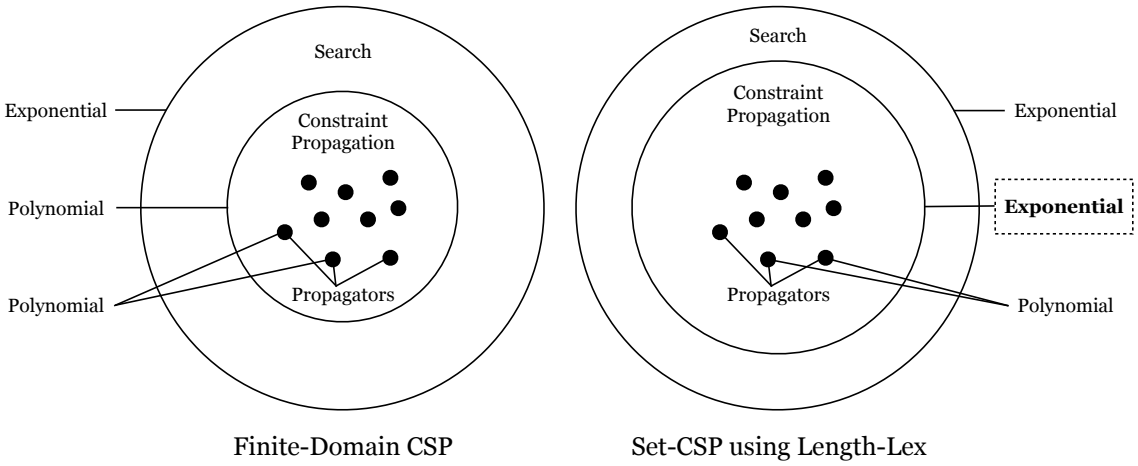

Figure 7.2: Constraint Propagation for Length-Lex is Exponential

whilst the subset-bound domain achieves *no propagation*.

This, perhaps, suggests why the length-lex representation visits dramatically smaller search tree than that of subset-bound, and why length-lex takes dramatically more time in each node as it is solving a harder problem.

Recall that the constraint propagation algorithm is the core of constraint programming. Propagators capture the problem semantics, and the propagation algorithm enables them to communicate through domains. This suggests us to put more emphasis on the constraint propagation than allowing the relatively agnostic search, which hardly exploits the problem semantics, to solve the problem.

Figure 7.2 presents the difference between finite-domain CSP, as well as the subset-bound domain, and the length-lex representation. The ultimate goal of using constraint programming is to solve a NP-hard problem, which is represented as the largest circle. We use propagators to specify relationship between variables, and the aggregated relationship is the solution to the problem. The constraint propagation algorithm provides a channel for communication, where inference takes part and infeasible values are removed from domains. When the constraint propagation algorithm is stuck and no further domain reduction is possible, the search kicks in, makes a guess and create a sub-problem, hopefully until a point which the problem is small enough for the propagators to find a solution.

Shifting the Exponential Behavior to Where We can Control. In the classical world, propagators and the constraint propagation algorithm usually run in polynomial time (of course, there are exceptions). The agnostic search does most of the dirty work and responsible for the exponential behavior. For the length-lex variables, despite most of the primitive unary constraints run in polynomial time, the constraint propagation algorithm potentially takes exponential time. The propagation algorithm, which is a mean of communication among constraints, exploits more problem semantics and lead to strong propagation. As we showed empirically, the length-lex domain visits a dramatically small search tree when comparing with the subset-bound domain.

This suggests that shifting some of the exponential behavior from the agnostic search to the constraint propagation algorithm is beneficial. It raises a question of whether we should make another step, which further shifting the exponential part to propagators where the problem semantics can be well-exploited. We examine this idea in the next section.

7.4 Exponential Filtering for Intersection Constraints

The previous sections reported intriguing theoretical and experimental results. The theory indicated that constraint propagation of even simple constraints may take exponential time in the worst case for the length-lex domain, while the experimental results clearly showed that the length-lex domain leads to the best and most robust performance despite of its high fails-to-time ratio. In this section, we reconsider the intractable unary intersection constraint. Instead of decomposing them into simpler unary constraints, we propose simple yet elegant exponential algorithms for enforcing bound consistency.

The goal is to move the potentially exponential behavior from the rather agnostic constraint propagation algorithm into the constraint itself in which the constraint semantics can be exploited. In short, we wants our propagator exponential, as shown in Figure 7.3.

Algorithm 14 implements $bc_{ll}\langle atmost1(\{s_1, ..., s_m\}, X)\rangle$ and is self-explanatory. The set $\mathcal{S}$ maintains a logical enumeration of all possible solutions. Both bounds of the length-lex domain are determined according to the bound consistency definition. Corollary 3 implies that there are no fixed-parameter tractable algorithm for Algorithm 14 since $hs_{ll}\langle\mathcal{C}\rangle$ is a special case for $bc_{ll}\langle\mathcal{C}\rangle$.

Theorem 16. Algorithm 14 runs in time $O(n^c mc)$ where $c = |u|$.

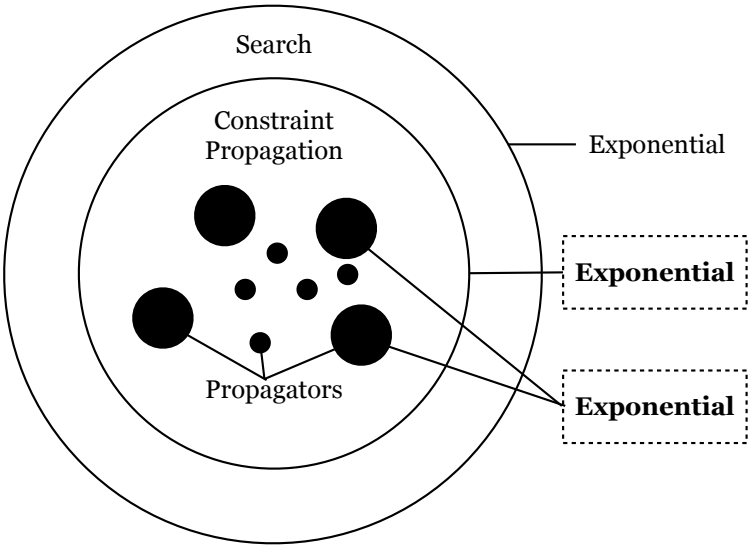

Figure 7.3: Embrace the Beauty of Exponential Propagator

Algorithm 14 $bc_{ll}\langle atmost1(\{s_1, .., s_m\})\rangle(X_{ll} = ll\langle l, u, n\rangle)$

1: $l' \leftarrow \min_{\preceq} \{s \in X_{ll} \mid \bigwedge_{1 \leq i \leq m} |s \cap s_i| \leq 1\}$
2: $u' \leftarrow \max_{\preceq} \{s \in X_{ll} \mid \bigwedge_{1 \leq i \leq m} |s \cap s_i| \leq 1\}$
3: **return** $ll\langle l', u', n\rangle$

Proof. X_{ll} contains at most $O(n^c)$ sets. Each set takes $O(mc)$ time to verify if it satisfies the constraint. □

Implementation Notes The exponential propagation does not explicitly enumerate all the sets. It relies on a technique on which most length-lex propagators rely. The inference reduces to a feasibility routine hs which takes a domain and returns a boolean value that indicates if the domain has any solution.

The idea is essentially equivalent to the generic algorithm for enforcing bound consistency for unary constraints. The only difference is that, since the feasibility routine is computationally expensive, unlike the linear time checking routine we discussed, the goal of the exponential filtering algorithm is to minimize the number of calls to the exponential feasibility routine. The length-lex bound can be seen as two arrays of finite domain variables (one for each bound), finding the bound is essentially equivalent to filling the two arrays. The array corresponds to the lower bound is the

least solution (according to the length-lex ordering) among all solutions. The least solution can be found by labeling the most significant position with the smallest element. The partial assignment is checked against the feasibility routine. If the routine returns true, meaning there is a solution with the current assignment, we label the second-most significant position. Otherwise, we try to label the second-smallest element to the most significant position, and invoke the feasibility routine. Such process continue until the bound is found. Finding the upper bound is essentially the same, we only label with the largest element instead of the smallest.

These tricks do not reduce the worst case time complexity since the feasibility routine runs in exponential time. However, from a practical standpoint, it dramatically improves the propagator's performance.

7.5 Evaluation

This section evaluates the performance of the proposed exponential propagators. The goal of it is to reduce the time spent by the constraint propagation algorithm which repeatedly iterates among all the binary intersection propagators in the model. We show that most benchmark instances used in the last section becomes trivial under the new exponential constraints. Results for larger and more difficult instances are shown.

Consider Figure 6.12, the COMET model for social golfer problem using dual modeling. Lines 9–10 are the *atmost*1 constraints which guarantee that no two players play more than once. They are the main culprits of making the constraint propagation runs forever. We propose another model with exponential propagators are applied to reduce the exponential run time. In particular, we substitute the binary *atmost*1 constraint with the following propagator.

```
9  forall (wi in Weeks, gi in Groups)
10   cp.post( atmost1( llx[wi,gi], all(wj in Weeks, gj in Groups : wj < wi) llx[wj,gj] ) );
```

The first argument is the length-lex variable to be propagated. The second argument corresponds to the array of sets s_i which defines the constraint (refer to Definition 21). Variables `llx[wj,gj]` are not considered in the propagator until it becomes a singleton. When the variable is bound, it's value is added to the array of sets. Since the search uses a vanilla week-wise labeling strategy, `wj < wi` is sufficient. Binary constraints are removed from the model and the exponential constraint

Domain	Subset-Bound		Length-Lex		Length-Lex	
Symmetry Breaking (Chapter 6)	✔		✔		✔	
Exponential Propagator ($bc_{ll}\langle atmost1\rangle$)					✔	
(g,s,w)	Time	Fails	Time	Fails	Time	Fails
(3,3,3)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(4,3,5)	0.11	968	0.04	**136**	**0.03**	140
(5,3,7)	25.63	157285	13.08	**16361**	**4.56**	19149
(5,4,6)	337.35	2013980	198.16	**259616**	**61.06**	286792
(6,5,5)	36.49	195986	45.76	**34375**	**7.72**	36017
(6,5,6)	314.24	1512264	433.62	**214075**	**57.47**	221033
(6,6,4)	x	x	x	x	**592.46**	**2049826**
(7,7,4)	x	x	50.96	**1739**	**0.78**	**1739**
(8,3,10)	1390.82	3782741	1062.72	572773	**119.06**	**542539**
(8,5,6)	92.3	308195	65.76	**20302**	**5.01**	**20302**
(9,5,7)	x	x	x	x	**222.2**	**730635**
(11,8,3)	53.54	230740	601.58	**3264**	**2.01**	**3264**

Table 7.2: Social Golfer Problem: Exponential Constraints that Speeds Up Convergence.

propagates only when variables are bound, less propagation is achieved. Nevertheless, we show that the gain in performance outweighs lost in propagation.

Table 7.2 reports the experimental results on the exponential *atmost*1 propagator. The model using the exponential propagator is the fastest. Comparing with the original length-lex model, it dramatically reduces the search time by orders of magnitude despite it visits a slightly larger search tree. It solves some problems which were too large to be solved within the time limit. In instance $(11, 8, 3)$, the solving time reduces from almost 10 minutes to 2 seconds, while both models visit the same search tree. The improvement in time is only caused by the application of the exponential propagator, which enables the constraint propagation algorithm to reach fixpoint in a relatively shorter time.

Table 7.3 studies the fails-to-time ratio of each model. The ratio, in a loose sense, represents the number of nodes visited per second. In other words, the number of constraint propagation algorithm invoked, which is the inverse of the average time spent by each algorithm. The exponential length-lex model speeds up the constraint propagation algorithm by a few times. While the previous table reveals that not many propagations are lost when the binary constraint is substituted by the unary exponential constraint, the difference in ratio suggests that it is highly beneficial to use the

Domain	Subset-Bound	Length-Lex	Length-Lex
Symmetry Breaking (Chapter 6)	✔	✔	✔
Exponential Propagator ($bc_{ll}\langle atmost1\rangle$)			✔
(g,s,w)	$\frac{\text{Fails}}{\text{Time}}$	$\frac{\text{Fails}}{\text{Time}}$	$\frac{\text{Fails}}{\text{Time}}$
(3,3,3)	0	0	0
(4,3,5)	9132.08	3578.95	5000
(5,3,7)	6137.47	1251.03	4196.58
(5,4,6)	5969.95	1310.16	4696.89
(6,5,5)	5371.39	751.23	4664.81
(6,5,6)	4812.4	493.69	3845.92
(6,6,4)	x	x	3459.86
(7,7,4)	x	34.13	2229.49
(8,3,10)	2719.78	538.97	4557.05
(8,5,6)	3339.02	308.74	4051.49
(9,5,7)	x	x	3288.19
(11,8,3)	4309.59	5.43	1622.27

Table 7.3: Social Golfer Problem: Fails-to-Time Ratio of Exponential Propagator

exponential propagator. It enables the solver to visit a few times more nodes per unit of time.

The exponential propagators proposed in this paper plays the role in accelerating the convergence of the constraint propagation algorithm. The introduction of exponential propagators for length-lex variables has no impact on propagation. The absence of binary length-lex constraint, which is explained earlier, accounts for the slight difference. And the difference is outweighed completely by the performance gain. The fails-to-time ratio reveals that the performance for the exponential model is very competitive with the subset-bound model, while length-lex allows a dramatically stronger propagation.

7.6 Related Work

This section briefly reviews some related work on exponential propagation and propagators. Perhaps the closest related work is the work on box consistency in the Numerica system [66]. The key idea of box consistency was to avoid the decomposition of a complex constraints into elementary ternary constraints. By enforcing box consistency on the original constraint, these systems improve the pruning, addresses the so-called dependency effect of interval propagation, and tackle the fact

	Subset-Bound (and its variants)	**Length-Lex**	ROBDD
Propagation	Loose	**Strong**	Very Precise
Space	$O(n)$	**O(c)**	Potentially Exponential
Efficiency	Fast	**O(poly(c))**	Potentially Slow
Convergence	Fast	**Fast**	Potentially Slow

Figure 7.4: Comparison over Different Set Domain Representations. (n is the universe size, c is the cardinality upper bound.)

that the fixpoint algorithm can take a long time to converge. Box consistency was enforced by a potentially exponential algorithm. The Newton and Numerica systems also include conditions to terminate the fixpoint algorithm prematurely when the propagation was not reducing the search space enough. Lebbah and Lhomme [47] considered the use of extrapolation methods to speed up the convergence of filtering algorithms for continuous CSPs, also dramatically the efficiency on these problems. These techniques could potentially be applied to set domains as well, but this paper took another, simpler, route: Using exponential propagators that have a more global view of the problem at hand. Also closely related is the work of Bessiere and Régin on solving CSPs on the fly. They recognize that, on certain applications, the pruning offered by the solver was not strong enough. They isolated a global constraint (i.e., the sum of n variables taking different values) for which they did not design a specific propagator. Instead, they use the CP solver recursively and solved CSPs on the fly to enforce arc consistency. Once again, the result is to move some of the exponential behavior from the search to the constraint propagation. Note also that several pseudo-polynomial algorithms have also been proposed in the past, including the well-known filtering algorithm for knapsack constraints [65].

7.7 Conclusion

Most research in constraint programming focuses on designing polynomial-time filtering algorithms. This section explored, for set CSPs, the idea of shifting some of the exponential behavior from the search component to the filtering component, and from the constraint-propagation algorithm to the propagators. More importantly, it presented exponential-time propagators for intractable unary intersection constraints and demonstrated that they bring considerable performance improvement by

speeding up constraint propagation. They indicate that it may sometimes be beneficial to embrace complexity in the filtering component and exploit the constraint semantics and locality, instead of relying on rather agnostic search and constraint propagation algorithms.

Chapter 8

Global Set Intersection Constraints

8.1 Overview

Global set constraints have received very little attention primarily because of intractability results on both bound consistency and feasibility checking. However, they still offer significant opportunities for improving the performance of set solvers, since the alternative, i.e., not to prune the search space, seems even worse. Recent work explores two possible approaches to deal with these computational difficulties. On the one hand, one may relax the requirement for polynomial-time algorithms and settle for algorithms that may be exponential in the worst case but are reasonable in practice and prune substantial parts of the search tree. On the other hand, one may take the more conventional approach and relax completeness of the filtering algorithm.

This chapter explores both approaches for several global intersection constraints. It has three main contributions, all of which are independent of the underlying representations of the set solver:

1. it introduces a feasibility checker for global *alldisjoint* constraint for an explicit set domain representation;

2. it presents a dual filter for the global *atmost-k* constraint that constrains the cardinalities of the dual variables;

3. it introduces primal/dual filters for the combination of a global *atmost-k* constraint and

symmetry-breaking constraints.

The dual and primal/dual filters for *atmost-k* constraint are particularly compelling. They depend on the solutions of some combinatorial problems which are themselves set-CSPs. In turn, these CSPs can be solved by constraint programs using the dual filter, which again depends on the solution of some smaller combinatorial problems which are solved recursively by constraint programming.

Experimental results show that these contributions are orthogonal and may substantially improve the performance of set solvers on some standard benchmarks, solving instances that could not be solved in reasonable time before and reducing CPU times by factors that exceeds 1,000.

The rest of the chapter presents the four contributions, reports the experimental results, review related work, and concludes.

8.2 A Feasibility Checker for The AllDisjoint Constraint

This section presents a feasibility checker for the *alldisjoint* constraint.

Definition 24 (The AllDisjoint Constraint)**.** $alldisjoint(X_1, ..., X_m) \equiv \bigwedge_{i<j} X_i \cap X_j = \emptyset$.

If the set domains are given explicitly, checking feasibility is NP-hard. Theorem 20 gives a similar result for the hybrid ls-domain.

Theorem 17. $hs\langle alldisjoint\rangle(X_1, ..., X_m)$ is NP-hard when $d(X_i)$ is specified as an explicit set of sets.

Proof. A trivial reduction from the SetPacking problem. Instance: a finite set S and a collection $\mathcal{S}$ of subset of S. Question: determine whether some m sets in $\mathcal{S}$ are pairwise disjoint. A solution to the problem is $\mathcal{S}' \subseteq \mathcal{S}$, where $|\mathcal{S}'| = m$ and sets in $\mathcal{S}'$ are pairwise disjoint.

We first assume all sets in $\mathcal{S}$ are not empty. Since otherwise, we can reduce the parameter m by the number of empty sets in $\mathcal{S}$. Given a SetPacking instance, we construct a set-CSP such that it is feasible if and only if there exists m pairwise disjoint sets. In the CSP, there are m set variables with initial domain $\mathcal{S}$, and a $alldisjoint(X_1, ..., X_m)$ constraint. Intuitively, the variables correspond to the SetPacking solution. The rewriting is obviously polynomial.

Algorithm 15 $hs\langle alldisjoint\rangle(X_1, ..., X_m)$

```
1: for σ in {[v_1, ..., v_n] | v_e ∈ {1, ..., m} ∪ {⊥}} do
2:    [T_1, ..., T_m, T_⊥] ← [D(X_1), ..., D(X_m), {{1..n}}]
3:    for e = 1 to n do
4:       T_σ(e) ← {t ∈ T_σ(e) | e ∈ t}
5:       for i in {1, 2, ..., σ(e) − 1, σ(e) + 1, ..., m} do
6:          T_i ← {t ∈ T_i | e ∉ t}
7:    if ⋀_{1≤i≤m} T_i ≠ ∅ then
8:       return true
9: return false
```

$\Rightarrow$ Given a solution to the SetPacking problem, we construct a solution to the set-CSP. Let $\mathcal{S}' = \{s_1, ..., s_m\}$, we assign $X_i = s_i$. Since X_i has a initial domain of $\mathcal{S}$, s_i is a feasible domain value. The assignment also satisfies the *alldisjoint* constraint, since $\mathcal{S}'$ are pairwise disjoint.

$\Leftarrow$ Given a solution to the set-CSP, we construct a solution to the SetPacking Problem. Consider a solution $[X_1 = s_1, ..., X_m = s_m]$, every pair of s_i are pairwise disjoint, and $s_i \in \mathcal{S}$. Moreover, as all s_i are non-empty sets, we have $s_i \neq s_j \quad \forall i < j$ since they are disjoint. Therefore, $\mathcal{S}' = \{s_1, ..., s_m\}$ is a solution to the SetPacking Problem. □

8.2.1 The Feasibility Checker

Algorithm 15 is a feasibility checker for the *alldisjoint* constraint, assuming that the set variables take their elements in $\{1..n\}$. In the worst case, the checker takes exponential time but experimental results will demonstrate that it can bring substantial benefits in practice. The checker takes a set of set variables and returns a boolean value indicating whether there are solutions. Its key idea is to enumerate all the dual assignment (Line 1) and to test whether they satisfy the domain constraints (Lines 2–8). Since an element can be assigned to at most one set, a dual assignment assigns a variable index v_e to each element e (or $\perp$ if the element is not assigned to any set). To test whether a dual assignment is feasible, the checker maintains T_i to denote the feasible sets for variable X_i. Initially, T_i is initialized to $D(X_i)$. The dual assignment is then used to filter the T_i's. In particular, the checker considers each element e in turn (Line 3) and removes from $T_{\sigma(e)}$ all the sets not containing e. In other words, $X_{\sigma(e)}$ is the variable e is assigned to and the checker prunes the domain of $X_{\sigma(e)}$ to ensure that they all contain e. It then prunes the domains of the other variables (Lines 5–6) to make sure that they do not contain e. The checker returns true if no domain has become empty at

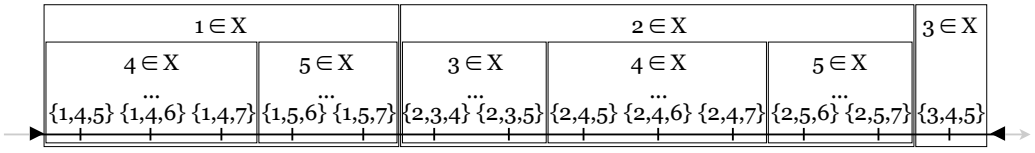

Figure 8.1: The Explicit Domain List has No Hole.

the end of the computation (Lines 7–8). If none of the dual permutations is a solution, the checker returns false (Line 9). Observe that set $T_\perp$ is never pruned, since it contains the set of all elements initially. Line 4 can never remove its set and lines 5–6 never considers $T_\perp$.

Example 40. Consider the domains $D(X_1) = \big\{\{1,2\},\{1,4\},\{2,4,6\}\big\}$, $D(X_2) = \big\{\{1,2\},\{2,5\},\{2,6\}\big\}$, $D(X_3) = \big\{\{1,5\},\{3\},\{5\}\big\}$, and $\sigma = [1,2,\perp,1,3,2]$. The dual assignment assigns element 1 to variable 1. The algorithm removes domain values from $T_1,\ldots,T_3$, giving $T_1 = \big\{\{1,2\},\{1,4\}\big\}$, $T_2 = \big\{\{2,5\},\{2,6\}\big\}$, and $T_3 = \big\{\{3\},\{5\}\big\}$. The same domain-reduction process is performed for all elements. At the end, $T_1 = \big\{\{1,4\}\big\}$, $T_2 = \big\{\{2,6\}\big\}$, and $T_3 = \big\{\{5\}\big\}$. Hence, the dual assignment is a solution. On the other hand, if the initial value of T_3 is $\big\{\{1,5\},\{3\}\big\}$, it will become empty after processing element 3. In this case, the dual assignment is infeasible. $\diamond$

Implementation Notes The technical insight behind the checker is that the T_i's are only a logical copy of the domain values, the actual explicit list of sets are *not* copied. We assume the input domains are lexicographically sorted. The domains are always consecutive throughout the domain reduction loop in lines 3–6. The checker only remembers the position of the first and last set. Figure 8.1 presents the idea. Suppose the sets on the line is the initial input domain. The checker marks the set $\{1,4,5\}$ as its starting point and the set $\{3,4,5\}$ as its ending point. When the checker labels $1 \in X$, all it needs is to removes all the sets in the back, by updating the ending set to $\{1,5,7\}$. On the other hand, when the checker labels $1 \notin X$, the starting point is upated to $\{2,3,4\}$. The same process repeats for all the elements. The key insight is that the check labels the variables in the same way it sorts the domain values, making it possible to represent the running-domain by keeping only the start and end values.

	Length-Lex			
Sym-Break (Chapter 6)	✔		✔	
Exponential ($bc_{ll}\langle atmost1\rangle$)	✔		✔	
Checker ($hs\langle alldisjoint\rangle$)			✔	
(g,s,w)	Time	Fails	Time	Fails
(5,3,7)	**4.56**	19149	5.07	**12211**
(5,4,6)	61.06	286792	**39.7**	**120438**
(6,5,5)	7.72	36017	**2.44**	**4877**
(6,5,6)	57.47	221033	**16.38**	**27545**
(6,6,4)	**592.46**	2049826	646.62	**1890962**
(7,3,9)	x	x	**1276.05**	**2837356**
(8,3,10)	119.06	542539	**47.01**	**88817**
(9,3,11)	14.45	61924	**2.05**	**2724**

Table 8.1: Social Golfer Problem: AllDisjoint Checker for Length-Lex Domain.

8.2.2 Evaluation

The *alldisjoint* global constraint expresses that all groups of the same week are disjoint. The *atmost1* unary constraint generates an list of domain values for every primal variable, and our *alldisjoint* feasibility checker uses such list (but is only applied if the domain size is no greater than 200). Such a *alldisjoint* constraint is posed for each week and is propagated at the end of every choice point. In COMET, it is very easy to make sure the checker runs only once every choice point, all we need is to modify the search component and run the checker after labeling. Following is the code, Lines 53–55 are code segment in which the checker take place. Each checker corresponds to a week, and we do not check weeks before 3 since they are not very constrained.

```
44 }using{
45   forall (pi in 1..g*s) cp.post(requiresValue(sbx[1,(pi-1)/s+1],pi));
46   forall (si in 1..s) cp.post(requiresValue(sbx[2,1],(si-1)*s+1));
47   forall (wi in 2..w)
48     while (or(gi in 1..g)(!sbx[wi,gi].bound()))
49       selectMin(gi in 1..g) ( sbx[wi,gi].getRequiredSet().getSize(), gi ){
50         selectMin(pi in 1..g*s: !sbx[wi,gi].isExcluded(pi) && !sbx[wi,gi].isRequired(pi) )(pi)
51           try<cp> cp.post(requiresValue(sbx[wi,gi],pi));
52           | cp.post(excludesValue(sbx[wi,gi],pi));
53         forall (wj in 4..w)
54           if (!alldisjoint[wj].hs())
55             cp.fail();
56     }
57 }
```

We evaluate the performance of the exponential checker on length-lex domain. (Readers may refer to later chapters for evaluation on other domains) The checker dramatically reduces the search tree size. For example, for instance $(6, 5, 6)$, the number of fails is reduced by more than 8 times. Some previously out of reach instances, such as $(7, 3, 9)$, are now solved. The checker is also very robust too, it reduces the run time for most instances. For the instances it doesn't perform well, e.g. $(5, 3, 7)$ and $(6, 6, 4)$, it only slightly dampens the solver.

8.3 A Dual Filter for The Global Atmost-k Constraint

This section discusses the global *atmost-k* constraint which guarantees that every pair of set variables shares at most k elements. It is at least as difficult as the global disjoint constraint since the latter is a special case where $k = 0$.

Definition 25 (atmost-k)**.** $atmost(k, X_1, ..., X_m) \equiv \bigwedge_{i<j} |X_i \cap X_j| \leq k$.

Early versions of the following theorem appeared in [6].

Theorem 18. $hs\langle atmost(k)\rangle(X_1, ..., X_m)$, where X_i are subset-bound, length-lex, or set variables with finite domains, is NP-hard.

We now present a dual filter for the global *atmost-k* constraint. For simplicity, we assume that all variables are of the same cardinality c.

Intuition The key idea behind the dual filter is to consider the possible elements for the sets (dual view) and answer the following two questions:

1. How many set variables can take an element e?
2. How many set variables can exclude element e?

Example 41 (Dual View)**.** Consider the case of 7 set variables of cardinality 3 drawing their elements from a universe of size 7 and subject to a global *atmost-1* constraint. We aim at determining how many set variables can take an element $e \in \{1..7\}$? Figure 8.2 illustrates the basic idea. Each row corresponds to a variable and each column an element. The symbol x on cell $(X_2, 4)$ denotes $4 \in X_2$.

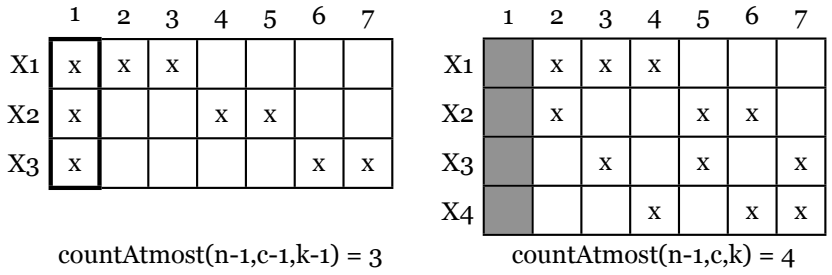

Figure 8.2: How Many Set Variables Can Take or Exclude a Value? ($n = 7, c = 3, k = 1$)

The left part of the picture illustrates how to compute the maximum number of set variables which can *take element 1*. Since all set variables take element 1, the remaining elements should be mutually disjoint. There can be at most 3 disjoint set of cardinality 2 taking elements from a universe of size 6. Hence, element 1 can occur in *at most* 3 set variables.

The right part of the picture illustrates how to compute the maximum number of set variables which can *exclude element 1*. This reduces to a similar atmost-k constraint in which the universe size is reduced by 1 . The maximum number of sets excluding element 1 is 4 (we will discuss how to compute this number shortly). In other words, element 1 has to occur in *at least* 7-4 = 3 variables.

As a consequence, we state a dual constraint requiring that element 1 appears in exactly 3 set variables. The same reasoning in fact applies to all elements. ◇

The Dual Filter The basic idea underlying the dual filter is to state a redundant dual model. The dual model assumes the existence of a function *countAtmost(n,c,k)* defined as follows.

Definition 26 (countAtmost)**.** Function *countAtmost(n,c,k)* returns the maximum number of sets of cardinality c taking their values in $\{1..n\}$ and sharing at most k values.

The dual filter is depicted in Figure 8.3. Its key idea is to impose a lower and upper bound for the occurrence of each element e in the universe. Line (8.1) defines the dual variables: Y_e represents the indices of set variables which include element e. Line (8.2) defines the channeling constraints between the primal and dual variables. Line (8.3) defines the upper bound on the cardinality of Y_e as $countAtmost(n-1, c-1, k-1)$. Indeed, consider the set of variables taking element e, each of them has at most $c-1$ free positions, which must be filled by elements drawn from a universe of size $n-1$. To satisfy the intersection constraint, each pair can share at most $k-1$ other elements since they

$$Y_e \subseteq \{1, ..., n\} \qquad \forall 1 \leq e \leq m \tag{8.1}$$
$$e \in X_i \Leftrightarrow i \in Y_e \qquad \forall 1 \leq i \leq n, 1 \leq e \leq m \tag{8.2}$$
$$|Y_e| \leq \mathit{countAtmost}(n-1, c-1, k-1) \qquad \forall 1 \leq e \leq m \tag{8.3}$$
$$m - \mathit{countAtmost}(n-1, c, k) \leq |Y_e| \qquad \forall 1 \leq e \leq m \tag{8.4}$$
$$\sum_{1 \leq e \leq m} |Y_e| = m\ c \tag{8.5}$$

Figure 8.3: The Redundant Dual Filter for $\mathit{atmost}(k, X_1, ..., X_m)$.

are already sharing e. Hence, the maximum cardinality is bound by $\mathit{countAtmost}(n-1, c-1, k-1)$. Line (8.4) defines the lower bound on the cardinality of Y_e as $m - \mathit{countAtmost}(n-1, c, k)$. Indeed, consider the set of variables not taking element e. These variables must draw elements from a universe of size $n-1$, from which they have to pick c elements and each pair of variables can share at most k elements. The maximum number of variables not taking element e is therefore bound by $\mathit{countAtmost}(n-1, c, k)$ and element e has to occur in at least $m - \mathit{countAtmost}_m(n-1, c, k)$ variables. Finally, Line (8.5) ensures that the sum of the cardinalities is equal to $m \times c$, i.e., the number of variables multiplied by their cardinalities.[1]

Observe that the dual filter is independent of the representation of set variables, which makes it widely applicable. Salder and Gervet [55] presented a special case of this dual filter but only considered the *atmost-1* constraint and the upper bound. This section generalized the idea to *atmost-k* constraint and the lower bound, which complicates significantly the implementation.

Implementation of countAtmost It remains to discuss how to implement function *countAtmost*. There are at least three possibilities:

1. when available, it can be a lookup from a combinatorics table [9];
2. it can be a constant-time approximation using extremal set theory [40];
3. it can be implemented as an optimization problem!

[1]For different cardinalities, we can simply replace c by the minimum cardinality of all variables, since this gives conservative calls to the *countAtmost* function.

For the second case, let $s_1, ..., s_m$ be sets of cardinality c and n be their union size. If $\forall 1 \leq i < j \leq m, |s_i \cap s_j| \leq k$, then

$$n \geq \frac{c^2 m}{c + (m-1)k}.$$

This inequality can be used to obtain an upper bound on m.

Our implementation views the implementation of *countAtmost* as an optimization problem which can be specified as

$$\begin{aligned} &\texttt{maximize } m \texttt{ s.t.} \\ &|X_i \cap X_j| \leq k \qquad \forall 1 \leq i < j \leq m \\ &|X_i| = c \qquad \forall 1 \leq i \leq m \\ &X_i \subseteq \{1, ..., n\} \qquad \forall 1 \leq i \leq m \end{aligned}$$

This optimization problem can be solved by a sequence of feasibility problems using various values for m. As a result, *countAtmost* itself can be implemented in terms of set-CSPs. Moreover, these set-CSPs also use a global *atmost-k* constraint and hence they can use all the filters presented in this paper. In particular, our implementation posts the dual filter shown in Figure 8.3 which obviously depends on the values $countAtmost(n-1, c-1, k-1)$ and $countAtmost(n-1, c, k)$. These are computed recursively as two additional optimization problems. Since these recursive calls may involve the same sub-optimization problems, our implementation memoizes the result of each suboptimization and reuses them whenever appropriate in order to avoid solving the same suboptimizations repeatedly.

The computation of these subproblems takes negligible time in our benchmarks and only takes place at the root of the tree. It is however interesting to see how the derivation of the dual filter requires the solving of set-CSPs which in turn uses the dual filter itself on smaller subproblems.

8.4 Primal/Dual Filters for Symmetry-Breaking Atmost-k

Section 6.3 presented a primal filter for the symmetry-breaking *alldisjoint*. It recognized that the most significant element determines the size of the possible sets for a variable and the lexicographically greater variables, enabling to achieve stronger propagation. Section 8.3 on the other hand presented a dual filter based on a dual model: It exploits the observation that an element cannot appear in, or be excluded from, too many variables, which imposes some strong cardinality constraint on dual variables. These ideas can be combined for the implementation of a global *atmost-k*$_{\preceq}$ constraint, which combines a global *atmost-k* constraint and a chain of symmetry-breaking constraints.

Definition 27 (Symmetry-Breaking Atmost-k)**.** $atmost_{\preceq}(k, X_1, ..., X_m) \equiv atmost(k, X_1, ..., X_m) \wedge \bigwedge_{i<j} X_i \preceq X_j$.

Intuition The primal/dual filter aims at answering the following questions which combines primal and dual aspects:

1. How many set variables must include the first e elements of the universe?
2. How many set variables must exclude the first e elements of the universe?

In general, variables that are greater lexicographically do not take small elements: These are taken by the lexicographically smaller variables. For the symmetry-breaking *alldisjoint* constraint, it was relatively easy to answer that question since every element can be taken by at most one variable. For the symmetry-breaking *atmost-k* constraint, this situation is more complicated but we can reuse the function *countAtmost* introduced for the dual filter.

Example 42 (Primal/Dual Exclusion)**.** Consider 5 set variables $X_1, ..., X_5$ of cardinality 3 taking their values from a universe 1..7 and a global $atmost_{\preceq}(1, X_1, ..., X_5)$. Since $countAtmost(6, 3, 1)$ returns 4, it follows that at most 4 variables can start with elements greater than or equals to 2. Due to the lexicographic constraint, X_1 must not start with element 2. ◇

Example 43 (Primal/Dual Inclusion)**.** Consider 5 variables $X_1, ..., X_5$ of cardinality 3 taking their values from {1..7} and a global $atmost_{\preceq}(1, X_1, ..., X_5)$. There are at most 3 variables taking element 1 (see Figure 8.2). Hence, X_4 must start with element greater than 1 and we can post the constraint $\{2, 3, 4\} \preceq X_4$. ◇

Reduction Rules We are now ready to present the two primal/dual reduction rules. The first rule reasons about the maximum number of variables that can exclude the first e elements and derives a constraint preventing early variables from starting with large values.

Rule 2 (Symmetry-Breaking Atmost-k: Exclusion)**.**

$$\frac{1 \leq e \leq n-c \wedge \bigwedge_{i \leq j \leq m} |X_j| = c \wedge i = \mathit{countAtmost}(n-e, c, k) \wedge 1 \leq i \leq m}{\mathit{atmost}_{\preceq}(k, X_1, ..., X_m) \longmapsto \min(X_{m-i}) \leq e \wedge \mathit{atmost}_{\preceq}(k, X_1, ..., X_m)}$$

When the length-lex representation is used for set variables, the derived constraint can be used to update the upper bound of the set variables: only the sets starting with an element no greater than e are left in the domain.

The second rule reasons about the maximum variables that can take the first e elements and derives a constraint preventing late variables from taking the first e elements.

Rule 3 (Symmetry-Breaking Atmost-k: Inclusion)**.**

$$\frac{1 \leq e \leq k \wedge \bigwedge_{i \leq j \leq m} |X_j| = c \wedge i = \mathit{countAtmost}(n-e, c-e, k-e) \wedge 0 \leq i < m}{\mathit{atmost}_{\preceq}(k, X_1, ..., X_m) \longmapsto l \preceq X_{i+1} \wedge \mathit{atmost}_{\preceq}(k, X_1, ..., X_m)}$$

where $l = \{1, ..., e-1\} \uplus \{e+1, ..., c+1\}$

When the length-lex representation is used for set variables, the derived constraint can be used to update the lower bound of the set variables which must become at least $l = \{1, ..., e-1\} \uplus \{e+1, ..., c+1\}$. Observe that the rule prevents X_{i+1} from taking *all* elements in $\{1, .., e\}$. The smallest set lexicographically not taking all elements in e starts with $\langle 1, ..., e-1 \rangle$, excludes e, and fills the remaining free slots with as small elements as possible, i.e., $\langle e+1, e+2, \ldots, c+1 \rangle$.

8.5 Related Work

Many set-CSPs exhibit *variable interchangeability*: given any solution, it is possible to generate another by swapping the assignment of two interchangeable variables. Ideally these symmetries should be eliminated to prevent the solver from visiting symmetric subtrees. Let X_1 and X_2 be

two interchangeable set-variables. If $[X_1 = \{1,2\}, X_2 = \{1,3\}]$ is a solution, then the assignment $[X_1 = \{1,3\}, X_2 = \{1,2\}]$ is a symmetric solution. To eliminate such symmetric solutions, the model can post a static ordering constraint $X_1 \preceq X_2$. The choice of the ordering constraint $\preceq$ is arbitrary. Two common orderings are the lexicographical [22] and length-lex [29] orderings which coincide when sets have the same length. This paper uses the lexicographical ordering for breaking symmetries but obviously the underlying domain representation can be subset-bound, length-lex, or BDD-based.

There is considerable work on symmetry breaking in constraint programming and this section only reviews directly relevant work. Crawford et. al. [11] introduces a light-weight static method for eliminating symmetric solutions using predicate constraints. The models over set variables almost always impose static lexicographic constraints to break variable symmetries.

The complexity of global intersection constraints over sets was investigated in depth in [6]. The paper showed that even feasibility checking is hard for global set constraints under some established domain representations. Exponential-time algorithms for set constraints were used by Yip and Van Hentenryck [74] to enforce bound consistency on unary intersection constraints, showing significant improvements in performance. This paper proposes an exponential-time feasibility checker for *alldisjoint*.

The idea of pushing symmetry-breaking constraints into other constraints has appeared in various papers. Hnich, Kiziltan, and Walsh [37] proposed a global constraint that combines symmetry breaking with a sum constraint. Katsirelos, Narodytska, and Walsh [41] proposed a generic framework for global constraint with symmetry-breaking constraints for vectors of variables. Yip and Van Hentenryck [73] proposed a generic framework for combining arbitrary binary length-lex propagators with ordering constraints and studied their benefits experimentally. This paper studies combination of the *alldisjoint* and *atmost-k* global constraints with a chain of symmetry-breaking constraints.

Salder and Gervet [55] proposed a filter for the global *atmost-1* constraint, restricting how many sets can share an element. This paper significantly generalizes this idea to produce both lower and upper bounds on dual variables of *atmost-k* constraints. Hawkins, Lagoon, and Stuckey [34] proposed a BDD-based representation of sets, which represents both domain and global constraints. Combining propagators is achieved by combining BDDs. This paper implicitly combines *atmost-k* and *alldisjoint* constraints.

8.6 Conclusion

This paper studied feasibility checking and filtering for global constraints over set variables. It proposed an exponential-time feasibility checker for the *alldisjoint* constraint, by taking a dual perspective and enumerating all possible dual assignments. The paper also presented dual, and primal/dual filters for the *atmost-k* and the symmetry-breaking *atmost-k* constraints. The dual and primal/dual filters need to answer various counting problems (e.g., How many set variables must include/exclude the first e elements of the universe) which are viewed as optimization problems and solved using the filters recursively on smaller *atmost-k* constraints. Experimental results on the standard benchmark problem, the social golfer problem, show that the feasibility checker and the filters are very effective and significantly improve state-of-the-art results on these problems. In particular, they are able to solve open instances for set representations and reduce CPU times by a factor greater than 20 on some instances.

Chapter 9

Hybrid Domain Representation

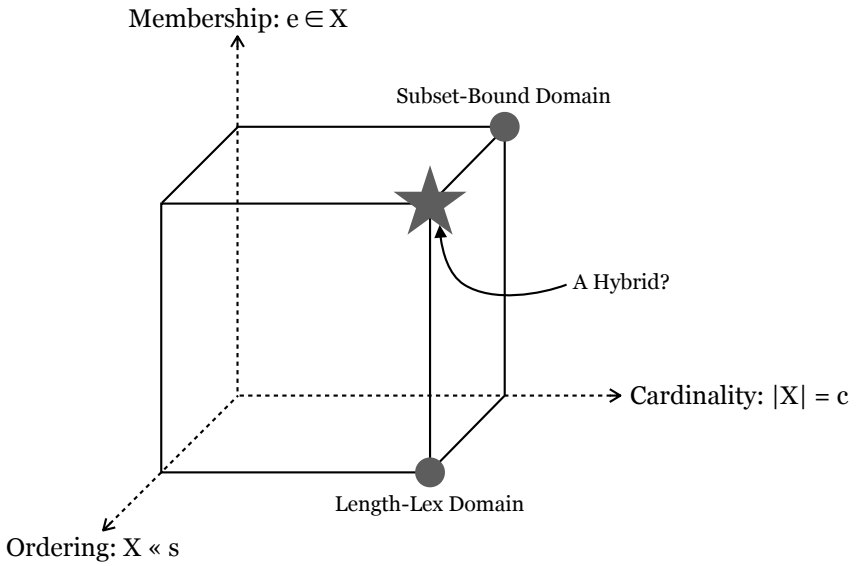

Figure 9.1: A Hybrid Domain Combining the Best of the Two Worlds?

9.1 Overview

In the previous chapter, we argue that the reason of why the length-lex domain representation is more effective in pruning the search space than the subset-bound representation is that it is possible

to prune a *value*, as oppose to an *element*, from the domain. It enables propagators to achieve a more fine-grained, hence stronger, inference. More propagation takes place in the constraint propagation, resulting in a dramatically small search tree. These phenomenon suggests that we should push even further in the domain representation, taking into account of more information and allowing propagators to exploit problem semantics, and leading to a more efficient and effective search.

So far in this thesis we have been focused our discussion solely on the length-lex representation, and we have shown that the representation enjoys, both theoretically and practically, a lot of advantages over the classical subset-bound domain. The subset-bound domain mainly captures the membership information by maintaining two sets that represent the state of an element: whether or not it belongs to the solution. The length-lex domain represents the set domain in a dual perspective, it features a total-ordering which primarily captures the cardinality information, making it an ideal vehicle to capture both cardinality and ordering constraints, and leaving the membership constraint in a relatively minor position. Figure 9.1 illustrates the characteristic of the two domain representations.

As we suggested in the last chapter that we may achieve a more efficient and effective search by allowing more propagation to take place. This chapter presents the idea of an hybrid domain: The intersection of length-lex and subset-bound domain, to obtain more propagation. We begin with the simple idea of maintaining two models at the same time and synchronizing them with a channeling constraint in Section 9.2. Then, we present exponential propagators for the product of two domains in Section 9.3. Last, we give a hardness proof for the global alldisjoint constraint for the hybrid domain in Section 9.4.

9.2 Connecting Two Representations

Constraint programming is highly modular. Variables and constraints can be added and removed from the model independently. Domains are the only interface of communication between constraints. Therefore, different implementations of the same constraint and different representations of the same variable type can be used in the same model. All we need is to make sure is that a correct interface exists.

In previous sections, we have presented models using the length-lex and subset-bound domains

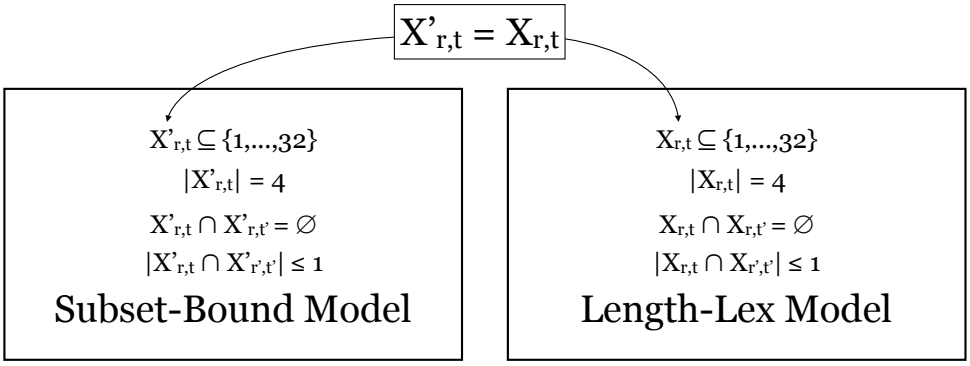

Figure 9.2: Connecting Two Representations using Channeling Constraints

respectively. To combine the two models, a trivial way is to put them together and connect corresponding variables with channeling constraints. Figure 9.2 illustrates the idea. The channeling constraint make sure both variables take the same value, it *channels* variables of different representations. Essentially, it is a equality constraint.

In the case of channeling the length-lex and subset-bound variables, the channeling constraint is composed of two parts. The first part propagates the length-lex variable and makes sure both bounds are domain values of the subset-bound variable. This can be trivially implemented as a unary constraint $sbc\langle r, p, \check{c}, \hat{c}\rangle(X) \equiv r \subseteq X \subseteq p \wedge \check{c} \leq |X| \leq \hat{c}$. The second part propagates the subset-bound variable in the same way. It deduces the required and possible set from the length-lex domain, and updates itself correspondingly.

Figure 9.3 presents the COMET model using both domain representations. The first box, Lines 15–26, are the length-lex constraints, the second box, Lines 27-31, are the subset-bound constraints, and the third box, Lines 32–33, are the channeling constraint connecting both models. Putting two models together cannot be more trivial.

9.2.1 Evaluation

We evaluate the performance of the three models: the subset-bound model, the length-lex model, and the hybrid model which uses both. To give a fair comparison, we do not add the exponential checker, which relies on the unary *atmost*1 constraint that is not yet introduced for the subset-bound domain. Table 9.1 presents the results. The hybrid model is clearly at least as strong as either the

```
LengthLexVar<CP> llx[Weeks,Groups](cp,p,s);
var<CP>{set{int}} sbx[Weeks,Groups](cp,Players,s..s);
var<CP>{int} y[Players,Weeks](cp,Groups);
var<CP>{set{int}} aux[Weeks](cp,Players,s..s);

solve<cp>{
  forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj)
    cp.post( disjointLe(llx[wi,gi],llx[wi,gj]) );

  forall (wi in Weeks, gi in Groups)
    cp.post( atmost1( llx[wi,gi], all(wj in Weeks, gj in Groups : wj < wi) llx[wj,gj] ) );

  forall (wi in Weeks, wj in Weeks : wi < wj)
    cp.post( atmostLe(llx[wi,1],llx[wj,1],1) );

  forall (wi in Weeks)
    cp.post( removeMin(llx[wi,1],aux[wi]) );
  cp.post( symbreak_alldisjoint(all(wi in 1..w) aux[wi],p) );

  forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj)
    cp.post( disjoint(sbx[wi,gi],sbx[wi,gj]) );

  forall (wi in Weeks, wj in Weeks, gi in Groups, gj in Groups : wi < wj)
    cp.post( atmost1(sbx[wi,gi],sbx[wj,gj]) );

  forall(wi in Weeks, gi in Groups)
    cp.post( channel(llx[wi,gi],sbx[wi,gi]) );

  forall (wi in DualWeeks)
    cp.post( dualChannel(all(gi in Groups)sbx[wi,gi], all(pi in Players)y[pi,wi]) );

  forall (pi in Players, pj in Players : pi < pj)
    cp.post( lexleq(all(wi in Weeks)y[pi,wi],all(wi in Weeks)y[pj,wi]) );
}
```

Figure 9.3: COMET Model for Social Golfer Problem using Both Domain Representations

subset-bound or the length-lex model. Therefore, it achieves the least number of fails node. The hybrid domain is also very robust, too. It solves all instances while the other two models cannot. Sometimes the hybrid domain runs a little bit slower than the pure length-lex model, since more propagations are taking place. But, overall it is faster. In terms of propagation strength, the impact of the hybrid domain is huge. For instance $(9, 3, 11)$, it reduces the number of fails by more than 7 times. It is also important to point out that, as the instance size grows larger, the performance gain becomes more apparent. We conclude that the hybrid domain is a robust model. Readers may refer to the experiment section for a more comprehensive comparison.

Domain	Subset-Bound		Length-Lex		Length-Lex × Subset-Bound	
Symmetry Breaking (Chapter 6)	✔		✔		✔	
Exponential Propagator ($bc_{ll}\langle atmost1\rangle$)			✔		✔	
(g,s,w)	Time	Fails	Time	Fails	Time	Fails
(3,3,3)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(4,3,5)	0.11	968	**0.03**	140	**0.03**	**126**
(4,4,6)	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
(5,3,6)	4.21	29791	**1**	6276	1.15	**4285**
(5,3,7)	25.63	157285	**4.56**	19149	11.25	**38454**
(5,4,5)	0.24	1594	**0.09**	448	0.1	**288**
(5,4,6)	337.35	2013980	**61.06**	286792	85.81	**225524**
(5,5,4)	0.33	2320	**0.05**	175	0.07	**151**
(6,3,7)	0.22	935	**0.05**	225	0.08	**107**
(6,4,5)	0.11	473	**0.03**	83	0.05	**57**
(6,5,5)	36.49	195986	**7.72**	36017	8.49	**19509**
(6,5,6)	314.24	1512264	**57.47**	221033	60.09	**127943**
(6,6,4)	x	x	**592.46**	2049826	952.55	**1964892**
(7,3,9)	x	x	x	x	x	x
(7,7,4)	x	x	**0.78**	1739	1.62	**1634**
(8,3,10)	1390.82	3782741	119.06	542539	**101.19**	**128833**
(9,3,11)	37.32	80005	14.449	61924	**9.66**	**8627**

Table 9.1: Social Golfer Problem: Hybrid Model.

9.3 Exponential Propagator for Hybrid Domains

Wandering in the No-Man Land of Hardness We have studied the performance gain of simply using two orthogonal domain representations together. This section studies the synergy of combining the two. Exponential propagators presented in the previous chapter accelerate the fixpoint algorithm by taking a collection of unary constraints into account at once. The problem is intractable. The question is either to let the agnostic constraint propagation algorithm to tackle the intractability or to design an informed exponential algorithm which exploits the problem semantics and yields a potentially faster propagation. The evaluation results clearly suggest the latter.

We embrace the complexity and wander in the no-man land of hardness. In this section, we propose the ls-domain, an intersection of the length-lex and subset-bound domain. And, we introduce exponential algorithms for the ls-domain which not only improves the convergence rate of the algorithm, but also prunes infeasible elements.

The Intersection We first define the ls-domain, the produce of the length-lex and subset-bound domain. (Similar hybrid domain representations were proposed in [56, 48].)

Definition 28 (ls-domain)**.** A length-lex × subset-bound domain (ls-domain) is the intersection of the two domains. A ls-domain $ls\langle l, u, n, r, p\rangle$ consists of two bounds l, u for the length-lex ordering, a universe size n, a required set r, and a possible set p. It represents the set of sets

$$ls\langle l, u, n, r, p\rangle \equiv ll\langle l, u, n\rangle \cap sbc\langle r, p, |l|, |u|\rangle$$

Example 44. The ls-domain $ls\langle\{1,3,8\},\{1,5,8\},8,\{1\},\{1,3,4,5,7,8\}\rangle$ denotes the set $\big\{\{1,3,8\},\{1,4,5\},\{1,4,7\},\{1,4,8\},\{1,5,7\},\{1,5,8\}\big\}$. ◇

Definition 29 (ls-bound consistency)**.** A set constraint $\mathcal{C}(X_1, ..., X_m)$ (X_i are set variables using the ls-domain) is said to be ls-bound consistent if and only if $\forall 1 \leq i \leq m$,

$$\begin{aligned}
& l_{X_i} \in d(X_i) \wedge \exists x_1 \in d(X_1), ..., x_m \in d(X_m) : C(x_1, .., x_{i-1}, l_{X_i}, x_{i+1}, .., x_m) \\
\wedge \quad & u_{X_i} \in d(X_i) \wedge \exists x_1 \in d(X_1), ..., x_m \in d(X_m) : C(x_1, .., x_{i-1}, u_{X_i}, x_{i+1}, .., x_m) \\
\wedge \quad & r_{X_i} = \bigcap_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} x_i \wedge p_{X_i} = \bigcup_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} x_i
\end{aligned}$$

where $d(X_i) = ls\langle l_{X_i}, u_{X_i}, n_{X_i}, r_{X_i}, p_{X_i}\rangle$

We now show that the ls-domain is strictly stronger than the conjunction of the ll-domain and sbc-domain.

Lemma 10. Enforcing bound consistency on a ls-domain is strictly stronger than enforcing bound consistency separately on the decomposition of the ll-domain and sbc-domain.

Proof. Clearly enforcing bound consistency on the ls-domain is at least as strong. Consider a unary constraint $|X \cap \{4,5,7\}| \leq 1$ and the ls-domain in Example 44. It is bound-consistent for the decomposition, since the lower and upper bounds satisfy the constraint and the required and possible sets are bound-consistent. However, for the ls-domain, only three domain values, i.e., $\{1,3,8\},\{1,4,8\},\{1,5,8\}$, satisfy the constraint. Element 8 belongs to all solutions and thus to the required set. Enforcing bound consistency for the ls-domain yields $ls\langle\{1,3,8\},\{1,5,8\},8,\{1,8\},\{1,3,4,5,7,8\}\rangle$. □

Algorithm 16 $bc_{ls}\langle atmost1(\{s_1,..,s_m\})\rangle(X_{ls} = ls\langle l,u,n,r,p\rangle)$

1: $\mathcal{S} \leftarrow \left\{s \in X_{ls} \mid \bigwedge_{1\leq i\leq m} |s \cap s_i| \leq 1\right\}$
2: $l', u' \leftarrow \min_{\preceq} \mathcal{S}, \max_{\preceq} \mathcal{S}$
3: $r', p' \leftarrow \bigcap_{s\in\mathcal{S}} s, \bigcup_{s\in\mathcal{S}} s$
4: **return** $ls\langle l', u', n, r', p'\rangle$

The Exponential Propagator In the previous chapter, we proposed an exponential propagator which improves the efficiency of the constraint propagation algorithm. We showed that it brings significant improvement in practice. We push the idea even further. Since the feasibility check, on which the exponential propagators rely, takes potentially exponential time, we would like to squeeze the most out of it. We introduce a simple exponential algorithms for enforcing bound consistency on the ls-domain. Our motivation is twofold:

1. An exponential filtering algorithm enables us to move the potentially exponential behavior from the rather agnostic constraint propagation algorithm into the constraint itself where the constraint semantics can be exploited.

2. The stronger filtering further increases the pruning of the search and may lead to additional domain reduction through constraint propagation of other constraints, an observation already pointed out in [7].

Algorithm 16 implements $bc_{ls}\langle atmost1(\{s_1,..,s_m\},X)\rangle$ and is self-explanatory. The set $\mathcal{S}$ maintains a logical enumeration of all possible solutions. All four bounds of the ls-domain are determined according to the ls-bound-consistency definition. Corollary 3 implies that there are no fixed-parameter tractable algorithm for Algorithm 16 since $hs_{ll}\langle\mathcal{C}\rangle$ is a special case for $bc_{ls}\langle\mathcal{C}\rangle$.

Theorem 19. Algorithm 16 runs in time $O(n^c mc)$ where $c = |u|$.

Proof. X_{ls} contains at most $O(n^c)$ sets. Each set takes $O(mc)$ time to verify if it satisfies the constraint. □

Sometimes enumerating all possible solutions is not cost-effective and hence we also consider an exponential filtering algorithm (Algorithm 14) for the length-lex bounds only. Obviously, lines 1–2 do not compute the set of solutions explicitly but only searches for the smallest and largest solution in the length-lex ordering. Algorithm 14 has the same worst case time complexity as Algorithm 16,

since its feasibility routine is $W[1]$-hard, but it may be significantly faster in practice. The same principles can be applied to other unary intersection constraints.

Implementation Notes The exponential propagation does not explicitly enumerate all the sets. It relies on a technique on which most length-lex propagators rely. The inference reduces to a feasibility routine hs which takes a domain and returns a boolean value that indicates if the domain has any solution. We discussed the length-lex component in the previous chapter. We focus on propagating the subset-bound component.

The subset-bound component has two parts: the possible set and the required set. A bound-consistent propagator decides whether to remove an element from the possible set, to add an element from the possible set to the required set, or to do nothing. An element is removed if no solution contains it; an element is added to the required set if all solutions contain it; otherwise, it remains in the possible set. These inferences rely on the feasibility routine. To test whether an element e should be removed from the possible set, it suffices to check if it belongs to no solution. Such task is achieved by adding e to the required set and invoking the feasibility routine. If the routine returns false, it implies no solution contains e, and it must be removed from the possible set to achieve bound consistency.

On the other hand, to test whether an element e belongs to all solutions, we perform a dual check and see if there is any solution where e is absent. We remove e from the possible set and invoke the feasibility routine. If the routine returns false, it implies all solutions contain e and hence it is a required element. We illustrate this with an example.

Example 45. Consider $bc_{sbc}\langle atmost1(\{s_1, s_2, s_3\})\rangle(X_{sbc})$, the exponential propagator which only propagates the subset-bound component, where $s_1 = \{2, 3\}$, $s_2 = \{2, 4\}$, $s_3 = \{2, 5\}$, and $X_{sbc} = sbc\langle\{\}, \{1, 2, 3, 4, 5\}, 3, 3\rangle$. The required set is empty and the possible set is $\{1, 2, 3, 4, 5\}$. The propagator scans all the possible elements twice: the first scan determines if they belongs to the possible set; the second scan determines the required set. In the first scan, the propagator determines if possible set elements have a support. Consider element 1: The propagator adds this element to the required set, get $X'_{sbc} = sbc\langle\{1\}, \{1, 2, 3, 4, 5\}, 3, 3\rangle$, and invokes the feasibility routine $hs_{sbc}(\langle atmost1(\{s_1, s_2\})\rangle)(X'_{sbc})$, which returns true as $\{1, 4, 5\}$ is a solution. Element 1 remains in the possible set. Now consider element 2: The propagator checks feasibility with the domain

$sbc\langle\{2\},\{1,2,3,4,5\},3,3\rangle$. There is no solution, the routine returns false, meaning that no solution contains element 2. Hence, element 2 is removed from the possible set. The process continues. At the end of the scan, all the other elements in the possible set belong to some solution and we have $sbc\langle\{\},\{1,3,4,5\},3,3\rangle$.

Indeed, in practice, it is not necessary to invoke hs for every element since results from previous calls can be reused. In particular, from the feasibility check of element 1, we know that element 3 and 5 also have support and hence don't need to invoke the computationally expensive check. Despite such trick doesn't improve the theoretical worst case complexity, it dramatically improves the overall performance of the exponential propagator.

The second scan determines if the elements belong to the required set. An element e belongs to the required set if and only if it belongs to all solutions. If there exists a solution where e is absent, e is not a required element. Consider element 1, the propagator removes it from the domain to obtain $sbc\langle\{\},\{3,4,5\},3,3\rangle$. The feasibility routine returns false indicating that all solutions contain element 1. Hence element 1 belongs to the required set. For element 3, the feasibility routine returns true since $\{1,4,5\}$ is a solution. Element 3 is not a required element. ◇

9.3.1 Evaluation

We evaluate the performance of the exponential propagator for the ls-domain. The propagator does two tasks: it speeds up the constraint propagation algorithm, and remove infeasible values. Line 18 is replaced by

```
1  atmost1(sbx[wi,gi], llx[wi,gi], all(wj in Weeks, gj in Groups: wj < wi)sbx[wj,gj] );
```

The first two arguments are the subset-bound and length-lex component of the concerned variable. The third argument corresponds to the array of sets s_i which defines the constraint (refer to Definition 21). Variables `sbx[wj,gj]` are not considered in the propagator until they become bound. Since the search uses a vanilla week-wise labeling strategy, `wj < wi` is sufficient. Binary constraints are removed from the model and, since the exponential constraint propagates only when variables are bound, less propagation is achieved in the length-lex component. Nevertheless, we show that the gain in performance outweighs the loss in propagation.

Domain	Length-Lex		Length-Lex × Subset-Bound		Length-Lex × Subset-Bound	
Symmetry Breaking (Chapter 6)	✔		✔		✔	
Checker ($hs\langle alldisjoint\rangle$)	✔		✔		✔	
Exponential Propagator ($bc_{ll}\langle atmost1\rangle$) ($bc_{ls}\langle atmost1\rangle$)	✔		✔		✔	
g,s,w	Time	Fails	Time	Fails	Time	Fails
(5,3,7)	**5.07**	12211	5.92	10181	6.05	**9291**
(5,4,6)	**39.7**	120438	56.46	108098	52.7	**79902**
(6,5,5)	**2.44**	4877	3.05	4193	3.01	**3405**
(6,5,6)	**16.38**	27545	21.49	24637	21.62	**19317**
(6,6,4)	**646.62**	1890962	930.65	1807408	1329.27	**1763316**
(7,3,9)	1276.05	2837356	1526.6	1778873	**1097.15**	**1018831**
(8,3,10)	47.01	88817	43.9	44249	**40.87**	**31547**
(9,3,11)	2.05	2724	2.89	1585	**1.13**	**330**

Table 9.2: Social Golfer Problem: Exponential Propagator for LS-domain

Table 9.2 reports the evaluation results. The first column gives the result of the pure length-lex model using all symmetry-breaking techniques discussed earlier and the exponential propagator $bc_{ll}\langle atmost1\rangle$ which *only* speeds up the convergence rate. The second column gives the result of the hybrid model: the length-lex and subset-bound models are posted separately and connected with channeling constraints. The third column gives the result of the hybrid model using the exponential propagator on the product of two domains, which improves the convergence rate *and* prunes infeasible values. Good and robust result worths a thousand of words. Most of the bold numbers are in the third column. For most instances, the ls-domain is the fastest approach, reducing both the search time and size of the search tree by orders of magnitude. The ls-domain is clearly the best.

Applying Exponential Checkers and Propagators on Subset-Bound Domain The exponential checker $hs\langle alldisjoint\rangle$ for the global alldisjoint constraint and exponential propagator $bc\langle alldisjoint\rangle$ for the unary atmost1 constraint can also be applied on the subset-bound domain. We also evaluate its performance. Table 9.3 evaluates the performance of each component. The model which uses checker and propagator gives the best result. It reduces the number of fails by orders of magnitude, and the run time is also much improved.

Domain	Subset-Bound					
Symmetry Breaking (Chapter 6)	✔		✔		✔	
Checker ($hs\langle alldisjoint\rangle$)					✔	
Exponential Propagator ($bc_{sbc}\langle atmost1\rangle$)			✔		✔	
g,s,w	Time	Fails	Time	Fails	Time	Fails
(5,3,7)	25.63	157285	19.97	63285	**19.13**	**40401**
(5,4,6)	337.35	2013980	195.72	583786	**160.73**	**346424**
(6,5,5)	36.49	195986	5.8	12001	**4.3**	**4529**
(6,5,6)	314.24	1512264	51.86	97991	**39.13**	**34773**
(6,6,4)	x	x	x	x	x	x
(7,3,9)	x	x	x	x	x	x
(8,3,10)	1390.82	3782741	404.97	492391	**273.07**	**204773**
(9,3,11)	37.32	80005	10.17	11601	**5.83**	**3557**

Table 9.3: Social Golfer Problem: Exponential Checkers and Propagators for Subset-Bound Domain

9.4 Hardness Proofs for AllDisjoint Global Constraint

The hybrid domain is a rich representation which encapsulate a lot of different information. It offers stronger propagation even for binary intersection constraints. It is also very effective in practice too. However, the rich domain makes propagation harder. One of the surprising example is that the alldisjoint constraint, which used to take polynomial-time for subset-bound domains, becomes intractable. The all-disjoint constraint ensures that every pair of set variables are mutually disjoint.

Definition 30 (All-Disjoint)**.** $alldisjoint(X_1, ..., X_m)$ holds if $X_i \cap X_j = \emptyset, \forall 1 \leq i < j \leq m$

This section gives two hardness proofs. The first shows that the all-disjoint constraint is intractable for the hybrid ls-domain. The second shows that it is intractable when the variables have an explicit list of possible domain values.

Hard for LS-Domain

Theorem 20. $hs_{ls}\langle alldisjoint\rangle(X_1, ..., X_m)$ is NP-hard.

Proof. Reduction from "Partition into Triangles". Instance: A graph $G = (V, E)$, with $|V| = 3q$ for some q. Question: Can the vertices of G be partitioned into q disjoint sets $V_1, V_2, ..., V_q$, each containing exactly 3 vertices, such that each of these V_i is the node set of a triangle in G.

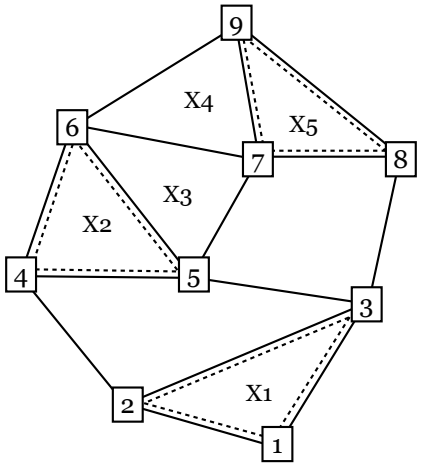

Figure 9.4: Reduction from 3-Triangles. Dotted lines represent the solution. Each triangle is represented by a set variable. A set variable takes only two values, the triangle or a set of two dummy elements.

Intuitively, every variable corresponds to a triangle in the input graph. Elements are drawn from a universe which consists of all vertices and some dummy elements. A variable takes 3 vertices from its triangle if and only it is one of the disjoint sets V_i, otherwise it takes the dummies. A partition is guaranteed by the alldiff relation since any vertex belongs to at most one set variable. There exists a partition if and only if the alldiff constraint has a solution.

Suppose there a $3q$ vertices and p triangles in the input graph. We construct an instance of CSP with p set variables whose element are drawn from a universe of $3q$ vertices and $2(p-q)$ dummy elements. One constraint over all set variables $alldiff(X_1, ..., X_p)$ is posted. We denote the universe as $\{v_1, ..., v_{3q}, d_1, ..., d_{2(p-q)}\}$ and abuse the length-lex notation and assume that $v_1 < v_2 < ... < v_{3q} < d_1 < ... < d_{2(p-q)}$. A set variable X_i $(1 \leq i \leq p)$ can either take 3 elements which corresponds to the vertices $(v_{i_1}, v_{i_2}, v_{i_3})$ in the triangle i, or 2 dummy elements. It has a ls-domain:

$$ls\langle\{d_1, d_2\}, \{v_{i_1}, v_{i_2}, v_{i_3}\}, 2p-q, \emptyset, \{v_{i_1}, v_{i_2}, v_{i_3}, d_1, ..., d_{2(p-q)}\}\rangle$$

This domain contains all 2-subsets of $\{d_1, ..., d_{2(p-q)}\}$ and a 3-set $\{v_{i_1}, v_{i_2}, v_{i_3}\}$.

Now we show the ls-domain contains only the aforementioned sets. It is trivial by the length-lex bound that it could take only sets of size 2 or 3. For any 2-set, it cannot contain any vertex v_j

since any 2-set $\{v_j, \bullet\}$ is smaller than $\{d_1, d_2\}$ in length-lex order. All domain values of size 2 are, therefore, subset of $\{d_1, ..., d_{2(p-q)}\}$. For any 3-set s, the possible set forbids s from taking any vertex v_j other than v_{i_1}, v_{i_2}, or v_{i_3}, and s cannot take any dummy element d_k since, say $\{v_{i_1}, v_{i_2}, d_k\}$, is greater than the length-lex upper bound $\{v_{i_1}, v_{i_2}, v_{i_3}\}$.

$\Rightarrow$ Given a partition of triangles, we construct a solution. For every triangle $\{v_{i_1}, v_{i_2}, v_{i_3}\}$ belongs to a partition, we assign it to the corresponding set variable X_i. Otherwise, when it is not, 2 dummy elements are assigned to X_i. There are p triangles and q partitions, $2(p-q)$ dummy elements suffices to satisfies the $alldiff$ constraint.

$\Leftarrow$ The argument is similar. For a set variable X_i taking $\{v_{i_1}, v_{i_2}, v_{i_3}\}$, these vertices form a partition. As there are q of them and they are mutually disjoint, they partition the graph. $\square$

9.5 Conclusion

Propagation is the core of constraint programming. CP models capture the problem semantics using variables and constraints, while solvers attempt to find solution by closely examining them. In order to enable the solver to solve a problem efficiently and perform inference effectively, variable domains should capture the inference result. In this thesis, we consider set domains, which is impossible to maintain domain values with an explicit representation. A few approximations schema have been proposed, each attempts to capture some pieces of important information. In particular, the subset-bound domain and the length-lex domain are two of the most prominent representations.

This chapter evaluates the intersection of the two domains. First, we give a lightweight and effortless method for combining model using the two domains respectively: everything is posted in the same model and the corresponding variables are connected using a channeling constraint. We show empirically that the combination gives a robust performance. Second, we introduce the ls-domain, the intersection of length-lex and subset-bound domain, and present an exponential propagator which exploits the synergy of the new domain. The hope is to shift the exponential behavior from the agnostic search to the propagator in which problem semantics can be exploited. The motivation of the exponential propagator is twofold: it improves the convergence rate of the constraint propagation algorithm, and it removes a lot of infeasible values. The empirical result is dramatic. It is clearly the fastest and most robust approach.

We also give theoretical result for the product of the two domains. The global alldisjoint constraint, which is tractable for the subset-bound domain, is intractable for the product. The newly proposed ls-domain may capture more domain information than before. As we demonstrated that exponential propagation may achieve dramatic improvement in performance, the intractability result suggests that it may also be beneficial to move some of the exponential behavior incurred by the alldisjoint constraint from the search to the filtering component.

Chapter 10

Exponential Checkers for Symmetry Breaking

10.1 Overview

Matrix models are a class of Constraint Satisfaction Problems that often exhibit significant symmetries and effective symmetry-breaking techniques are often critical in solving them in reasonable time. The LexLeader method is a common and elegant symmetry-breaking approach: It consists in posting a lex-ordering constraint for each symmetry to ensure that all non-canonical solutions are removed. Unfortunately, even for simple symmetry classes, the LexLeader method may generate an exponential number of constraints. A traditional way to overcome this limitation is to use only a subset of the symmetry-breaking constraints, which is the approach adopted in the DoubleLex and SnakeLex methods for matrix models. This chapter takes an orthogonal and complementary approach: instead of enumerating all the symmetry-breaking constraints for a symmetry class, it introduces the idea of a LexLeader feasibility checker that succeeds if a partial assignment can be extented into a canonical solution and fails otherwise. The implementation of the feasibility checker exploits a very interesting result from [42]: There exists an $O(n!nm \log m)$ algorithm to decide whether a solution is canonical in a $n \times m$ matrix model with row and column interchangeability. The paper shows how to use this algorithm for building LexLeader feasibility checkers. Moreover,

the chapter shows how LexLeader feasibility checkers can accommodate value symmetries and various variable orderings. The experimental results on 5 standard benchmarks show that LexLeader feasibility checkers may produce huge performance gains and are very robust overall.

This chapter is organized as follows. Section 10.2 describes the background and notations used in this paper. Section 10.3 introduces the novel idea of LexLeader Feasibility Checkers. Sections 10.4–10.6 present several extensions and improvements to the core idea. Section 10.7 concludes the chapter.

10.2 Background

A Constraint Satisfaction Problem (CSP) consists of a set of variables taking their values in a domain and a set of constraints. The problem is to find an assignment of values to variables satisfying all constraints. This paper focuses on matrix models [17] with n rows and m columns and variables are usually subscripted with row and column indices $X_{i,j}$. The domains are subsets of $\{1, ..., v\}$. A constraint specifies the allowed combinations of values for a subset of variables. An assignment is a function that maps all variables to values $\alpha(X_{i,j}) = a_{i,j}$ and a partial assignment is a partial function which maps a subset of variables to values. An assignment extends a partial assignment if they agree on the values of variables in the partial assignment.

A symmetry is a permutation of variables or values under which solutions are preserved. A permutation σ is denoted by, say, (23154), meaning $\sigma(1) = 2$, $\sigma(2) = 3$, and so on. This paper mostly focuses on the common symmetry types in matrix models $[X_{i,j}]$: A row symmetry σ_r is a row permutation $[X_{\sigma_r(i),j}]$, a column symmetry σ_c is a column permutation $[X_{i,\sigma_c(j)}]$, and a value symmetry σ_c is a value permutation $[\sigma_v(X_{i,j})]$. Of course, these various symmetries can be applied together, e.g., $[\sigma_v(X_{\sigma_r(i),\sigma_c(j)})]$, making symmetry breaking particularly challenging.

10.2.1 The LexLeader Method

The LexLeader method is a very common approach for breaking symmetries [11]: It eliminates symmetrically-equivalent solutions by keeping only a predefined canonical solution α. The canonical solution is usually the lexicographically smallest assignment for a predefined variable ordering.

Hence, to eliminate a variable symmetry σ, it suffices to post the following constraint:

$$[X_1, ..., X_n] \leq_{lex} [X_{\sigma(1)}, ..., X_{\sigma(n)}].$$

Example 46. Consider a CSP with two variables $X_1, X_2 \in \{0, 1\}$ and a constraint $X_1 \neq X_2$. There are two solutions: $\alpha_1(X_1) = 0, \alpha_1(X_2) = 1$ and $\alpha_2(X_1) = 1, \alpha_2(X_2) = 0$. There is a variable symmetry $\sigma = (21)$. Hence, the LexLeader method posts the lex-ordering constraint $[X_1, X_2] \leq_{lex} [X_2, X_1]$. The solution α_2 violates the ordering constraint and is therefore removed. $\diamond$

Similarly, to eliminate value symmetry σ, it suffices to post

$$[X_1, ..., X_n] \leq_{lex} [\sigma(X_1), ..., \sigma(X_n)].$$

10.2.2 The LexLeader Method in Matrix Models

Many matrix models exhibit both row and column interchangeability, a property called *full-interchangeability* [18]. To eliminate symmetries in a fully-interchangeable matrix model with n rows and m columns, the LexLeader method may define a row-wise variable ordering $row([X_{ij}]) \equiv [X_{1,1}, \ldots, X_{1,m}, X_{2,1}, \ldots, X_{2,m}, \ldots, X_{n,m}]$ and post the lex-ordering constraint

$$row([X_{i,j}]) \leq_{lex} row([X_{\sigma_r(i),\sigma_c(j)}])$$

for each row symmetry σ_r and column symmetry σ_c. There are respectively $n!$ and $m!$ different row and column permutations. Hence breaking all symmetries this way is forbiddingly expensive since there are $n!m!$ such lex-ordering constraints. In fact, breaking symmetries in fully-interchangeable matrix models is particularly challenging, since deciding whether a solution to such a model is canonical is already NP-complete [5].

10.2.3 The DoubleLex Method

The DoubleLex method is a popular method for breaking symmetries in fully-interchangeable matrix models [17].

Specification 9. The DoubleLex method takes a matrix model and enforces lex-ordering among

pairs of rows and columns.

$$\bigwedge_{1 \leq i < i' \leq n} [X_{i,1}, ..., X_{i,m}] \leq_{lex} [X_{i',1}, ..., X_{i',m}]$$
$$\wedge \bigwedge_{1 \leq j < j' \leq m} [X_{1,j}, ..., X_{n,j}] \leq_{lex} [X_{1,j'}, ..., X_{n,j'}].$$

The DoubleLex method does not break all symmetries.

Example 47. Consider a 2×3 fully-interchangeable matrix model and the following two assignments which satisfy the DoubleLex constraints:

$$\begin{array}{cc} 112 & 122 \\ 221 & 211 \end{array}$$

They are symmetrical under $\sigma_r = (21)$ and $\sigma_c = (321)$. ◇

In addition, complete filtering of DoubleLex constraints is computationally difficult.

Theorem 21 ([42])**.** Enforcing domain consistency on the DoubleLex constraints is NP-hard.

As a result, in practice, as well as in the evaluation section of this paper, the DoubleLex constraints are posted as a set of lex-ordering constraints among pairs of rows and columns independently.

10.2.4 The RowWiseLexLeader Method

Katsirelos, Narodytska, and Walsh [42] introduced an interesting method for checking if an assignment is a canonical solution to a fully-interchangeable matrix model. The RowWiseLexLeader method determines whether there exists a symmetrical solution which is smaller than a given assignment. If such solution exists, by transitivity, the current assignment cannot be canonical. Let S_k be the set of all permutations of $\{1, ..., k\}$.

Specification 10. The RowWiseLexLeader method takes an assignment α on a matrix model

and returns

$$\exists \sigma_r \in S_r, \sigma_c \in S_c :$$
$$row([\alpha(X_{\sigma_r(i),\sigma_c(j)})]) <_{lex} row([\alpha(X_{i,j})]).$$

The method is based on the observation that, if there is *only* one type of symmetry, the above test reduces to sorting. As a result, the RowWiseLexLeader method first enumerates all row symmetries and, for each of them, sorts the matrix and compares the resulting and original assignments.

Example 48. Consider a 2×3 fully-interchangeable matrix model and the assignment

122

211

Applying the row symmetry $\sigma_r = (21)$ produces

211

122

Now it remains to check if there exists a smaller assignment under column interchangeability. Sorting the columns produces the assignment

112

221

which is lexicographically smaller than the original assignment. The original assignment is not canonical. ◇

Theorem 22 ([42])**.** For a $n \times m$ fully-interchangeable matrix model, the RowWiseLexLeader method runs in $O(n!nm \log m)$ time.

[42] also showed that the DoubleLex method may leave $n!$ symmetries on some $2n \times 2n$ matrix models. Moreover, by applying Theorem 22 at the leaves of the search tree, they showed empirically

that DoubleLex may leave a large number of symmetries in some benchmarks.

10.3 LexLeader Feasibility Checkers

The key idea behind this paper is to turn Theorem 22 into a practical tool for removing symmetries during search. We first generalize Theorem 22 to partial assignments. If a partial assignment α of the first n' rows is such that another partial assignment of the same rows is lexicographically smaller than α, then any solution extending α is not canonical and the subtree corresponding to α may be pruned. Consider Example 48 and assume that the matrix has more than two rows. The partial assignment [(122),(211)] has exactly two rows filled. Since it is not canonical for the submatrix, any solution extending it is not canonical either and the algorithm can backtrack at this stage without trying to extend the assignment.

A LexLeader feasibility checker can directly use the implementation idea behind Theorem 22. Moreover, since the bottleneck in Theorem 22 is the $n!$ enumeration of the row symmetries, checking partial assignments early in the search, i.e., partial assignments with a small n', will be more efficient and may potentially prune large portions of the search space.

Specification 11 (RowCol Feasibility Checker)**.** RowColFC takes a partial assignment α of the first n' rows of a matrix model and returns

$$\exists \sigma_r \in S_{n'}, \sigma_c \in S_c :$$
$$row_{n'}([\alpha(X_{\sigma_r(i),\sigma_c(j)})]) <_{lex} row_{n'}([\alpha(X_{i,j})]).$$

where $row_{n'}$ only linearize the first n' rows. Note that σ_r only considers the interchangeability among the first n' rows.

Theorem 23. RowColFC takes $O(n'!n'm \log m)$ time.

Theorem 24. RowColFC removes all but the canonical solution in a n by m fully-interchangeable matrix model.

Proof. We first prove soundness: Only non-canonical solutions are removed. Then we prove completeness: All non-canonical solutions are removed.

Soundness: The lex-ordering relation is dominated by the prefix. Consider a partial assignment α of the first n' rows and a solution α_c extending it. If α has a symmetric partial assignment that is strictly smaller lexicographically than under any symmetries $\sigma_r \in S_{n'}$ and $\sigma_c \in S_c$, then α_c also has a symmetric assignment smaller than it. Formally,

$$\begin{aligned}
&\forall \sigma_r \in S_{n'}, \sigma_c \in S_c : \\
&\quad row_{n'}([\alpha(X_{\sigma_r(i),\sigma_c(j)})]) <_{lex} row_{n'}([\alpha(X_{i,j})]) \\
&\quad \Rightarrow row([\alpha_c(X_{\sigma_r(i),\sigma_c(j)})]) <_{lex} row([\alpha_c(X_{i,j})]).
\end{aligned}$$

RowColFC returns true when α cannot be extended into a canonical solution and no canonical solutions are removed.

Completeness: When the partial assignment is complete, RowColFC is equivalent to RowWiseLexLeader. □

For those cases in which the domain size v is much smaller than the number of rows and columns. In these cases, the running time of the algorithm can be improved with a bucket sort, reducing the complexity by a factor of $\log m$.

Theorem 25. RowColFC takes $O(n'!n' \max(m, v))$ time.

10.4 Variable Orderings

Canonical solutions depend on a pre-defined variable ordering. This section shows how to generalize LexLeader feasibility checkers to different variable orderings. In the literature, most models apply either a row-wise or column-wise canonical ordering. Recently, [30] introduced a very interesting variable ordering called SnakeLex. This section restricts attention to row-wise SnakeLex ordering since the column-wise counterpart is essentially equivalent. The SnakeLex ordering orders variables in a snake fashion. It takes variables from left to right in the first row, from right to left in the second, from left to right again in the third, and all the way until the last row. Empirical results in [30, 42] demonstrated that SnakeLex sometimes breaks more symmetries. A LexLeader feasibility checker can be naturally defined for the SnakeLex ordering.

Definition 31 (Snake Linearization)**.** $snake([X_{ij}]) \equiv [X_{1,1}, ..., X_{1,c}, X_{2,c}, X_{2,c-1}, ..., X_{2,1}, X_{3,1}, ...]$

Specification 12 (RowCol-Snake Feasibility Checker)**.** RowCol_SnakeFC takes a partial assignment α of the first n' rows of a matrix model and returns

$$\exists \sigma_r \in S_{n'}, \sigma_c \in S_m :$$
$$snake_{n'}([\alpha(X_{\sigma_r(i),\sigma_c(j)})]) <_{lex} snake_{n'}([\alpha(X_{i,j}]).$$

It is not difficult to see that RowCol_SnakeFC has the same time complexity as RowColFC. For instance, with the conventions used in this paper, it suffices to negate the even rows and to apply RowColFC.

10.5 Value Symmetries

This section generalizes LexLeader feasibility checkers to value symmetries, starting with value interchangeability.

Definition 32 (ValRowCol Feasibility Check)**.** ValRowColFC takes a partial assignment α of the first n' rows of a matrix model and returns

$$\exists \sigma_r \in S_{n'}, \sigma_c \in S_c, \sigma_v \in S_v :$$
$$row([\alpha(\sigma_v(X_{\sigma_r(i),\sigma_c(j)}))]) <_{lex} row([\alpha(X_{i,j})]).$$

The implementation for ValRowColFC adds an extra layer of value permutation to the top of RowColFC.

Theorem 26. ValRowColFC takes $O(v!n'!n' \max(m, v))$ time.

Example 49. Consider the partial assignment in a matrix model with row, column, and value interchangeability,

112233

121233

To check whether it can be extended into the canonical solution, we enumerate all value permutations and apply RowColFC. For instance, $\sigma_v = (231)$ produces the submatrix

223311

232311

RowColFC returns true and the partial assignment cannot be extended into a canonical solution. ◇

This approach is not limited to value interchangeability only; the same principle can be applied to any kind of value symmetries. The key is simply to enumerate all but one type of symmetry and to exploit the semantics of the remaining one. We illustrate the approach by presenting a feasibility checker for a specific value symmetry class.

Definition 33 (Error Correcting Code, Lee Distance (ECCLD))**.** The problem is to find d codewords of length-q that drawn from 4 symbols $(1, 2, 3, 4)$ such that the Lee Distance between every-pair of codeword is exactly c. The Lee Distance between two symbols a, b is $\min(|a - b|, 4 - |a - b|)$.

The ECCLD problem can be modelled as a matrix model. It has row and column symmetries and an interesting class of value symmetries. Indeed, the values are not interchangeable but the symmetry class Σ_{lee} contains 8 symmetries:

$$\{(1234), (1432), (2143), (2341), (3214), (3412), (4123), (4321)\}$$

The value symmetries apply to each column independently, since the only constraint is the Lee distance between corresponding columns in each row.

Example 50. The two ECCLD solutions

1122 1111

2434 2223

are symmetric. The first column is obtained by identity, the second by (1432), the third by (4123), and the last by (2143). ◇

A LexLeader feasibility checker for the ECCLD problem can be obtained by enumerating all row and column symmetries and leaving the value symmetry to the sorting step. This is more efficient than enumerating all value symmetries since there are 8^m of them.

Specification 13 (RowColLee Feasibility Checker)**.** RowColLeeFC takes a partial assignment α of the first n' rows of a matrix model and returns

$$\exists \sigma_r \in S_{n'}, \sigma_c \in S_c, \sigma_v \in \Sigma_{lee} :$$
$$row_{n'}([\alpha(\sigma_v(X_{\sigma_r(i),\sigma_c(j)}))]) <_{lex} row_{n'}([\alpha(X_{i,j})]).$$

Theorem 27. RowColLeeFC runs in $O(n!m!nm)$ time.

Proof. First, we enumerate all possible row and column symmetries: There are $n!m!$ of them. The resulting matrices only contain value symmetries and the task is to determine if there exists a value symmetry for each column that would produce a new assignment lexicographically smaller than α. Let $Y_{ij} = X_{\sigma_r(i),\sigma_c(j)}$ for $1 \leq i \leq n, 1 \leq j \leq m$ be such a matrix. For each column j, we introduce a variable $Z_j \in \{0, ..., 7\}$ to denote its possible value symmetries in Σ_{lee}. We perform a row-wise scan of the matrix and, for every Y_{ij}, we check if there exists a value symmetry in Z_j yielding a smaller value than X_{ij}. If such a symmetry is found, α is not canonical solution. Otherwise, we remove all value symmetries in Z_j that would yield a larger value than X_{ij} (the remaining values in Z_j preserves the values of $Y_{1j}, \ldots, Y_{ij}$). The whole matrix needs to be scanned only once and the check in each cell takes $O(1)$ time (we only need to index each symmetry pattern with the value of Y_{ij}). The total runtime is therefore $O(n!m!nm)$. □

10.6 Practical Considerations

The earlier section illustrated a key aspect of the approach: one can choose which symmetries to enumerate to obtain the best performance. For instance, on some problems, it may be more appropriate to enumerate the value symmetries first, while it may be unpractical to do so in others.

In practice, it may not be cost-effective to break all symmetries systematically. For instance, one can restrict the feasibility checker to the first k rows of the model, reducing the running time to $O(k!k \max(m, v))$. On some problems, this may significantly improve the performance of the

approach. Once again, it is useful to note that the earlier LexLeader feasibility checks cost less and may prune large portions of the search tree. This is a nice property of the approach.

Finally, it is always possible to use several feasibility checkers simultaneously capturing different combinations of symmetries, provided that they use the same variable ordering. For instance, in a fully-interchangeable matrix model with value symmetries, one may use one checker for row and value symmetries, another for column and value symmetries, and a third for row and column symmetries.

Please refer to Chapter 11.6 for the evaluation of feasibility checkers on various standard benchmarks.

10.7 Conclusion

This chapter proposed the idea of LexLeader feasibility checkers that verify, during search, whether the current partial assignment can be extended into a canonical solution. The feasibility checkers are based on a result by [42] on how to check efficiently whether a solution is canonical. This paper showed how to generalize this result to partial assignments, various variable orderings, and value symmetries. Several checkers combining different types of symmetries can be used simultaneously, instead of tackling all symmetries together which may be prohibitive. Empirical results on 5 standard benchmarks showed that feasibility checkers may bring significant, sometimes spectacular, performance gains.

Chapter 11

Experimental Results

11.1 Overview

In this thesis, we proposed a series of efficient and effective propagators and modeling techniques for the length-lex set variables. We have shown that, using just a few instances in the social golfer problem, models using the length-lex set variables yield the smallest number of fail nodes as well as the shortest solving time comparing with models using the classical subset-bound domain.

This chapter provides a comprehensive comparison between our proposal, the length-lex propagators, and earlier attempts over standard benchmarks used in the constraint-programming community. All of them can be found in the CSPlib. The objective is to support the central thesis:

Length-Lex is an Effective Set Domain Representation for Constraint Programming.

We show that length-lex is a robust, efficient, and effective domain representation. We demonstrate that it is a accurate approximation of the set domain which allows effective propagation. We claim that, despite of its accuracy, it is possible to achieve efficient filtering algorithms. We advocate that, in many problems, as long as strong propagation is achieved, the labeling heuristics have a marginal impact in performance. To support such claim, in each benchmark problem, we use one vanilla static labeling routine for *all instances*, and the overall performance is orders of magnitude faster

than most previous techniques.

In particular, we focus on four benchmark problems which can be naturally modeled using set variables. They are the social golfer problem, the steiner triple system, the weighted error correcting code problem, and the balanced incomplete block design problem. In each problem, we present a comprehensive comparison between length-lex, as well as its variants, and earlier attempts. Since there are too many of them, only the best results for each approach are shown. We solve every instance solvable by other techniques. Empty cells means non-reported instances, while the symbol x illustrates a timeout.

From a scientific standpoint, we are also interested in understanding the impact of each component in the model and provide statistics for a couple of variants based on the length-lex model.

11.2 The Social Golfer Problem

11.2.1 Problem Statement

The task is to find a w-weekends schedule for $p = g \times s$ golfers, each of whom plays golf once a weekend, and always play in a group of s golfers. To make sure each golfer has the maximum opportunity of playing with others, every pair of golfers can play at most once. [31]

11.2.2 Earlier Work

This benchmark is derived from a post in sci.op-research in May 1998. It gained a lot of attention in the constraint programming community thanks to its elegant CP model and complex symmetry class. Various methods have been proposed to solve the problem. Dotú, Fernández and Van Hentenryck[10, 14], and Harvey and Winterer[32] proposed local search algorithms for finding solutions. Focacci and Milano[19], Smith[61], and Fahle, Schamberger and Sellmann[16] remove symmetry dynamically during search.

We focus on those which applies static model. Perhaps the most commonly used model is given by Barnier and Brisset[3]. They proposed an integer model and a set model. Various proposals on set representations as well as global constraints are based on these two models. The Cardinal system, proposed by Azevedo, introduces a cardinality component to the classical subset-bound domain to

enhance propagation[2]. A multiple viewpoint model, a similar variant proposed by Law and Lee, removes interchangeability among golfers in a dual viewpoint[44]. Law and Lee also introduces a global constraint for breaking a pair of interchangeable values[45]. Frisch, Hnich, Kiziltan, Miguel, and Walsh introduced a lexicographical constraint for breaking symmetry in matrix models[21, 43]. Hawkins, Lagoon, and Stuckey proposed an exact domain representation using reduced-ordered binary-decision-diagram[34]. Gange, Lagoon, and Stuckey used the binary-decision-diagram library as a black box for clause generation, applied learning algorithm and randomized search strategy to find solutions[23]. Van Hoeve and Sabharwal presented a bound-consistent propagator for binary atmost1 constraint[70].

11.2.3 Model

We used the social golfer model as an running example to demonstrate the impact of different proposed techniques throughout the thesis. Readers may refer to section A.1 for a detailed description. We use the model proposed by Barnier and Brisset, figure A.3. Figure 11.2 presents the model in the COMET language. The model uses both length-lex and subset-bound domain. All techniques introduces in this thesis are applied:

Technique	Chapter	Lines
Efficient Length-Lex Propagators	4, 5, B.2	16,17,22,23,35,36
Pushing Symmetry-Breaking into Binary Propagators	6.2	16,17,22,23
Global Symmetry-Breaking AllDisjoint Propagator	6.3	25–26
Dual Modeling for Breaking Value Symmetry	6.4	35–42
Hybrid Domain Representation	9.2	29–36
Exponential Unary Propagators	7, 9.3	19,20
Exponential Feasibility Checker for AllDisjoint Constraint	8.2	13,54–56

We evaluate the pure length-lex domain and the ls-domain, the product of length-lex and subset-bound domain. The model for pure length-lex domain is omitted to save a few trees in the amazon forest, it is obtained by removing the `disjoint` and `atmost1` binary constraints and replacing the exponential `atmost1` constraint by `atmost1_ll` which only propagates the length-lex bounds.

```
int g=3;
int s=3;
int w=3;
int p = g*s;
range Groups = 1..g;
range Weeks = 1..w;
range Players = 1..p;
LengthLexVar<CP> llx[Weeks,Groups](cp,p,s);
var<CP>{set{int}} sbx[Weeks,Groups](cp,Players,s..s);
var<CP>{int} y[Players,Weeks](cp,Groups);
var<CP>{set{int}} aux[Weeks](cp,Players,s..s);

AllDisjointChecker alldisjoint[wi in Weeks](cp,llx,sbx,wi);

solve<cp>{
  forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj)
    cp.post( disjointLe(llx[wi,gi],llx[wi,gj]) );

  forall (wi in Weeks, gi in Groups)
    cp.post( atmost1( llx[wi,gi], all(wj in Weeks, gj in Groups : wj < wi) llx[wj,gj] ) );

  forall (wi in Weeks, wj in Weeks : wi < wj)
    cp.post( atmostLe(llx[wi,1],llx[wj,1],1) );

  forall (wi in Weeks)
    cp.post( removeMin(llx[wi,1],aux[wi]) );
  cp.post( symbreak_alldisjoint(all(wi in 1..w) aux[wi],p) );

  forall (wi in Weeks, gi in Groups, gj in Groups : gi < gj)
    cp.post( disjoint(sbx[wi,gi],sbx[wi,gj]) );

  forall (wi in Weeks, wj in Weeks, gi in Groups, gj in Groups : wi < wj)
    cp.post( atmost1(sbx[wi,gi],sbx[wj,gj]) );

  forall(wi in Weeks, gi in Groups)
    cp.post( channel(llx[wi,gi],sbx[wi,gi]) );

  forall (wi in DualWeeks)
    cp.post( dualChannel(all(gi in Groups)sbx[wi,gi], all(pi in Players)y[pi,wi]) );

  forall (pi in Players, pj in Players : pi < pj)
    cp.post( lexleq(all(wi in Weeks)y[pi,wi],all(wi in Weeks)y[pj,wi]) );

}
```

Figure 11.1: COMET Model for Social Golfer Problem

11.2.4 Discussion

Figure 11.1 illustrate the result. Two length-lex models solve all instances and are the fastest. None of the previous works is comparable with us except those by Stuckey and friends (Hawkins, Lagoon, and Gange). In the ROBDD model, the split variant, which separates fixed and unfixed elements in two diagrams, is used. The split domain enforces extremely strong propagation, in many

```
45 using{
46   forall (pi in 1..g*s) cp.post(requiresValue(sbx[1,(pi-1)/s+1],pi));
47   forall (si in 1..s) cp.post(requiresValue(sbx[2,1],(si-1)*s+1));
48   forall (wi in 2..w)
49     while (or(gi in 1..g)(!sbx[wi,gi].bound()))
50       selectMin(gi in 1..g) ( sbx[wi,gi].getRequiredSet().getSize(), gi ){
51         selectMin(pi in 1..g*s: !sbx[wi,gi].isExcluded(pi) && !sbx[wi,gi].isRequired(pi) )(pi)
52           try<cp> cp.post(requiresValue(sbx[wi,gi],pi));
53           | cp.post(excludesValue(sbx[wi,gi],pi));
54         forall (wj in 4..w)
55           if (!alldisjoint[wj].hs())
56             cp.fail();
57     }
58 }
```

Figure 11.2: COMET Search for Social Golfer Problem

instances, it achieves almost no failure. However, it is not as robust. The diagram size may grow exponentially large, resulting a huge computational cost. For instance, in (9,4,4), it spends 107 seconds in computation and has no fail nodes. On the other hand, BDD-SAT based on a subset-bound domain and apply clause learning techniques with widely used in the SAT community. The performance is good, the average runtime is only 2.7 seconds and it solves all instances.

Length-lex and the ls-domain is clearly the best. Both models, on average, solve an instance in around 0.3 seconds. The number of fails is dramatically smaller than those using classical subset-bound domain too. This supports our thesis, that length-lex is an effective and efficient domain representation for set variables.

11.2.5 A Close Look

We show that the length-lex domain and the ls-domain are clear the state-of-the-art technique. Instances studied in the previous section are too small and trivial to tell the difference of contribution between different components. Since the start of the thesis, we have been amending many interesting model techniques, for example, pushing symmetry-breaking constraints into binary propagators, dual modeling, which make solving larger instances possible. We conclude that strong propagation is the key of solving large instances. On this premise, we introduce exponential propagators, which stresses more on propagation, and are able to proceed even more. Afterwards, we introduced an exponential feasibility checker for the global alldisjoint constraint, which takes explicit domain representations. We include all of them in the model for evaluation.

The goal of this thesis is to show that length-lex is an effective domain representation for set variables. We have already shown in previous sections that length-lex is dramatically better than the pure subset-bound model. Therefore, here we compare the pure length-lex model and the hybrid ls-domain model. Table 11.2 presents the results. Length-lex is faster than ls-domain in many instances, especially for the small instance since the additional propagation done by the subset-bound variables seems redundant. However, length-lex is not as robust as the hybrid ls-domain. For example, instance $(9, 5, 7)$, ls-domain is more than 5 times faster than length-lex, and visiting substantially less nodes. The difference between the two approaches becomes more apparent in large instances, where adding the subset-bound component facilitates propagation.

The Social Golfer Problem

Method	Set-Int		Cardinal	Pair-Atmost-1		ROBDD		ROBDD		BDD-SAT	Length-Lex		LS-Domain	
Heuristic	Min-Dom		Min-Dom	Min-Dom		Static		Min-Dom		VSIDS	Weekwise		Weekwise	
CPU	Sun Blade 1000		P4 2.4GHz	Xeon 3.8GHz		P4 2.8GHz		P4 2.8GHz		C2D 3GHz	C2D 2.4GHz		C2D 2.4GHz	
Timeout	900s		900s			600s		600s			1800s		1800s	
(g,s,w)	Time	Fails	Time	Time	Fails	Time	Fails	Time	Fails	Time	Time	Fails	Time	Fails
(4,2,4)	0.19	266								**0.01**	**0.01**	**1**	**0.01**	**1**
(4,2,5)	0.52	884								0.02	**0.01**	**1**	**0.01**	**1**
(4,2,6)	0.51	721								0.02	**0.01**	**1**	**0.01**	**1**
(4,2,7)	0.15	97								0.02	**0.01**	**2**	**0.01**	**1**
(4,3,4)	0.81	3222								**0.01**	**0.01**	5	**0.01**	**2**
(4,3,5)			165.63			32.1	5165	26	3812	0.46	**0.03**	140	0.04	**112**
(4,3,6)			94.67			23	2132	15.2	1504	0.03	**0.01**	**0**	**0.01**	**0**
(4,4,4)	8.11	15759								0.02	**0.01**	**0**	**0.01**	**0**
(4,4,5)	5.63	7510								0.02	**0.01**	**0**	**0.01**	**0**
(5,2,3)	1.07	521								0.02	**0.01**	**0**	**0.01**	**0**
(5,2,4)	78.46	95063								0.01	**0.01**	**0**	**0.01**	**0**
(5,2,5)	1956.71	2554588								0.02	**0.01**	**0**	**0.01**	**0**
(5,2,6)										0.01	**0.01**	**0**	0.02	**0**
(5,2,7)										0.04	**0.01**	2	0.02	**1**
(5,2,8)										0.06	**0.01**	**1**	0.03	**1**
(5,2,9)	6841.59	3895064								0.12	**0.02**	1	0.03	**0**
(5,3,3)	140.06	491452								0.02	**0.01**	**1**	**0.01**	**1**
(5,3,4)										0.03	**0.01**	**1**	**0.01**	**1**
(5,3,5)										0.03	**0.01**	**1**	0.02	2
(5,3,6)			x			1.5	82	1	34	0.8	**1.05**	3174	1.1	**2123**
(5,3,7)			x					13.1	528	6.32	**5.1**	12211	6.04	**9291**

Method	Set-Int		Cardinal	Pair-Atmost-1		ROBDD		ROBDD		BDD-SAT	Length-Lex		LS-Domain	
(g,s,w)	Time	Fails	Time	Time	Fails	Time	Fails	Time	Fails	Time	Time	Fails	Time	Fails
(5,4,2)			0.83			0.1	**0**	0.1	**0**	0.02	**0.01**	**3**	**0.01**	3
(5,4,3)	98.23	658755	1.89			0.3	**0**	0.3	**0**	0.03	**0.01**	**3**	**0.01**	3
(5,4,4)	4770.94	30802587	3.13			0.6	**0**	0.6	**0**	0.07	**0.01**	5	0.02	3
(5,4,5)			28.65			2.3	41	1.7	18	0.98	**0.07**	180	0.08	**106**
(5,5,3)	0.46	1418								0.02	**0.01**	2	**0.01**	**0**
(5,5,4)	2.1	13009								0.05	**0.05**	175	0.09	**151**
(5,5,5)										0.07	**0.06**	175	0.13	**147**
(5,5,6)										0.15	**0.07**	171	0.11	**139**
(5,5,7)			x			0.4	**0**	0.4	**0**	3.28	**0.01**	**0**	**0.01**	**0**
(6,3,2)	1.16	30								0.02	**0.01**	**3**	**0.01**	**3**
(6,3,3)										0.03	**0.01**	5	**0.01**	**4**
(6,3,4)										0.04	**0.01**	5	0.02	**4**
(6,3,5)										0.04	**0.02**	5	0.03	**4**
(6,3,6)			1.2			1.4	**0**	1.3	7	0.39	**0.03**	7	0.04	**4**
(6,4,2)	0.99	45	1.75			0.1	**0**	0.1	**0**	0.04	**0.01**	6	**0.01**	6
(6,4,3)			4.62			1.4	**0**	0.9	**0**	0.06	**0.01**	6	0.02	6
(6,5,2)	0.1	60		17.2	171664					0.13	**0.01**	**8**	**0.01**	**8**
(6,5,3)				29.6	197607					0.15	**0.02**	**8**	**0.02**	**8**
(6,5,4)			x	39.7	197837	80.7	**0**	40.1	**0**	0.45	**0.02**	11	0.04	8
(6,5,5)				75.5	239966					15.24	**2.44**	4877	2.79	**3405**
(6,6,3)	414.16	1521747								0.04	**0.02**	4	0.03	**2**

Method	Set-Int		Cardinal	Pair-Atmost-1		ROBDD		ROBDD		BDD-SAT	Length-Lex		LS-Domain	
(g,s,w)	Time	Fails	Time	Time	Fails	Time	Fails	Time	Fails	Time	Time	Fails	Time	Fails
(7,2,2)	0.69	21								0.03	**0.01**	**0**	**0.01**	**0**
(7,3,2)	58.94	42								0.02	**0.01**	**0**	**0.01**	**0**
(7,4,2)	236.73	63	2.82			0.4	**0**	0.4	**0**	0.02	**0.01**	2	**0.01**	2
(7,4,3)			6.37			8.4	**0**	1.7	**0**	0.07	**0.02**	2	0.03	2
(7,4,4)			12.46	4.4	27877	481.6		5.1	**0**	0.08	**0.03**	2	0.05	2
(7,4,5)			17.18			x	x	12.8	**0**	0.22	**0.05**	4	0.07	2
(7,5,2)	42.98	84								0.45	**0.02**	**9**	**0.02**	**9**
(7,6,2)	0.53	105								1.44	**0.02**	**18**	0.04	**18**
(7,7,2)	0.7	308								0.06	**0.03**	**0**	0.04	**0**
(8,3,5)			1.01			7.8	**0**	3.9	**0**	0.09	**0.04**	3	0.06	3
(8,4,4)				157.7	738393					0.12	**0.04**	8	0.07	**7**
(8,5,2)			x			1.6	**0**	1.6	**0**	0.56	**0.02**	13	0.04	13
(9,4,4)			42.45			x	x	107.4	**0**	0.19	**0.05**	1	0.08	1
(10,3,6)				17.3	57364					0.91	**0.09**	2	0.17	**0**
(10,3,9)				52.4	78613					16.66	**0.28**	15	0.43	**3**
(10,3,10)				67.2	78976					110.8	**0.36**	15	0.54	**13**
(10,4,4)				4	22043					0.52	**0.06**	**5**	0.12	**5**
(10,4,5)				4.5	22044					0.76	**0.1**	**5**	0.19	**5**
Count	26		15	11		17		20		**60**	**60**		**60**	
Average	563.94	1.5×10^6	25.64	42.68	1.6×10^5	37.86	463.75	11.69	295.15	2.71	**0.17**	355.08	0.21	**260.62**

Table 11.1: Social Golfer Problem: Comparing Length-Lex with Earlier Attempts.

Domain	Length-Lex		Length-Lex × Subset-Bound	
Sym-Break (Chapter 6)	✔		✔	
Exponential				
($bc_{ll}\langle atmost1\rangle$)	✔			
($bc_{ls}\langle atmost1\rangle$)			✔	
($hs\langle alldisjoint\rangle$)	✔		✔	
(g,s,w)	Time	Fails	Time	Fails
(5,3,7)	**5.07**	12211	6.05	**9291**
(5,4,6)	**39.7**	120438	52.28	**79902**
(6,5,5)	**2.44**	4877	3405	**3405**
(6,5,6)	**16.38**	27545	21.62	**19317**
(6,6,4)	**646.62**	1890962	1329.27	**1763316**
(7,3,9)	1276.05	2837356	**1097.15**	**1018831**
(7,4,6)	**0.07**	**5**	0.09	**5**
(7,5,5)	**0.07**	17	0.09	**11**
(7,6,4)	**0.06**	27	0.09	**23**
(7,7,4)	**0.64**	875	1.15	**612**
(8,3,10)	47.01	88817	**40.87**	**31547**
(8,4,7)	**0.24**	143	0.31	**74**
(8,5,6)	**0.24**	114	0.24	**29**
(8,6,5)	**0.21**	**67**	0.27	**56**
(8,7,4)	**0.14**	45	0.2	**43**
(9,3,11)	2.05	2724	**1.13**	**330**
(9,4,8)	**0.89**	427	2.26	**707**
(9,5,7)	13.63	6237	**2.67**	**172**
(9,6,6)	**2.48**	536	10.07	**93**
(9,7,5)	34.1	102	**4.35**	**77**
(9,8,4)	**4.19**	79	4.35	**77**
(10,3,11)	**0.46**	25	0.65	**17**
(10,4,9)	1.28	417	**1.02**	**132**
(10,5,7)	**0.44**	37	0.58	**23**
(10,6,6)	**0.41**	34	0.69	**25**
(10,7,5)	**0.68**	82	0.82	**46**
(10,8,4)	**0.48**	408	0.61	**401**

(Continue next page)

Domain	Length-Lex		Length-Lex × Subset-Bound	
Sym-Break (Chapter 6)	✔		✔	
Exponential				
$(bc_{ll}\langle atmost1\rangle)$	✔			
$(bc_{ls}\langle atmost1\rangle)$			✔	
$hs\langle alldisjoint\rangle$	✔		✔	
(g,s,w)	Time	Fails	Time	Fails
(11,3,13)	**0.88**	200	1.28	**14**
(11,4,10)	1.66	477	**1.31**	**50**
(11,5,8)	**1.08**	**26**	1.81	**26**
(11,6,7)	**3.74**	89	5.73	**76**
(11,7,5)	**0.47**	95	0.79	**92**
(11,8,5)	**61.04**	3291	61.37	**3282**
(11,9,3)	2.56	809	**0.91**	**809**
(11,10,3)	4.25	148	**0.61**	**148**
(12,3,14)	**1.6**	119	2.13	**40**
(12,4,11)	53.15	15264	**18.65**	**3384**
(12,5,9)	7.23	586	**5.66**	**263**
(12,6,7)	**1.73**	431	2.56	**41**
(12,7,6)	1.95	460	**1.83**	**441**
(12,8,4)	**0.93**	1477	1.54	**1474**
(12,9,4)	36.66	8967	**7.71**	**8984**
(12,10,4)	74.3	9085	**8.93**	**3693**
(13,3,16)	**2.4**	92	3.39	**47**
(13,4,12)	**2.99**	117	3.12	**19**
(13,5,10)	**11.09**	317	13.89	**489**
(13,6,8)	6.94	138	**3.52**	**153**
(13,7,7)	**22.01**	3155	28.51	**3134**
(13,8,6)	174.49	40404	**71.68**	**6670**
(13,9,5)	10.43	677	**4.14**	**530**

Table 11.2: Social Golfer Problem: Length-Lex Domain vs Hybrid Length-Lex × Subset-Bound Domain.

11.3 The Steiner Triple System

11.3.1 Problem Statement

The problem is also called Steiner Triple System, a common benchmark problem. Given v elements in the universe. The goal is to find $b = v(v-1)/6$ blocks such that each block consists 3 elements and every pair of blocks share exactly 1 element in common.

11.3.2 Earlier Work

The steiner triple system is a popular benchmark because of its remarkably simplicity. We name a few. Frisch, Hnich, Kiziltan, Miguel, and Walsh introduced a lexicographical constraint, as well as a combination of the symmetry-breaking constraint and a sum constraint, for breaking symmetry in matrix models[21, 43]. Sadler and Gervet introduced hybrid domain representation which takes cardinality and lexicographical information into account during propagation [56, 57]. The Cardinal system, proposed by Azevedo, introduces a cardinality component to the classical subset-bound domain to enhance propagation[2]. Law and Lee proposed a multiple viewpoint model for breaking symmetries[44]. Law and Lee also introduces a global constraint for breaking a pair of interchangeable values[45]. Hawkins, Lagoon, and Stuckey proposed an exact domain representation using reduced-ordered binary-decision-diagram[34]. Gange, Lagoon, and Stuckey used the binary-decision-diagram library as a black box for clause generation, applied learning algorithm and randomized search strategy to find solutions[23]. Van Hoeve and Sabharwal presented a bound-consistent propagator for binary atmost1 constraint[70].

11.3.3 Model

We use two models for comparison with earlier works, the length-lex domain model and the ls-domain model.

The LS-Domain Model Figure 11.3 illustrates the latter in the COMET language. The value `v` in line 1 is the only parameter of this problem. Lines 2–10 are the initialization steps. The primal variables `llx[xi]`, `sbx[xi]` denotes the set of blocks in which element `xi` occurs, the dual

```
1 int v = 7;
2 int k = 3;
3 int r = (v-1)/(k-1);
4 int b = v*r/k;
5 range V = 1..v;
6 range B = 1..b;
7 Solver<CP> cp();
8 LengthLexVar<CP> llx[V](cp,b,r);
9 var<CP>{set{int}} sbx[V](cp,B,r..r);
10 var<CP>{set{int}} sby[B](cp,V,k..k);
11
12 solve<cp> {
13   forall(xi in V : xi > 1){
14     cp.post( exact1_ll(sbx[xi],llx[xi],all(xj in V : xj < xi)sbx[xj]) );
15     cp.post( llx[xi-1] <= llx[xi] );
16   }
17
18   forall(xi in V, xj in V : xi < xj)
19     cp.post(exact1(sbx[xi],sbx[xj]) );
20
21   forall(xi in V)
22     cp.post(channel(llx[xi],sbx[xi]));
23
24   cp.post(channeling(sbx,sby));
25
26   forall(yi in B, yj in B : yi < yj)
27     cp.post( lexleq( all(ye in V) sby[yj].getRequired(ye),
28                      all(ye in V) sby[yi].getRequired(ye) ));
29 }using {
30   forall (xi in V)
31     forall (xe in B : !sbx[xi].isRequired(xe) && !sbx[xi].isExcluded(xe))
32       try<cp> cp.requires(sbx[xi],xe);
33       | cp.excludes(sbx[xi],xe);
34 }
```

Figure 11.3: COMET Model for Steiner Triple System

variables `sby[yi]` denotes the set of elements taken by block `yi`.

Line 14 is the exponential length-lex propagator for the `exact1` constraint, notice that the subset-bound component is *not* propagated as we will show later than propagating only the length-lex bounds is sufficient. Line 15 breaks the symmetry between elements. Line 19 is the binary `exact1` constraint for the subset-bound domain. Line 22 channeling the primal and dual variables. Lines 26-28 removes symmetry among blocks. Lines 30–33 is a vanilla labeling method, which labels primal variables sequentially, it tries to include an element first, and excludes it after failure.

The Length-Lex Model We obtain the length-lex model by removing lines 18–19, the binary `exact1` constraint for subset-bound variables. The subset-bound variables `sbx[xi]` and `sby[yi]` remain in the model for the purpose of breaking value symmetry.

```
12 int propLimit = 12;
13 solve<cp> {
14   forall(xi in V : xi > 1)
15     cp.post(exactLe(llx[xi],llx[xj],1,propLimit));
16
17   forall(xi in V)
18     cp.post(channel(llx[xi],sbx[xi]));
19
20   cp.post(channeling(sbx,sby));
21
22   forall(yi in B, yj in B : yi < yj)
23     cp.post( lexleq( all(ye in V) sby[yj].getRequired(ye),
24                      all(ye in V) sby[yi].getRequired(ye) ));
25 }
```

Figure 11.4: COMET Model for Steiner Triple System in Length-Lex

11.3.4 Discussion

We evaluate our proposal and compare with previous proposed techniques for solving steiner triple system. Figure 11.3 presents the result. Both length-lex based models, Length-Lex and LS-Domain, solves all the instances in the least amount of time. They are able to solve three more instances unsolvable by previous approaches. They are both several orders of magnitude faster than previous formulations. It is interesting to point out that, both models achieves no failure (except the instances $v = 33$), simply computing the length-lex bound suffice to solve the problem efficiently. Moreover, the difference between length-lex and ls-domain is negligible, which suggests that the subset-bound component plays almost no importance in solving the problem.

11.3.5 A Close Look

The two length-lex based models drastically outperform earlier attempts. To have a better understanding of why they are doing better, we closely examine the contribution of each component in our model. In this section, we isolate different propagators and study their performance.

Binary Length-Lex Constraints Perhaps the simplest model would be the one using binary length-lex intersection constraints. It is shown in figure 11.4. The initialization and search procedures are identical to the previous model hence skipped. A binary intersection constraint `exactLe` is posted for every pair of length-lex variable in the primal model. It is the only set of basic constraints

in the model, the remaining are auxiliary variables for labeling and symmetry breaking. It takes polynomial time to enforce bound consistency on such binary constraint (Theorem 6). However, such constraint takes exponential time to converge (Theorem 14). To remedy this problem, for each binary intersection constraint, we limit the number of propagations per choice point by setting `propLimit = 12`. It is passed to the propagator as the last parameter.

We compare three variations. The first is a binary model with propagation limit. The second is the same but without propagation limit. The three is a model using exponential `exact1` constraints. Since the only function for the exponential constraint is to speed up propagation for the length-lex bound, no propagation is incurred. The second and third model should have a similar number of failures.

Table 11.4 presents the result. The binary length-lex model is able to solve up to $v = 39$. The second model, the one without propagation limit, takes too much time in bouncing between different bounds. The largest instance it can solve is $v = 21$, in which it takes much more time to find a solution. Despite the second model has no failure, it is outperformed by the first one. The exponential propagator is introduced to tackle the problem of exponential convergence rate, it examines all bound variables at once and exploits their semantics. As a result, it enjoys the best of the two worlds, there are very few failures and all the instances are solved efficiently.

Exponential Propagators Now, we evaluate the impact of the exponential propagator using ls-domain. We compare three models. First, a basic model in which only binary constraints are used. Second, a model using exponential constraint which propagates all four bounds of the ls-domain. Third, a model using exponential constraint which propagates only the two length-lex bounds of the ls-domain.

Table 11.5 illustrates the result. The subset-bound component is a complement to the length-lex domain. We are able to achieve stronger propagation using the ls-domain. Comparing it with the binary length-lex model, it cuts the search tree size by more than half in larger instances, the time is also reduced. We gain almost nothing in propagation when all four ls-domain bounds are propagated, since we have shown that only updating the length-lex bound is effectively enough for solving the steiner triple system. In the second column, the exponential constraint is taking much more time than the one which only propagates the length-lex bound. And because it is so heavy, it

is incapable to solve larger instances.

Method	Cardinal	Matrix Lex		Matrix LexSum		ValPrec-Set		ROBDD Domain		BDD -SAT	BDD -SAT	Length-Lex		LS-Domain	
Heuristic	Static	Static		Static		Min-Dom		Static		Static	VSIDS	Static		Static	
CPU	P4 2.4GHz	P3 1GHz		P3 1GHz		Sun Blade 1000		P4 2.8GHz		3GHz	3GHz	C2D 2.4GHz		C2D 2.4GHz	
Time-out	900s	3600s		3600s		7200s		600s		600s	600s	1800s		1800s	
v	Time	Time	Fails	Time	Fails	Time	Fail	Time	Fails	Time	Time	Time	Fails	Time	Fails
7	0.01	**0**	2	**0**	1	**0**	12	0.1	8	0.01	0.03	0.01	**0**	0.01	**0**
9	0.05	0.1	336	0.1	250	0.03	153	0.1	9	0.02	0.02	**0.01**	**0**	**0.01**	**0**
13	0.61					1738.24	3935567	109.2	24723	0.06	0.02	**0.02**	**0**	0.04	**0**
15	0.91							1.3	**0**	0.07	0.32	**0.05**	**0**	0.06	**0**
19	7.94									0.37	0.07	**0.18**	**0**	0.23	**0**
21	39.07									0.82	39.19	**0.41**	**0**	0.42	**0**
25										7.1	x	**0.98**	**0**	1.02	**0**
27										12.88	229.59	**1.19**	**0**	1.67	**0**
31	48.52							x	x	5.38	x	**2.8**	**0**	3.53	**0**
33										443.07	19.3	**7.3**	**1**	7.37	**1**
37												**36.06**	**0**	37.05	**0**
39												32.49	**0**	**32.38**	**0**
43												555.26	**0**	**538.88**	**0**

Table 11.3: Steiner Triple System: Comparing Length-Lex with Earlier Attempts.

	Length-Lex Binary, Limit = 12		Length-Lex Binary, No Limit		Length-Lex LL-EXP	
v	Time	Fails	Time	Fails	Time	Fails
7	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
9	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
13	0.03	**0**	0.04	**0**	**0.02**	**0**
15	0.06	**0**	0.1	**0**	**0.05**	**0**
19	0.26	15	1.32	**0**	**0.18**	**0**
21	0.62	69	6.4	**0**	**0.41**	**0**
25	3.33	1698	x	x	**0.98**	**0**
27	5.36	3037	x	x	**1.19**	**0**
31	4.01	**0**	x	x	**2.8**	**0**
33	32.81	13608	x	x	**7.3**	**1**
37	1719.04	668019	x	x	**36.06**	**0**
39	495.01	126603	x	x	**32.49**	**0**
43	x	x	x	x	**555.26**	**0**

Table 11.4: Steiner Triple System: Three Length-Lex Models.

	LS-Domain Binary, Limit=12		LS-Domain LS-EXP		LS-Domain LL-EXP	
v	Time	Fails	Time	Fails	Time	Fails
7	**0.01**	**0**	**0.01**	**0**	**0.01**	**0**
9	**0.01**	**0**	0.02	**0**	**0.01**	**0**
13	0.05	0	0.09	**0**	**0.04**	**0**
15	0.08	0	0.22	**0**	**0.06**	**0**
19	0.32	12	0.84	**0**	**0.23**	**0**
21	0.64	53	1.65	**0**	**0.42**	**0**
25	3.95	1532	13.26	**0**	**1.02**	**0**
27	6.69	2685	33.61	**0**	**1.67**	**0**
31	4.6	**0**	14.23	**0**	**3.53**	**0**
33	24.37	6014	395.29	**0**	**7.37**	1
37	1097.62	282229	x	x	**37.05**	**0**
39	328.19	56786	x	x	**32.38**	**0**
43	x	x	x	x	**538.88**	**0**

Table 11.5: Steiner Triple System: Three LS-Domain Models.

11.4 The Error Correcting Code (Hamming Distance)

11.4.1 Problem Statement

The error correcting code problem is defined in terms of three parameters: (l, d, w). It is an optimization problem that finds the largest number of codewords satisfying the following constraints: a codeword is a 0/1-vector of length l, the sum of the vector is w, and every pair of codewords have a Hamming distance of at least d.

11.4.2 Earlier Work

The error correcting code is a challenging problem since it requires an optimality proof which requires exhausting the whole search tree. Sadler and Gervet introduced hybrid domain representation which boosters propagation of symmetry-breaking constraints[56]. Hawkins, Lagoon, and Stuckey used the binary-decision-diagram-based representation[34]. Gange, Lagoon, and Stuckey used the bdd library for clause generation, and applied learning algorithm in SAT solvers[23]. The last method has proved to be extremely efficient. Especially since the solver is revisiting many similar subtrees, the learnt clauses can be used repeatedly.

11.4.3 Model

Overview The problem can be modeled using set variables. Each 0/1-vector is the characteristic function of a set variable X_i, hence it draws elements from the universe $\{1, ..., l\}$. Its cardinality is w. Between every pair of vectors, the Hamming distance restriction is guaranteed by the *atmost-k* intersection constraint, i.e. $|X_i \cap X_j| \leq w - d/2$. The problem is essentially equivalent to the *countAtmost* problem (Definition **??**). Hence, we reduce an error correcting code instance (l, d, w) to a *countAtmost* instance $(l, w, w - d/2)$. The rest of this section focuses on how to solve the *countAtmost* problem.

The problem exhibits both variable and value symmetry. Every pair of vectors (variables) are interchangeable; every pair of elements (values) are also interchangeable. The model has to take care of it.

A few techniques introduced in this thesis will be used. First, the symmetry-breaking binary constraints. Second, the dual filter for global *atmost-k* constraint. Third, the primal/dual filter for global symmetry-breaking *atmost-k* constraint. Last, the exponential feasibility checker for symmetry-breaking constraint.

We begin our discussion with the basic model. Then, we will discuss more advanced modeling techniques. The original problem is an maximization problem, which can be solved as a series of decision problems. At each decision problem, a constraint is posted to restrict the optimal value. We start from the infeasible region, by setting the optimal value so high to the extend that it is trivially no solution. Then we decrease the optimal value by one at a time and solve the adjusted decision problem, until the problem becomes feasible and the optimal solution is found.

Basic Model: Decision Problem Figure 11.7 gives a complete model to the decision problem. We first discuss basic model (lines 1–21) and the search component (lines 47–53). The decision problem is defined in terms of four parameter: the number of elements in the universe n, the number of set variables m, the cardinality of set variables c, and the maximum intersection size between every pair of variables k. The 3-d array **int**`[,,] table` caches the solution of the optimization problem *countAtmost* to avoid recomputing the same problem. The primal length-lex variables `llx[M]` are the basic variables. The primal and dual subset-bound variables `sbx[M]` and `sby[N]` are auxiliary variables for breaking value symmetry and the dual filter for the global *atmost-k* constraint, which we will discuss later. The universe and cardinality constraint are defined in the variable declaration.

Lines 11-12 define the core restriction: every pair of variables share at most k elements. Since variables are interchangeable and combing propagators with symmetry-breaking constraints achieves strong propagation, we post the binary `atmostLe` constraint which is a symmetry-breaking intersection propagator. Lines 14–21 eliminate value symmetry using dual modeling techniques: the channeling constraint (lines 14–15) connects the primal length-lex variables with primal subset-bound variables, which are then dual channeled to the dual subset-bound variables, and an ordering constraint is posted to eliminate value symmetry. Lines 47–52 define the search procedure. Static labeling is applied. We label the variables in increasing order. Try to include the smallest element and exclude it during backtrack.

```
1  function bool bool_atmostk(int n, int m, int c, int k, int[,] table){
2    Solver<CP> cp();
3    range M = 1..m;
4    range N = 1..n;
5    LengthLexVar<CP> llx[M](cp,n,c);
6    var<CP>{set{int}} sbx[M](cp,N,c);
7    var<CP>{set{int}} sby[N](cp,M,0..m);
8    var<CP>{int} aux[M,N](cp,0..1);
9
10   solve<cp>{
11     forall (i in M, j in M : i < j)
12       cp.post( atmostLe(llx[i],llx[j],k) );
13
14     forall (i in M)
15       cp.post( channel(llx[i],sbx[i]) );
16
17     cp.post( dualChannel(sbx,sby) );
18
19     forall (e in N, ee in N : e < ee)
20       cp.post( lexleq( all(i in M)(sby[ee].getRequired(i)),
21                        all(i in M)(sby[ e].getRequired(i)) ));
22     forall (e in N){
23       cp.post( sby[e].getCardinalityVariable() <= countAtmost(n-1,c-1,k-1,table,param) );
24       cp.post( m - countAtmost(n-1,c,k,table,param) <= sby[e].getCardinalityVariable() );
25     }
26     cp.post( sum(e in N)(sby[e].getCardinalityVariable()) == m*c );
27     forall (e in 1..n-c){
28       int i = countAtmost(n-e,c,k,table,param);
29       if (i < 0 || i >= m) continue;
30       UList<CP> ub(c);
31       ub.put(0,e);
32       ub.setLntuple(1,n,c-1);
33       cp.post( ZeroToLL_LessEqC(llx[m-i],ub) );
34     }
35
36     forall (e in 1..k){
37       int i = countAtmost(n-e,c-e,k-e,table,param);
38       if (i < 0 || i >= m) continue;
39       UList<CP> lb(c);
40       lb.setFntuple(0,1,c);
41       lb.setFntuple(e-1,e+1,c-e+1);
42       cp.post( ZeroToLL_GreaterEqC(llx[i+1],lb) );
43     }
44     forall (i in M, e in N)
45       cp.post( aux[i,e] == 1-sbx[i].getRequired(e) );
46     cp.post( RowCol_FeasibilityChecker(m,n,aux,7) );
```

Figure 11.5: COMET Model for Error Correcting Code (Decision Problem)

Basic Model: Optimization Problem Lines 55-59 are the model for the optimization problem. It wraps the decision problem and starts from the infeasible region where the optimal value `m` is high. It decreases `m` by one at a time. When a solution is found, the corresponding value of `m` is optimal. The initial value of `m` is determined by the same argument we used for the dual filter in *atmost-k*

```
47   }using{
48     forall (i in M)
49       forall (e in N : !sbx[i].isRequired(e) && !sbx[i].isExcluded(e))
50         try<cp> cp.post(requiresValue(sbx[i],e));
51         | cp.post(excludesValue(sbx[i],e));
52   }
53   return (cp.getSolution() != null);
54 }
```

Figure 11.6: COMET Search Procedure for Error Correcting Code (Decision Problem)

constraint: an element cannot occur or be absent in too many variables respectively. Hence, the initial value is the sum of the two bounds (line 56).

Basic Model: Caching Subproblems' Solution Lines 60–70 are the table-lookup routine. The whole table is initialized to -1. When the set of parameters is not known, the routine attempts to find the solution either using a search (line 67), or by simple combinatorial arguments for simple cases. The results are stores in the table.

The Dual Filter for the Global *atmost-k* Constraint Chapter 8.3 discusses a dual filter for the global *atmost-k* constraint. It takes a dual perspective of the problem and imposes constraints on the dual variables, which map value to variable. It is from the intuition that an element cannot occur in too many variables and an element, as well, cannot be absent in too many variables. Cardinality constraints are imposed on the dual variables. Lines 22-26 shows the filters in our model. For each element `e`, constraints are posted to restrict the upper and lower bound of the dual variable associated with it, i.e. `sby[e]`. Line 26 is a redundant constraint on the sum of the cardinality of all dual variables.

The Primal/Dual Filter for the Global Symmetry-Breaking *atmost-k* Constraint Section 8.4 presents a primal/dual filter which pushes the lexicographical ordering into a global intersection constraint. Two domain reduction rules are introduced (Rule 2 and 3) and they are posted to the model before the search starts. Lines 27–43 illustrate the code. Lines 27–34 correspond to Rule 2 and Lines 36–43 correspond to Rule 3. The data structure `UList<CP>` is a tuple which supports most operation in length-lex. Lines 31,32,40,41 initialize the bound according to the domain reduction rule. Unary ordering constraints are then posted.

```
55 function int opt_atmostk(int n, int c, int k, int[,,] table){
56   int m = countAtmost(n-1,c-1,k-1,table) + countAtmost(n-1,c,k,table);
57   while (!bool_atmostk(n,m,c,k,table) m--;
58   return m;
59 }
```

Figure 11.7: COMET Model for Error Correcting Code (Optimization Problem)

```
60 function int countAtmost(int n, int c, int k, int[,,] table){
61   if (table[n,c,k] == -1){
62     if (k == 0)
63       table[n,c,k] = n/c;
64     else if (n < c)
65       table[n,c,k] = 0;
66     else
67       table[n,c,k] = opt_atmostk(n,c,k,table);
68   }
69   return table[n,c,k];
70 }
```

Figure 11.8: COMET Model for Error Correcting Code (Table Lookup)

The Exponential Feasibility Checker for Breaking Fully-Interchangeability The problem exhibits fully interchangeability, where both variable and value are interchangeable. Posting symmetry breaking constraint among variables and values respectively does not complete eliminate all symmetric solutions as we discussed in Section 10.2.1. A compete checker, which is able to eliminate all symmetric solution due to fully interchangeability, is proposed in the same section. The checker `RowCol_FeasibilityChecker` ensures that the solution is the lexicographically least solution in its symmetry class. It works on a matrix of variables. We transform our set variable model into a matrix model using the auxiliary variables `aux[M,N]` defined in Line 8. Lines 44-46 post the checker. From practical standpoint, in Line 46, a parameter `7` passed to the checker, it limits the maximum number of rows to be enumerated.

11.4.4 Discussion

We evaluate our contribution and compare with previous proposed techniques for solving the error correcting code problem. We use instances from Sadler and Gervet [56], the easy ones are omitted since they are solved in negligible time. Some larger instances are added. Figure 11.6 presents the result. Length-Lex is clearly the best result. Most of the instances now become trivial. For instance $(10, 4, 5)$, it reduces the running time by more than 80 times.

Method		ROBDD		BDD-SAT		Length-Lex	
		Domain					
Heuristics		Static		VSIDS		Static	
CPU		P4 2.8GHz		3GHz		C2D 2.4GHz	
Time-Out		600s		600s		1800s	
(l,d,w)	Opt	Time	Fails	Time	Fails	Time	Fails
(8,4,4)	14	1.6	224	**0.03**	61	0.05	**0**
(9,4,3)	12	11.3	5615	0.06	300	**0.05**	**1**
(9,4,4)	18	x	x	1.04	4466	**0.15**	**23**
(9,4,5)	18	x	x	4.03	21651	**0.19**	**26**
(9,4,6)	12	25.4	16554	**0.06**	256	0.07	**3**
(10,4,3)	13	x	x	2.37	16755	**0.11**	**42**
(10,4,4)	30	x	x	14.66	34503	**0.27**	**31**
(10,6,5)	6	26.7	16635	0.03	145	**0.05**	**14**
(10,4,5)	36	x	x	104.39	184051	**1.23**	**1127**
(10,4,6)	30	x	x	48.96	131379	**1.29**	**1163**
(10,4,7)	13	x	x	1.96	13533	**0.15**	**15**
(11,4,3)	17					**0.19**	**211**
(11,2,3)	165					**4.57**	**0**
(11,6,4)	6					**0.07**	**46**
(11,6,5)	11					**0.09**	**14**

Table 11.6: Error Correcting Code (Hamming Distance): Comparing Length-Lex with Earlier Attempts

11.4.5 A Close Look

There are four main modules in our model: the basic model, the dual filter for the global *atmost-k* constraint, the primal/dual filter for the global symmetry-breaking *atmost-k* constraint, and the exponential symmetry-breaking feasibility checker. This section studies the contribution of each module.

Table 11.7 and 11.8 reveal the contribution of each component. It is interesting to compare the difference between the model where only *atmost-k* or *atmost-k*$_{\preceq}$ (but not both) is used. There are some cases where using the symmetry-breaking global propagator is better (instances $(9, 4, 4)$ and $(11, 4, 3)$), and there are some cases where using the global *atmost-k* propagator is better (instances $(10, 4, 5)$ and $(10, 4, 6)$). It suggests that these two components work orthogonally and prune different part of the search tree. Hence, when both are used together, the improvement is significant. Using the exponential feasibility checker RowCol_Fc improves the search for large instances too.

Indeed, the dual model, primal/dual filter, and the feasibility checker are independent of the

Domain		Length-Lex							
atmost-k (Fig. 8.3)				✔				✔	
atmost-k$_{\preceq}$ (Rules 2 & 3)						✔		✔	
RowCol_FC (Spec. 11)									
(l,d,w)	Opt	Time	Fails	Time	Fails	Time	Fails	Time	Fails
(8,4,4)	14	0.02	2	0.02	2	**0.01**	**0**	0.02	**0**
(9,4,3)	12	0.03	84	**0.02**	64	**0.02**	**1**	**0.02**	**1**
(9,4,4)	18	13.25	46003	3.61	14229	0.1	31	**0.09**	**23**
(9,4,5)	18	21.83	66527	0.76	1931	0.49	1003	**0.14**	**208**
(9,4,6)	12	0.05	100	**0.03**	68	0.04	10	0.04	**3**
(10,4,3)	13	1.96	12399	1	6869	**0.05**	**42**	**0.05**	**42**
(10,4,4)	30	149.76	227707	3.62	14301	0.15	39	**0.13**	**31**
(10,6,5)	6	**0.03**	58	0.04	46	**0.03**	16	0.04	**14**
(10,4,5)	36	x	x	7.83	22352	29.11	14211	**3.53**	**6425**
(10,4,6)	30	512.88	676025	5.81	10434	26.36	17460	**4.57**	**8067**
(10,4,7)	13	2.82	12680	0.58	2384	**0.08**	32	0.09	**25**
(11,4,3)	17	40.72	167518	12.48	64772	**0.09**	**211**	0.1	**211**
(11,2,3)	165	4.73	**0**	4.09	**0**	**4.02**	**0**	4.06	**0**
(11,6,4)	6	**0.04**	173	**0.04**	97	**0.04**	80	**0.04**	**46**
(11,6,5)	11	0.06	86	0.06	58	**0.04**	17	0.05	**14**

Table 11.7: Error Correcting Code (Hamming Distance): Length-Lex Domain, A Close Look I

underlying domain representation. It can be used for any kind of domain representation. Despite some (e.g. length-lex) are better at propagating those constraints. We also evaluate the impact of our proposal on the subset-bound domain. Table 11.9 gives the result. The basic model, which all extra filters and checkers are absent, gives the worst result and cannot solve large instances. The one with both intersection filters works well and fast, but is not the most robust. The model which applies all techniques is the best one. It is able to solve all instances.

11.5 The Balanced Incomplete Block Design

11.5.1 Problem Statement

Definition An instance of balanced incomplete block designs (BIBD) is defined in terms of 5 parameters (v, b, r, k, λ). The task is to find an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks,

Domain		Length-Lex							
atmost-k (Fig. 8.3)				✔				✔	
atmost-k$_{\preceq}$ (Rules 2 & 3)						✔		✔	
RowCol_FC (Spec. 11)		✔		✔		✔		✔	
(l,d,w)	Opt	Time	Fails	Time	Fails	Time	Fails	Time	Fails
(8,4,4)	14	**0.05**	2	**0.05**	2	**0.05**	**0**	**0.05**	**0**
(9,4,3)	12	0.07	84	0.07	64	**0.05**	**1**	**0.05**	**1**
(9,4,4)	18	2.12	4173	0.96	1817	0.16	31	**0.15**	**23**
(9,4,5)	18	3.16	5045	0.67	721	0.35	273	**0.19**	**26**
(9,4,6)	12	0.1	90	0.1	64	0.08	10	**0.07**	**3**
(10,4,3)	13	0.78	2199	0.43	1153	**0.1**	**42**	0.11	**42**
(10,4,4)	30	75.63	96329	1.1	1889	0.28	39	**0.27**	**31**
(10,6,5)	6	0.05	58	0.06	46	**0.05**	16	**0.05**	**14**
(10,4,5)	36	x	x	2.66	3614	6.29	2837	**1.23**	**1127**
(10,4,6)	30	35.52	41459	1.95	1996	5.38	2870	**1.29**	**1163**
(10,4,7)	13	0.51	774	0.37	456	0.16	22	**0.15**	**15**
(11,4,3)	17	6.7	24474	2.82	11776	**0.19**	**211**	**0.19**	**211**
(11,2,3)	165	4.59	**0**	**4.57**	**0**	4.61	**0**	**4.57**	**0**
(11,6,4)	6	**0.07**	173	0.08	97	**0.07**	80	**0.07**	**46**
(11,6,5)	11	**0.09**	86	0.1	58	**0.09**	17	**0.09**	**14**

Table 11.8: Error Correcting Code (Hamming Distance): Length-Lex Domain, A Close Look II

and every pair of objects occur together in exactly λ blocks.

11.5.2 Earlier Work

Meseguer and Torras introduced variable heuristics and domain reduction techniques which eliminate symmetry [51]. Prestwich gave a SAT encoding for the problem [52]. Flener, Frisch, Hnich, Kiziltan, Miguel, Pearson, and Walsh applied a 0/1-matrix model, and utilized the lexicographic-ordering constraint between pairs of row and column vectors [17]. Hnich, Kiziltan, and Walsh introduced a global constraint which combines the lexicographic-ordering and sum propagators [37]. It is interesting to point out that despite there is a trivial transformation from the 0/1-matrix model to set model, to the best of our knowledge, there was no attempt of using set variable for solving this problem.

Domain		Subset-Bound									
atmost-k				✔				✔		✔	
atmost-k$_{\preceq}$						✔		✔		✔	
RowCol_FC										✔	
(l,d,w)	Opt	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails
(8,4,4)	14	0.07	16	0.07	8	**0.06**	4	**0.06**	**1**	0.09	**1**
(9,4,3)	12	0.51	1997	0.15	351	**0.09**	53	**0.09**	**49**	0.12	**49**
(9,4,4)	18	x	x	223.09	574188	0.48	279	**0.45**	207	0.48	**143**
(9,4,5)	18	x	x	43.74	100988	13.64	50552	2.54	6749	1.08	**1551**
(9,4,6)	12	0.83	2614	0.18	345	0.18	385	**0.12**	66	0.15	**20**
(10,4,3)	13	812.6	2218607	24.93	65568	0.3	244	**0.28**	**190**	0.33	**190**
(10,4,4)	30	x	x	235.6	591496	1.38	1360	**1.13**	1072	1.21	**904**
(10,6,5)	6	0.05	131	0.06	67	**0.04**	38	0.05	**22**	0.06	**22**
(10,4,5)	36	x	x	x	x	x	x	x	x	**339.11**	**388369**
(10,4,6)	30	x	x	x	x	x	x	x	x	**412.968**	**469265**
(10,4,7)	13	x	x	24.3	64158	0.57	1487	0.39	517	**0.36**	**105**
(11,4,3)	17	x	x	885.1	1952008	0.9	2137	**0.85**	**1661**	0.92	**1661**
(11,2,3)	165	11.71	140	11.67	119	11.6	84	**11.47**	**56**	13.26	**56**
(11,6,4)	6	0.15	698	0.1	170	**0.08**	249	0.09	97	0.1	**77**
(11,6,5)	11	0.16	396	0.11	107	**0.08**	55	0.09	**30**	0.13	**30**

Table 11.9: Error Correcting Code (Hamming Distance): Subset-Bound Domain

11.5.3 Model

A length-lex model is used for evaluating our proposal with earlier work. Figure 11.9 gives the Comet model for the BIBD problem. The model is compact and neat. It mainly uses the length-lex variable `llx` for pruning. The subset-bound variables `sbx` are used for symmetry breaking. The main constraint is in Lines 11-12: `exactLe` constraint. It is a binary constraint which ensures the two variables share exactly `l` elements. There is a chance that these intersection constraints take forever to reach the fixpoint, we further introduce a propagation limit: the propagator is allowed to be invoked at most `v` times in each choice point.

Table 11.10 evaluates the performance. The length-lex model is generally the fastest and visit fewest failure nodes.

```
int v = 7;
int b = 7;
int r = 3;
int k = 3;
int l = 1; // lambda
LengthLexVar<CP> llx[1..v](cp,b,r);
var<CP>{set{int}} sbx[1..v](cp,1..b,r);

solve<cp>{

    forall(vi in 1..v, vj in 1..v : vi < vj)
        cp.post( exactLe(llx[vi],llx[vj],l,v) );

    forall (vi in 1..v)
        cp.post( channel(llx[vi],sbx[vi]) );

    forall (bi in 1..b)
        cp.post( sum(vi in 1..v)( sbx[vi].getRequired(bi) == true ) == k);

    forall (vi in 1..v, vj in 1..v : vi < vj)
        cp.post( lexleq( all(bi in 1..b)sbx[vj].getRequired(bi), all(bi in 1..b)sbx[vi].getRequired(bi) ) );

    forall (bi in 1..b, bj in 1..b : bi < bj)
        cp.post( lexleq( all(vi in 1..v)sbx[vi].getRequired(bj), all(vi in 1..v)sbx[vi].getRequired(bi) ) );

}using{
    forall (vi in 1..v)
        forall (bi in 1..b : !sbx[vi].isRequired(bi) && !sbx[vi].isExcluded(bi))
            try<cp> cp.post( requiresValue(sbx[vi],bi) );
            | cp.post( excludesValue(sbx[vi],bi) );
}
```

Figure 11.9: COMET Model for Balanced Incomplete Block Design Problem

Method Heuristics CPU Time-Out	Max-Variety Max-Degree Ultra60		Matrix Static PIII 1GHz 3600s		Length-Lex Static C2D 2.4GHz 1800s	
v,b,r,k,l	Time	Fails	Time	Nodes	Time	Fails
(6,20,10,3,4)	0.03	61	**0**	43	0.01	**0**
(6,30,15,3,6)	0.14	95	0.1	68	**0.02**	1
(6,40,20,3,8)	0.39	128	0.1	108	**0.03**	**8**
(6,80,40,3,16)	3.6	245	**0.1-1**	100-1000	**0.15**	**82**
(7,21,9,3,3)	0.05	75	**0**	42	0.01	**0**
(7,28,12,3,4)	0.12	86	0.1	64	**0.01**	**0**
(7,35,15,3,5)	0.27	109	0.1	88	**0.01**	**0**
(7,42,18,3,6)	0.48	139	0.2	115	**0.02**	**0**
(7,84,36,3,12)	4.2	254	**0.1-1**	100-1000	**0.05**	**0**
(7,91,39,3,13)	5.4	280	**0.1-1**	100-1000	**0.07**	**0**
(9,24,8,3,2)	2.7	252	0.1	48	**0.01**	**1**
(9,72,24,3,6)	2.7	252	**0.1-1**	100-1000	**0.12**	**70**
(9,84,28,3,7)	4.2	257	1 - 10	1000-10000	**0.22**	**130**
(9,96,32,3,8)	6.3	296	1 - 10	1000-10000	**0.37**	**224**
(9,108,36,3,9)	14	365	1 - 10	1000-10000	**0.61**	**332**
(9,120,40,3,10)	14	**268**	1 - 10	1000-10000	**0.98**	474
(10,90,27,3,6)	5.3	**289**	1 - 10	**100-1000**	**0.58**	532
(10,120,36,3,8)	13	**377**	**1 - 10**	**1000-10000**	**2.44**	1710
(11,110,30,3,6)	16	**366**	**1 - 10**	**100-1000**	**3.65**	3173
(12,88,22,3,4)	5.1	**296**	**1 - 10**	**100-1000**	**2.88**	3071
(13,52,12,3,2)	2.9	218	**0.1-1**	100-1000	**0.11**	**29**
(13,78,18,3,3)	3.5	282	**0.1-1**	**100-1000**	**0.32**	**170**
(13,104,24,3,4)	8.7	**344**	1 - 10	100-1000	**0.78**	480
(15,21,7,5,2)	5.5	**383**	**10 - 100**	100000-1000000	**27.1**	24038
(15,70,14,3,2)	5.5	**383**	**0.1-1**	100-1000	**0.16**	0
(16,32,12,6,4)	4.7	**485**	10 - 100	1000000-10000000	**2.99**	1603
(16,80,15,3,2)	4.7	485	1 - 10	**100-1000**	**0.64**	**377**
(19,57,9,3,1)	8.2	802	1 - 10	100-1000	**0.18**	**7**
(22,22,7,7,2)			**10 - 100**	100000-1000000	**81.37**	**19928**

Table 11.10: Balanced Incomplete Block Design Problem: Comparing with Earlier Work

	DoubleLex		RowCol-RowWise		ValRowCol-RowWise	
(q,l,d,v)	#s	time	#s	time	#s	time
(3,3,2,3)	6	**0.01**	6	**0.01**	**1**	**0.01**
(4,3,3,3)	16	0.07	8	0.05	**2**	**0.03**
(4,4,2,3)	12	0.02	12	0.03	**1**	**0.02**
(3,4,6,4)	11215	27.49	1427	13.38	**263**	**3.79**
(4,3,5,4)	61267	329.97	8600	117.3	**371**	**8.62**
(4,4,5,4)	72309	682.05	9696	252.15	**419**	**15.83**
(5,3,3,4)	21	1.56	5	1.04	**1**	**0.19**
(3,3,4,5)	71	0.69	18	0.39	**4**	**0.15**
(3,4,6,5)	77535	662.7	4978	130.33	**864**	**29.88**
(4,3,4,5)	2708	77.52	441	27.42	**27**	**2.98**
(4,4,2,5)	12	0.07	12	0.24	**1**	**0.06**
(4,4,4,5)	4752	137.03	717	54.55	**45**	**5.29**
(4,6,4,5)	7662	253.85	819	96.57	**51**	**8.98**
(5,3,4,5)	24619	1731.65	3067	573.38	**43**	**15.59**
(6,3,4,5)	x	x	x	x	**58**	**69.91**

Table 11.11: Equidistant Frequency Permutation Array Problem : RowWise

11.6 Evaluation of the Feasibility Checker

This section evaluates the performance of LexLeader feasibility checkers empirically. The primary goal is to assess the effectiveness of the approach, i.e., whether the reduction in search space outweighs the time spent in the feasibility checkers. Five benchmark problems were used, most of which can be found in the CSPLib. They are concerned with finding either all solutions or the optimal solution. Unless otherwise specified, variables are labeled in the symmetry-breaking order and values are tried in increasing order.

11.6.1 Equidistant Frequency Permutation Array problem (EFPA)

The task is to find a set of v codewords drawn from q symbols and each symbol appears for exactly λ times such that the Hamming distance between every pair of codeword is exactly d. The non-boolean model in [39] is used. The model is a $v \times q\lambda$ matrix of variables with domain $\{1, ..., d\}$. There are three main classes of symmetry: row interchangeability, column interchangeability, and value interchangeability. In [42], it was shown that completely breaking row and column interchangeability significantly reduces the number of solutions found. Our evaluation confirms this and pushes it further: Completely breaking row, column, and value interchangeability achieves the best runtime

	SnakeLex		RowCol-Snake		ValRowCol-Snake	
(q,l,d,v)	#s	time	#s	time	#s	time
(3,3,2,3)	6	**0.01**	6	**0.01**	**1**	**0.01**
(4,3,3,3)	16	0.06	8	0.05	**2**	**0.03**
(4,4,2,3)	12	0.02	12	0.03	**1**	**0.02**
(3,4,6,4)	10760	24.05	1427	14.79	**263**	**3.93**
(4,3,5,4)	58582	221.43	8600	96.88	**371**	**8.11**
(4,4,5,4)	66977	422.14	9696	187.46	**419**	**15.68**
(5,3,3,4)	20	0.76	5	0.55	**1**	**0.15**
(3,3,4,5)	71	0.55	14	0.38	**4**	**0.15**
(3,4,6,5)	71186	512.21	4876	128.22	**864**	**27.26**
(4,3,4,5)	2754	45.06	447	20.08	**27**	**2.45**
(4,4,2,5)	14	0.05	12	0.24	**1**	**0.06**
(4,4,4,5)	5354	83.34	822	42.43	**45**	**4.85**
(4,6,4,5)	21782	181.09	3017	117.8	**51**	**8.27**
(5,3,4,5)	28214	818.31	3523	337.19	**43**	**11.18**
(6,3,4,5)	x	x	x	x	**58**	**45.36**

Table 11.12: Equidistant Frequency Permutation Array Problem : Snake

performance.

The experiments compare several approaches. DoubleLex and SnakeLex post static symmetry-breaking constraints, while RowCol-RowWise, RowCol-Snake, ValRowCol-RowWise, and ValRowCol-Snake add a feasibility checker on the top of the static model. RowCol only considers row and column symmetries: all row symmetries are first enumerated and columns are then sorted as in Theorem 23. ValRowCol considers all symmetries: All value and row symmetries are enumerated and the columns are sorted (as in Theorem 26). The sorting uses a specific variable ordering (either RowWise or Snake), producing four feasibility checkers.

Table **??** presents the results. The feasibility checkers ValRowCol break all symmetries, find the fewest solutions and are the fastest. The improvements are more than 3 orders of magnitude when compared with the static methods and many times faster than the feasibility checkers breaking only row and column symmetries. The difference between the two variable orderings RowWise and Snake is small compared to the overall improvement, with a slight avantage to Snake. Note that both orderings achieve the same number of non-symmetric solutions since the choice of variable ordering has no effect on the number of solutions when the symmetry-breaking method is complete.

	DoubleLex		RowCol-RowWise		RowCol-RowWise (r=8)	
(v,k,l)	#s	time	#s	time	#s	time
(5,2,7)	**1**	**0.01**	**1**	0.05	**1**	0.05
(5,3,6)	**1**	**0.01**	**1**	0.02	**1**	0.02
(6,3,4)	21	**0.02**	**4**	0.09	**4**	0.1
(6,3,6)	134	**0.14**	**6**	0.18	**6**	0.21
(7,3,5)	33304	17.95	**109**	**4.49**	**109**	5.03
(7,3,6)	250878	177.29	**418**	**19.08**	**418**	21.54
(7,3,7)	1460332	1315.66	**1508**	**83.29**	**1508**	92.95
(8,4,6)	2058523	1341.93	**2310**	**73.35**	**2310**	82
(10,3,2)	724662	281.83	**960**	74.83	12563	**43.84**
(10,5,4)	8031	18.69	**21**	2.14	68	**1.91**
(22,7,2)	**0**	11.12	**0**	23.21	**0**	**2.86**

Table 11.13: Balanced Incomplete Block Design Problem : RowWise

	SnakeLex		RowCol-Snake		RowCol-Snake (r=8)	
(v,k,l)	#s	time	#s	time	#s	time
(5,2,7)	**1**	0.03	**1**	0.1	**1**	0.1
(5,3,6)	**1**	**0.01**	**1**	0.02	**1**	0.02
(6,3,4)	25	0.03	**4**	0.1	**4**	0.14
(6,3,6)	146	0.22	**6**	0.25	**6**	0.27
(7,3,5)	85242	51.78	**109**	8.05	**109**	8.98
(7,3,6)	566230	452.1	**418**	38.45	**418**	40.88
(7,3,7)	x	x	**1508**	182.92	**1508**	193.42
(8,4,6)	x	x	**2310**	150.29	**2310**	153.23
(10,3,2)	x	x	**960**	341.98	14420	203.8
(10,5,4)	13069	78.33	**21**	7.61	89	8
(22,7,2)	**0**	85.1	**0**	14.23	**0**	14

Table 11.14: Balanced Incomplete Block Design Problem : Snake

11.6.2 Balanced Incomplete Block Design (BIBD)

The experiments use the boolean fully-interchangeable matrix model from [17] and compare static methods and the LexLeader feasibility checkers. We label large value first. In some large instances, the number of rows becomes very large ($v = 10$ is a matrix with 10 rows and requires $10! = 3,628,800$ permutations). Hence, the experiments also evaluate the performance of the feasibility checker in which only the first k rows in the matrix are enumerated, for some specified value of k. Tables 11.13 and 11.14 present the result. The complete feasibility checkers return the fewest solutions and are generally faster than the static method. Checkers with a row limitation achieves the most robust results: They are up to 8 times faster on large instances (RowWise on (22,7,2)) and only slightly

	DoubleLex		RowCol-RowWise		ValRowCol-RowWise	
(t, k, g, b)	#s	Time	#s	Time	#s	Time
(2, 3, 2, 4)	2	**0.01**	2	**0.01**	**1**	**0.01**
(2, 3, 2, 5)	15	**0.01**	8	**0.01**	**4**	**0.01**
(2, 3, 3, 9)	12	**0.01**	6	**0.01**	**3**	**0.01**
(2, 3, 3, 10)	368	0.14	104	0.09	**21**	**0.05**
(2, 3, 3, 11)	6824	2.33	1499	1.26	**271**	**0.46**
(2, 3, 4, 16)	576	0.33	150	0.25	**15**	**0.32**
(2, 3, 4, 17)	43368	23.52	8236	11.93	**391**	**3.13**
(2, 3, 5, 25)	161280	134.91	27280	**77.71**	**283**	92.91
(2, 4, 2, 5)	10	**0.01**	5	**0.01**	**3**	**0.01**
(2, 4, 2, 7)	2285	1.06	333	0.32	**175**	**0.19**
(2, 4, 3, 9)	36	0.03	5	**0.02**	**2**	**0.02**

Table 11.15: Cover Array Problem

slower on others.

11.6.3 Cover Array problem (CA)

The experiments use the integrated model in [38] which has row, column, and value symmetries. The traditional comparisons are performed. However, since in earlier benchmarks, the impact of the variable ordering was negligible among complete checkers, only the RowWise ordering was considered for this, and subsequent, problems. Table 11.15 gives the results and the complete method is generally the fastest.

11.6.4 Error Correcting Code (Lee Distance)

This is CSPLib 036, an optimization problem whose the goal is to find the maximum number m of codeword of length n drawn from 4 symbols such that the Lee distance between every pair of codewords is exactly c. The decision problem is to find m codewords satisfying the constraints. The search starts with $m = 1$ and increases m by 1 each time. Optimality is proven when no solution is found. The model has both row and column symmetries and value symmetries discussed earlier in the paper. It is forbiddingly expensive to enumerate all possible value symmetries for each column. Instead the experiments use a number of feasibility checkers dealing with different combinations of symmetries. The results show that this approach dramatically reduces the search space.

		DoubleLex		LeeRowCol (k=8)		RowLee (k=5)		LeeRowCol (k=8) +RowLee (k=5)	
(n, c)	Opt	Time	Fails	Time	Fails	Time	Fails	Time	Fails
(4, 2)	8	1.44	21327	0.37	2882	**0.11**	253	0.12	**240**
(4, 4)	8	101.17	1834887	9.38	93085	1.29	4280	**0.88**	**2761**
(4, 6)	2	0.05	1211	0.03	337	**0.01**	**77**	**0.01**	**77**
(5, 2)	10	13.98	159808	1.75	11826	0.27	516	**0.26**	**435**
(5, 4)	8	·	·	425.09	4615063	13.87	63425	**6.05**	**27492**
(5, 6)	6	649.75	13466477	37.58	469869	0.4	2597	**0.32**	**1861**
(5, 8)	2	0.08	2152	0.04	509	**0.01**	**115**	0.02	**115**
(6, 2)	12	300.86	3114351	8.49	47859	0.69	1133	**0.53**	**752**
(6, 4)	8	·	·	·	·	39.39	246749	**12.79**	**79602**
(6, 8)	4	92.8	2187585	8.82	129252	**0.05**	584	0.07	**553**
(7, 2)	14	·	·	35.73	166890	1.85	2842	**1.09**	**1343**
(7, 4)	8	·	·	·	·	73.78	522444	**20.17**	**132809**
(8, 2)	16	·	·	156.45	599460	5.5	8011	**2.64**	**2701**
(8, 4)	8	·	·	·	·	154.18	972759	**30.55**	**183120**

Table 11.16: Error Correcting Code (Lee Distance)

More precisely, LeeRowCol enumerates all the value symmetries globally (like in value interchangeability), meaning that it ignores that each column can have its own symmetry. RowLee implements the algorithm from Theorem 27, except that the column symmetries are not enumerated. Both checkers have row limitation as well. Table 11.16 depicts the results. Due to its huge symmetry size, static symmetry-breaking approaches only solve a few small instances. RowLee produces significant improvements in performance: On instance $(5, 6)$, it reduces the time from 600 seconds to a fraction of a second. Overall, the combination of LeeRowCol and RowLee produces the best results as the rightmost column indicates. The performance improvements on this benchmark are spectacular.

Chapter 12

Conclusion

This thesis investigated the length-lex representation for set variables. The length-lex domain was proposed by Gervet and Van Hentenryck and was shown to offer some fundamental advantages over other set representations [29]. This thesis aimed at addressing the open issues about the theoretical and algorithmic properties, as well as about experimental behavior of the new domain. The main contribution of this thesis was to show that

length-lex is an effective set domain representation for constraint programming.

This statement was supported by a number of theoretical, algorithmic, and experimental contributions. In particular, the thesis made the following technical contributions.

1. **Generic and Efficient Basic Propagators.** It presented a generic and simple method for implementing polynomial-time bound-consistency propagators for unary and binary constraints.

2. **Global Symmetry-Breaking Constraints.** It gave novel symmetry-breaking techniques which exploit the semantic of the length-lex representation. It demonstrated the strength of combining symmetry-breaking constraints with other constraints both theoretically and empirically. It introduced a generic method for pushing symmetry-breaking constraints into arbitrary binary constraints.

3. **Dual Modeling for Breaking Value Symmetries.** It adapted the well-known dual modeling method for breaking value symmetries. It gave a soundness proof showing that it is feasible to enforce the length-lex ordering among dual variables.

4. **Exponential Propagators.** It improved the strength of length-lex domains by exploiting the fact that its constraint-propagation algorithm is more computationally expensive than that of subset-bound domains. It introduced exponential propagators which exploit constraint semantics and shift the exponential behavior from the constraint-propagation algorithm to the filtering algorithm. Experimental results show that models using the exponential propagators improves the performance of the constraint-propagation algorithm by a few orders-of-magnitudes while maintaining the same propagation strength.

5. **Global Intersection Checker and Filters.** It studied a number of global intersection set constraints which had drawn little attention because they are intractable. However, the exponential length-lex propagators suggest that it may be beneficial to shift some of the exponential behavior from the agnostic search component to filtering algorithms. It further exploited this idea by introducing a few heavy-weight checkers and filters for these constraint. What makes them particularly appealing is that they are independent of the underlying domain representation.

6. **Hybrid Domain Representations.** It presented a seamless integration method for length-lex with other domain representations, yielding a hybrid representation which provides a few orthogonal approximations to the domain. It offered two main contributions to the hybrid domain: It gave a light-weight method for using both multiple domains simultaneously and it introduced exponential propagators working on the product of two domains which allows stronger propagation.

7. **Exponential Checkers for Breaking Compositional Symmetries.** It addressed the problem of compositional symmetries, to account for the interplay of different classes of symmetry. Under traditional static methods, eliminating compositional symmetries is hard: Either an exponential number of ordering constraints are required or an exponential number of symmetric subtrees are not pruned away. The thesis tackled the problem by introducing an

exponential-time complete checker for classes of compositional symmetries.

8. **Extensive Evaluation.** It evaluated the contributions of each aforementioned components. It carried out an extensive comparison between the length-lex domain, and its variants, with earlier techniques on set domains over several standard benchmarks. The results indicated that the length-lex representation and its variants are very robust and efficient. They bring several orders-of-magnitudes improvement in performances over earlier techniques. They are able to solve many benchmark instances that were unsolvable before in constraint programming.

Perspective

The length-lex domain representation offered an orthogonal perspective of approximating the set domain. It is a rich representation which encapsulates a lot of domain information, making it a more precise approximation. Despite of the fact that it is computationally more expensive to propagate the length-lex domain than the classical subset-bound domain, the experimental results over several benchmarks indicate that the length-lex domain is much more robust and effective. Moreover, exponential checkers, filters and propagators proposed in this thesis bring significant performance gains too.

These results suggest that future research should consider effective propagation techniques for rich modeling objects. Existing techniques are usually based on modeling rich objects using a set of finite-domain variables, e.g., using 0/1-characteristic vectors for set variables. This thesis illustrated that the resulting propagation can be rather weak as it loses part of the problem structure. Constraint programming over rich combinatorial objects is thus promising avenue to remedy this limitation. For example, a permutation may use a totally-ordered representation which resembles the length-lex domain, a matrix may be decomposed into several vectors for better propagation, a sequence may utilize the regular graph for exact representation, and a graph may be represented as a collection of a few objects. Having accurate domain representations makes effective propagation possible.

Appendix A

Models

A.1 Social Golfer Model

A.1.1 Classical CSP

A model for the social golfer problem in classical CSP is given in figure A.1. Two sets of finite-domain variables are used to express constraints from different perspectives. The primal variables $X_{g,i,w}$ is the player who plays in position i of group g at week w (ex. A.1). The dual variables $Y_{p,w}$ is the group where player p plays in week w (ex. A.2). Ex. A.3 guarantees that in each week, golfers are allocated to groups of size S. The primal and dual variables are linked via the channeling constraints (ex. A.4). Ex. A.5 restrict that every two players play at most once.

This is a basic model that doesn't remove any symmetries.

We acknowledge that it is not necessary to have the primal variables $X_{g,i,w}$ as we can enforce a global cardinality constraint `gcc` on the dual variables $Y_{p,w}$ to make exactly s golfers play in a group in each week. We use the primal-dual model to make it consistent with the whole document.

A.1.2 Set CSP

Figure A.2 gives a social golfer model in Set-CSP. $X_{w,g}$ is the primal variable which maps group g in week w to a set of players, i.e. a subset of $\mathcal{P}$. Y_p is the dual variable used for eliminating value

$$X_{g,i,w} \in \mathcal{P} \qquad \forall g \in \mathcal{G}, i \in \mathcal{S}, w \in \mathcal{W} \tag{A.1}$$
$$Y_{p,w} \in \mathcal{G} \qquad \forall g \in \mathcal{G}, w \in sW, p \in \mathcal{P} \tag{A.2}$$
$$\texttt{gcc}([X_{1,1,w}, ..., X_{g,s,w}], [1, 2, .., G], [S, ..., S]) \qquad \forall w \in \mathcal{W} \tag{A.3}$$
$$X_{g,i,w} = p \Leftrightarrow Y_{p,w} = g \qquad \forall g \in \mathcal{G}, i \in \mathcal{S}, w \in \mathcal{W}, p \in \mathcal{P} \tag{A.4}$$
$$Y_{p,w} = Y_{p',w} \Rightarrow Y_{p,w'} \neq Y_{p',w'} \qquad \forall p, p' \in \mathcal{P}, \forall w, w' \in \mathcal{W} \text{ s.t. } p \neq p', w \neq w' \tag{A.5}$$

Figure A.1: Social Golfer Model in Classical CSP

$$X_{w,g} \subseteq \mathcal{P} \qquad \forall g \in \mathcal{G}, w \in \mathcal{W} \tag{A.6}$$
$$|X_{w,g}| = S \qquad \forall g \in \mathcal{G}, w \in \mathcal{W} \tag{A.7}$$
$$X_{w,g} \cap X_{w,g'} = \emptyset \qquad \forall g < g' \in \mathcal{G}, w \in \mathcal{W} \tag{A.8}$$
$$|X_{w,g} \cap X_{w',g'}| \leq 1 \qquad \forall g, g' \in \mathcal{G}, w < w' \in \mathcal{W} \tag{A.9}$$
$$X_{w,g} \prec X_{w,g'} \qquad \forall g < g' \in \mathcal{G}, \forall w \in \mathcal{W} \tag{A.10}$$
$$X_{w,1} \prec X_{w',1} \qquad \forall w < w' \in \mathcal{W}. \tag{A.11}$$
$$Y_p \subseteq \{(w, g) \mid g \in \mathcal{G}, w \in \mathcal{W}\} \qquad \forall p \in \mathcal{P} \tag{A.12}$$
$$p \in X_{w,g} \Leftrightarrow (w, g) \in Y_p \qquad \forall g \in \mathcal{G}, w \in \mathcal{W}, p \in \mathcal{P} \tag{A.13}$$
$$Y_p \prec Y_{p'} \qquad \forall p < p' \in \mathcal{P} \tag{A.14}$$

Figure A.2: Social Golfer Model in Set-CSP

symmetry, an ordering constraint is post between very pair of players.

Figure A.3 is the model proposed by Barnier and Brisset [3]. The matrix of finite-domain variables $Y_{p,w}$ are used in the dual model for eliminating value symmetry. This model has advantage over the one which uses dual set variables since a player plays exactly once each week. Using a vector of variables gives better propagation. Indeed, most of the experiments we conduct in this thesis are based on this model.

A.2 Steiner Triple System Model

Given V elements. The goal is to find B triples, each of which consists of 3 elements, such that every pair of elements occurs in exactly one triple. The system is indeed a special case of balanced incomplete block design by setting $B = V(V-1)/6$, $R = (V-1)/2$, $K = 3$, and $\lambda = 1$ [9].

We have $X_v \subseteq \{1, ..., V\}$ represents the set of blocks in which element v occurs in, and $Y_b \subseteq \{1, ..., B\}$ the set of elements occur in block b. Figure A.4 presents the model. The first five

$$X_{w,g} \subseteq \mathcal{P} \qquad \forall g \in \mathcal{G}, w \in \mathcal{W} \tag{A.15}$$
$$|X_{w,g}| = S \qquad \forall g \in \mathcal{G}, w \in \mathcal{W} \tag{A.16}$$
$$X_{w,g} \cap X_{w,g'} = \emptyset \qquad \forall g < g' \in \mathcal{G}, w \in \mathcal{W} \tag{A.17}$$
$$|X_{w,g} \cap X_{w',g'}| \leq 1 \qquad \forall g, g' \in \mathcal{G}, w < w' \in \mathcal{W} \tag{A.18}$$
$$X_{w,g} \prec X_{w,g'} \qquad \forall g < g' \in \mathcal{G}, \forall w \in \mathcal{W} \tag{A.19}$$
$$X_{w,1} \prec X_{w',1} \qquad \forall w < w' \in \mathcal{W}. \tag{A.20}$$
$$Y_{p,w} \in \mathcal{G} \qquad \forall w \in \mathcal{W}, p \in \mathcal{P} \tag{A.21}$$
$$Y_{p,w} = g \Leftrightarrow p \in X_{w,g} \qquad \forall w \in \mathcal{W}, g \in \mathcal{G}, p \in \mathcal{P} \tag{A.22}$$
$$[Y_{p,1}, ..., Y_{p,W}] \leq_{lex} [Y_{p',1}, ..., Y_{p',W}] \qquad \forall p < p' \in \mathcal{P} \tag{A.23}$$

Figure A.3: Social Golfer Model by Barnier and Brisset [3]

$$X_v \subseteq B \qquad \forall v \in V \tag{A.24}$$
$$Y_b \subseteq V \qquad \forall b \in B \tag{A.25}$$
$$|X_v| = r \qquad \forall v \in V \tag{A.26}$$
$$|Y_b| = 3 \qquad \forall b \in B \tag{A.27}$$
$$|Y_b \cap Y_{b'}| = 1 \qquad \forall b < b' \in B \tag{A.28}$$
$$X_v \preceq X_{v'} \qquad \forall v < v' \in V \tag{A.29}$$
$$Y_b \preceq Y_{b'} \qquad \forall b < b' \in B \tag{A.30}$$

Figure A.4: Steiner Triple System Model in Set-CSP

expressions are the basic constraints. The last two are symmetry breaking constraints since both blocks and elements are interchangeable.

Appendix B

Specialized Propagators

B.1 Unary Constraints

B.1.1 Overview

In Chapter 4, we introduced a generic algorithm for enforcing bound-consistency for unary constraints. The algorithm only depends on a feasibility routine which takes an interval and returns a boolean value indicating whether the interval contains a solution. The algorithm performs, essentially, a binary search on it and locates the new bounds. The total runtime for enforcing bound consistency is $O(c^2 + c\alpha \log n)$, where n is the number of elements in the universe, c is the upper bound cardinality, and the feasibility routine takes $O(\alpha)$ time.

Indeed, it is possible to implement a more efficient bound consistency propagation by exploiting the fact that in the binary search, the parameters for the feasibility routine share are lot of common structure between successive calls. We can take advantage of this fact and introduce different schema for implementing bound-consistent propagators.

We first review the idea of enforcing bound consistency.

Example 51 (Enforcing Bound Consistency)**.** Consider the length-lex domain $X_{ll} = ll\langle\{1,2,3\},\{3,5,6\},6\rangle$ and the unary constraint $\mathcal{R}_5(X) \equiv 5 \in X$. $\{1,2,3\}$ is not a solution and hence the input domain is not bound-consistent. The algorithm $bc\langle R_5\rangle(X_{ll})$ returns a bound-consistent

	`generic` (Chapter 4)	`specialized` (Chapter B.1.2)	`amortized` (Chapter B.1.3)
locate	$O(c\alpha + c^2)$	$O(c\alpha + c^2)$	$O(\gamma)$
construct	$O(c\alpha \log n)$	$O(\beta)$	$O(\beta + c)$

Figure B.1: Three Schemas for Bound-Consistent Algorithm on Unary Constraints

domain $ll\langle\{1,2,5\},\{3,5,6\},6\rangle$.

Example 52 (Detecting failure)**.** Consider the length-lex domain $X_{ll} = ll\langle\{2,3,4\},\{3,5,6\},6\rangle$ and unary constraint $\mathcal{R}_1(X) \equiv 1 \in X$. There is no possible successor of $\{2,3,4\}$ of cardinality 3 that could contain element 1. There is no solution and hence $bc\langle\mathcal{R}_1\rangle(X_{ll})$ returns $\perp$ that indicates failure.

We focus our attention on the $succ\langle\mathcal{C}\rangle(X_{ll})$ since the predecessor algorithm operates in a symmetrical manner.

The algorithm mainly bases on the decomposition idea illustrated in Chapter 4. There are two phases, *locate* and *construct*. In *locate* phase, the input length-lex interval is partitioned into some PF-intervals and the algorithm locates the first PF-intervals that contains a solution. In *construct* phase, the algorithm takes the PF-interval found in the previous phase, and constructs the smallest solution within it. We give three different schemas, namely `generic`, `specialized`, and `amortized`. Each of them corresponds to different implementation of the *locate* and *construct* phase. In `generic`, both phases only assume one boolean feasibility routine; in `specialized`, the *construct* phase depends on an additional constraint specific construction routine; while `amortized` involves extra book-keeping techniques in the *locate* phase and yield the best performance.

Figure B.1.1 illustrates the difference between difference schemas. $O(\alpha)$ is the time complexity for the feasible routine of PF-interval, $O(\beta)$ is the time for the specialized bound construction algorithm, and $O(\gamma)$ is the time for the specialized locate routine. In this chapter, we illustrate these schemas with the unary membership constraint ($R_e(X) \equiv e \in X$). We demonstrate it is possible to implement a bound-consistent algorithm just by providing a feasibility routine for a PF-interval, and it is also possible to attain a very efficient $O(c)$ bound-consistent algorithm using an amortization analysis. In particular, we show that for membership constraint, $O(\alpha) = O(\beta) = O(\gamma) = O(c)$.

Algorithm 17 $succ\langle\mathcal{R}_e\rangle(X_{pf} = pf\langle P, \check{f}, \hat{f}, n, c\rangle)$

```
1: if not hs⟨R_e⟩(X_pf) then
2:    return ⊥
3: flag ← false
4: for i ← 1 to |P| do
5:    s_i ← P_i
6:    flag ← flag ∨ (s_i = e)
7: f ← f̌
8: i ← |P| + 1
9: while i ≤ c do
10:   if i = c and flag = false then
11:      s_c ← e
12:   else
13:      s_i ← f
14:      flag ← flag ∨ (s_i = e)
15:      f ← f + 1
16:   i ← i + 1
17: return s
```

B.1.2 `specialized` Successor Construction Routine for $e \in X$ for PF-Interval

The `generic` schema applies a generic bound construction routine. In this section, we present a `specialized` schema, that replace the generic $succ\langle\mathcal{C}\rangle(X_{pf})$ algorithm by a constraint specific construction routine. For example, for unary membership constraint, it reduces the overall time complexity from $O(c^2 \log n)$ to $O(c^2)$.

Algorithm 17 ($succ\langle\mathcal{R}_e\rangle(X_{pf})$) demonstrates a specialized construction routine for the unary constraint $\mathcal{R}_e$. The algorithm first checks whether there is a solution (line 1), and return $\perp$ if there is none. Starting from line 3, we know the solution lies in somewhere in the PF-interval. The basic idea is the pick the smallest element while making sure that the feasibility condition holds. For unary membership constraint, the algorithm has to make sure that there is room for take the element e, if it is not taken yet. In Algorithm 17, it keeps a boolean $flag$ that indicates whether the element has been taken (lines 6, 14). When it reaches the last position, if $flag$ is false indicating the element is not taken yet, we must put element e to the last position (lines 10–11).

Example 53. Let a unary constraint be $\mathcal{R}_6(X)$, and a PF-interval $X_{pf} = pf\langle\{1\}, 3, 6, 8, 4\rangle$. After line 7, the prefix $\{1\}$ is assigned to s. The algorithm iterates from position 2 to 4. In the first

iteration, it assigns the smallest possible element to s (line 13), checks if that element is the required one (line 14), and moves to the next element. It does the same thing in the second iteration. In the last iteration, the $flag$ is still false since the required 6 has not be taken yet, it triggers the condition in line 10 and forces the required element to the last position (line 11). Afterwards, the algorithm returns $s = \{1, 3, 4, 6\}$, the smallest supported value in the input PF-interval.

Theorem 28. Algorithm 17 takes $O(c)$.

Proof. Lines 4–6 loops at most $O(c)$ times. Lines 9–16 loops at most $O(c)$ times. Every instruction inside the loop takes constant time. Hence it is $O(c)$ in total. □

Therefore, under the `specialized` schema, a specialized routine is used for bound construction. Applying Theorem 4, by setting $O(\alpha) = O(\beta) = O(c)$, the overall runtime for bound-consistent algorithm on unary membership constraint is $O(c^2)$.

B.1.3 `amortized` Successor Algorithm for Length-Lex Interval

The section present the `amortized` schema, further improves the runtime for bound-consistent algorithm by introducing a specialized routine for the *locate* phase. Recall the 2 phase routine presented in Algorithm 3, the *locate* phase takes at least $O(c^2)$ time since it is the cost for the decomposition routine.

The key observation is that the PF-intervals obtained from the decomposition are very similar. Two consecutive PF-intervals usually have similar prefix, the F-set are also closely related. The feasibility routine can therefore take into account of such common structure between consecutive intervals, avoid recomputing everything from scratch and amortize the overall run time.

We first take a close look to the decomposition and discuss the nice structure of some special length-lex interval. Then, we introduce the `amortized` schema and show how to implement a amortized *locate* phase for unary membership constraint.

The Decomposition Revisited

The decomposition algorithm (Algorithm 1) partitions an interval X into a head H, a body B, and a tail T. However, when the upper bound is maximal, the resulting PF-intervals exhibit a nice structure.

Example 54. Given the length-lex interval $ll\langle\{1,3,5,6\}, \bigtriangledown_4 = \{5,6,7,8\}, 8\rangle$ is decomposed into 5 PF-intervals.

$pf\langle \quad \{1,3,5\}, \quad 6, \quad 6, \quad 8, \quad 4 \quad \rangle$

$pf\langle \quad \{1,3,5\}, \quad 7, \quad 8, \quad 8, \quad 4 \quad \rangle$

$pf\langle \quad \{1,3\}, \quad 6, \quad 7, \quad 8, \quad 4 \quad \rangle$

$pf\langle \quad \{1\}, \quad 4, \quad 6, \quad 8, \quad 4 \quad \rangle$

$pf\langle \quad \{\}, \quad 2, \quad 5, \quad 8, \quad 4 \quad \rangle$

Formally, a length-lex interval $\langle\{l_1, l_2, .., l_c\}, \bigtriangledown_c, n\rangle$ can be decomposed into at most $c+1$ PF-intervals which falls into two categories: First the lower bound l itself

$$pf\langle l_{1..c-1}, l_c, l_c, n, c\rangle$$

and then remaining PF-intervals

$$pf\langle l_{1..i-1}, l_i + 1, n - c + i, n, c\rangle$$

if $l_i + 1 \leq n - c + i$ for $i \in \{1, ...c\}$.

Observe that the prefixes in the second category decreases by one element at a time, which will allow the algorithm to perform some incremental book-keeping and avoid having to compute the feasibility routine from scratch, amortizing its cost across the decomposition.

The Location Phase

The location routine combines the decomposition and feasibility routines and does not explicitly construct a list of PF-intervals. Its goal is similar to lines 1–6 of Algorithm 3 in that it finds the first PF-interval containing a feasible set. More precisely, it takes $ll\langle l, \bigtriangledown_c, n\rangle$ as input, and return the smallest supported PF-interval or $\perp$ if there is no solution.

Specification 14 ($locate_{succ}$)**.** Given $X_{ll} = ll\langle l, \bigtriangledown_c, n\rangle$,

$$locate_{succ}\langle\mathcal{C}\rangle(X_{ll}) \quad = \quad \begin{cases} \perp & \text{if } \nexists s \in X_{ll} : \mathcal{C}(s) \\ pf\langle l_{1..c-1}, l_c, l_c, n, c\rangle & \text{if } \mathcal{C}(l) \\ \max_{i \in 1..c}\{X^i_{pf} : hs\langle\mathcal{C}\rangle(X^i_{pf})\} & \text{otherwise} \end{cases}$$

Algorithm 18 $succ_A\langle\mathcal{C}\rangle(X_{ll} = ll\langle l, u, n\rangle)$

1: $X_{pf} \leftarrow locate_{succ}\langle\mathcal{C}\rangle(ll\langle l, \bigtriangledown_{|l|}, n\rangle)$ {*locate* phase}
2: **if** $X_{pf} = \bot$ **then** {*construct* phase}
3: **return** $\bot$
4: $l' \leftarrow succ\langle\mathcal{C}\rangle(X_{pf})$
5: **if** $l' \succ u$ **then**
6: **return** $\bot$
7: **return** l'

where $X^i_{pf} = pf\langle l_{1,..,i-1}, l_{i+1} + 1, n - c + i, n, c\rangle$.

Successor Algorithm

We modify the successor routine accordingly. Algorithm 18 ($succ_A\langle\mathcal{C}\rangle(X_{ll})$) implements the successor routine under the `amortized` schema. Instead of decomposing the input length-lex interval, the routine invokes $locate_{succ}$ and get the smallest supported PF-interval (line 1). It calls the specialized successor algorithm for PF-interval if there is a solution and get a new lower bound (line 4). Since we are not applied to $ll\langle l, \bigtriangledown_c, n\rangle$, it may happen that the result new lower bound l' is greater than the original upper bound u, we have to perform an extra check of domain consistency(lines 5–6).

Example 55. Suppose we have constraint $\mathcal{R}_5$, and the length-lex interval used in Example 54. The smallest PF-interval containing a solution is $pf\langle\{1\}, 4, 6, 8, 4\rangle$. We construct it using the *succ* routine (line 4) and we will get $\{1, 4, 5, 6\}$. The new lower bound is still in the original length-lex interval, hence we are good.

Example 56. Suppose the given length-lex interval is $ll\langle\{1, 3, 5, 6\}, \{1, 3, 7, 8\}$, the return value from $locate_succ$ is also $pf\langle\{1\}, 4, 6, 8, 4\rangle$ as the *locate* routine doesn't consider the upper bound. We invoke the *succ* routine as in the previous example, and we get $\{1, 4, 5, 6\}$. However, it exceeds the original upper and we can infer the domain is inconsistent.

Complexity Analysis

Theorem 29. Algorithm 18 ($succ_A\langle\mathcal{C}\rangle(X_{ll})$) takes $O(\gamma + \beta + c)$ time, where $O(\gamma)$ is the time complexity for $locate_{succ}\langle\mathcal{C}\rangle$.

Proof. Line 1 takes $O(\gamma)$, line 4 takes $O(\beta)$. Each remaining line takes $O(c)$. □ □

Algorithm 19 $locate_{succ}\langle\mathcal{R}_e\rangle(X_{ll} = ll\langle l, \bigtriangledown_{|l|}, n\rangle)$

```
1: if e ∈ l then
2:    return pf⟨l_{1..c−1}, l_c, l_c, n, c⟩
3: for i ← |l| downto 1 do
4:    if l_i + 1 ≤ e then
5:       return pf⟨l_{1,..,i−1}, l_{i+1} + 1, n − c + i, n, c⟩
6: return ⊥
```

`amortized` Locate Routine for $e \in X$

It remains to show how the *locate* routine that amortize the decomposition cost is implemented. Algorithm 19 ($locate_{succ}\langle\mathcal{R}_e\rangle$) implements Specification 14 for $\mathcal{C} = \mathcal{R}_e$.

Example 57. We use the length-lex interval in Example 54. The algorithm first check if the lower bound (i.e. $\{1, 3, 5, 6\}$) is a solution. Afterwards, in Lines 3–5, it starts looking from the smallest to the largest PF-interval. Notice that the difference between every successive PF-interval is one element in the prefix and the F-set. From line 1, we already infer that the element 4 does not belong to the lower bound, and hence also won't belong to any prefix, we don't have to consider it. Now, what is important is that we need to infer is from the range that PF-interval can take element from, if it contains 4, which can be done by a constant time check in line 4.

Theorem 30. Algorithm 19 ($locate_{succ}\langle\mathcal{R}_e\rangle(X_{ll})$) takes $O(c)$ time.

Proof. Line 1 takes $O(c)$. Lines 3–5 iterates at most $O(c)$ times and each line takes $O(1)$. □

Therefore, under the `amortized` schema, we have $O(\gamma) = O(c)$. The bound-consistent algorithm for unary membership constraint $\mathcal{R}_e(X) \equiv (e \in X)$ runs in $O(c)$ time.

B.2 Binary Constraints

B.2.1 Overview

Similar to unary constraint, we give three schemas of bound consistency algorithm. From a generic one which only depends on a feasibility routine, to a specialized which is most efficient but depends on some amortization analysis. Figure B.2.1 gives the time complexity of each schema.

	`generic` (Chapter 5)	`specialized` (Chapter B.2.2)	`amortized` (Chapter B.2.3)
locate	$O(c^2\alpha)$	$O(c^2\alpha)$	$O(c\gamma)$
construct	$O(c^2\alpha \log n)$	$O(c\beta)$	$O(c\beta)$

Figure B.2: Binary Constraint

Algorithm 20 $succ\langle\mathcal{D}\rangle(X_{pf} = pf\langle P_X, \check{f_X}, \hat{f_X}, n_X, c_X\rangle, Y_{pf})$

```
1: if not hs⟨D⟩(X_pf, Y_pf) then
2:    return ⊥
3: flag = ∃f ∈ {f̌_Y, ..., f̂_Y} : f ∉ P_X ∧ f < f̌_X
4: s_{1,...,|P_X|} ← P_X
5: cur ← f̌_X
6: for i = |P_X| + 1 to c_X do
7:    while cur ∈ P_Y do
8:       cur ← cur + 1
9:    if cur = f̂_Y ∧ not flag then
10:      cur ← cur + 1
11:   s_i ← cur
12:   cur ← cur + 1
13: return s
```

We will use binary disjoint constraint ($\mathcal{D}(X, Y) \equiv X \cap Y = \emptyset$) as example, and we will demonstrate that it is possible to achieve bound consistency for binary disjoint constraint in $O(c^2)$ time.

B.2.2 `specialized` Successor Algorithm for $X \cap Y = \emptyset$ for PF-intervals

In this section, we give the `specialized` schema. Likewise in unary constraints, this schema replaces the generic $succ\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ routine by a constraint specific successor algorithm. In binary disjoint constraint, with this routine available, we improve the time complexity of our algorithm from $O(c^3 \log n)$ to $O(c^3)$.

Algorithm 20 implements the successor algorithm for binary disjoint constraint that takes two PF-intervals. It constructs the smallest set in X_{pf} that can find a support from Y_{pf} with regard to binary disjoint constraint.

If the feasibility acknowledges the non-existent of solutions, it simply returns $\perp$(lines 1–2). In the rest of the routine, it essentially greedily appends the smallest possible element while maintaining the feasible condition. The algorithm first assignment the prefix to s (line 4) and then iterates over all remaining positions (lines 6–12). There are essentially two conditions need to be considered.

First, disjointness forbids X from taking any element from the prefix of Y (lines 7–8). Second, X cannot take all F-set elements of Y, since any set in Y needs at least one of them, and by disjointness X and Y cannot share the same element(lines 3,9–10). Variable *cur* marks the element that we are about to append. If Y_{pf} contains any F-set element that is unreachable by X, the second condition mentioned above is satisfied. The binary variable $flag$ is used to mark this condition (lines 3, 9–10). Once every position is filled, the routine returns the smallest supported set s in X_{pf}(line 13).

Example 58. Suppose $X_{pf} = pf\langle\{1\}, 2, 6, 7, 3\rangle$, $Y_{pf} = pf\langle\{2\}, 3, 4, 7, 3\rangle$. The algorithm first determines whether or not there is a solution (line 1), and return $\perp$ when inconsistency is detected by the feasibility routine. It then see if the F-set element requirement of Y is automatically satisfied, in this case, X can potentially takes all F-set element of Y, hence $flag$ is false(line 3). Since all sets in X_{pf} starts with the prefix, it assigns the prefix $\{1\}$ to s(line 4). As a result, we now have $s_1 = 1$ and it starts filling all remaining positions with the for-loop in lines 6–12. It begins with the smallest element and assigns 2 to *cur*. However, element 2 cannot be used as it is in the prefix of Y, and the algorithm proceeds to the next element (lines 7–8). It hasn't taken every possible F-set element from Y yet and therefore can be used. We have $s_2 \leftarrow 3$ (line 11). And we now move to the next position with $cur = 4$. It is not in the prefix and passes lines 7–8. However, it is the largest element of F-set of Y and we have to leave one element for Y. We cannot take this element and have to advance to the next. In this iteration, we have $s_3 \leftarrow 5$. All position of s are filled, the algorithm returns the new lower bound $\{1, 3, 5\}$.

Theorem 31. Algorithm 20 takes $O(c)$ time.

Proof. Line 1 takes $O(c)$. Line 3 takes $O(c)$, simply by checking if the holes lying between consecutive elements in P_X that belongs to the first-set of Y. Line 4 takes $O(c)$. The loop in lines 6–12 iterates $O(c)$ times. Cost for lines 7–8 can be amortized over the whole loop by maintaining an extra counter to store the most recently checked position in P_Y, the overall amortized cost is $O(c)$. Every remaining lines in the for-loop takes $O(1)$ time. Hence, it is $O(c)$ in total. □

Hence, under the `specialized` schema, by Theorem 6 and set $O(\alpha) = O(\beta) = O(c)$, enforcing bound consistency on binary disjoint constraint takes $O(c^3)$ time.

Algorithm 21 $succ_A\langle\mathcal{C}\rangle(X_{ll} = ll\langle l_X, u_X, n_X\rangle, Y_{ll} = ll\langle l_Y, u_Y, n_Y\rangle)$

Require: $n_X = n_Y$
1: $[Y^1_{pf}, Y^2_{pf}, ..., Y^j_{pf}] \leftarrow decomp(l_Y, u_Y, \emptyset, n_Y)$ {*locate* phase}
2: $X_{pf} \leftarrow \min_{Y'_{pf} \in [Y^1_{pf}, Y^2_{pf}, ..., Y^j_{pf}]}(locate_{succ}\langle\mathcal{C}\rangle(ll\langle l_X, \bigtriangledown_{|l_X|}, n_X\rangle, Y'_{pf}))$
3: **if** $X_{pf} = \bot$ **then** {*construct* phase}
4: **return** $\bot$
5: $l'_X \leftarrow \min_{Y'_{pf} \in [Y^1_{pf}, Y^2_{pf}, ..., Y^j_{pf}]}(succ\langle\mathcal{C}\rangle(X_{pf}, Y'_{pf}))$
6: **if** $l'_X \succ u_X$ **then**
7: **return** $\bot$
8: **return** l'_X

B.2.3 `amortized` Successor Algorithm for Length-Lex Intervals

This section presents the `amortized` schema. It improves the runtime for bound-consistent algorithm by amortizing the cost in the *locate* phase. The main observation is that the PF-intervals obtained from the decomposition enjoy a nice structure, we can exploit this structure and reduce the cost between every consecutive call to the feasibility routine. The idea is similar to the one proposed in the unary constraint section. It depends on a constraint specific locate routine that returns the first supported PF-interval, and then construct the smallest supported set presented in last section.

Specification 15 ($locate_{succ}$)**.** Given $X_{ll} = ll\langle l, \bigtriangledown_c, n\rangle$, Y_{pf}

$$locate_{succ}\langle\mathcal{C}\rangle(X_{ll}) \quad = \quad \begin{cases} \bot & \text{if not } hs\langle\mathcal{C}\rangle(X_{ll}, Y_{pf}) \\ pf\langle l_{1..c-1}, l_c, l_c, n, c\rangle & \text{if } hs\langle\mathcal{C}\rangle(\{l\}, Y_{pf}) \\ \max_{i \in 1..c}\{X^i_{pf} : hs\langle\mathcal{C}\rangle(X^i_{pf}, Y_{pf})\} & \text{otherwise} \end{cases}$$

where $X^i_{pf} = pf\langle l_{1,..,i-1}, l_{i+1} + 1, n - c + i, n, c\rangle$.

Following a generic BC algorithm and a specialized one for the disjoint constraint, we present an amortized algorithm that first locates the first element to be updated and constructs the new supported set.

Algorithm 21 implements the generic algorithm that allows us to amortize the cost in *locate* phase. It is similar to its unary counterpart. The input length-lex interval X_{ll} is not explicitly decomposed. Instead, the algorithm invokes a $locate_{succ}$ routine to find the first supported PF-interval. Since we decompose Y_{ll} into some PF-intervals, we have to compare against each of them. The min function returns the smallest PF-interval w.r.t to the length-lex ordering. After the PF-interval is located,

the algorithm *constructs* the smallest set(lines 5–8). Since different PF-intervals in Y_{ll} gives different support, we have to construct against each of them (line 5). Moreover, since the locate routine is not directly applied to the input interval, the bound may exceed the input upper bound. The algorithm has to perform an extra check for domain consistency(lines 6–7).

Theorem 32. Suppose $locate_{succ}\langle\mathcal{C}\rangle(X_{ll}, Y_{pf})$ runs in time $O(\gamma)$ and $succ\langle\mathcal{C}\rangle(X_{pf}, Y_{pf})$ run in time $O(\beta)$. Algorithm 21 ($succ_A\langle\mathcal{C}\rangle(X_{ll}, Y_{ll})$) takes $O(c^2 + c\gamma + c\beta)$ time.

Proof. Line 1 takes $O(c^2)$. Line 2 call $locate_{succ}$ at most $O(c)$ times, hence it is $O(c\gamma)$. Lines 3–4 takes $O(1)$. Line 5 make $O(c)$ calls to the successor routine, therefore $O(c\beta)$. Hence, $O(c^2 + c\gamma + c\beta)$ in total. □

Example 59. We use Example 26, we have $X_{ll} = ll\langle\{1, 2, 5\}, \{4, 6, 7\}, 7\rangle$ and the length-lex interval Y_{ll} is decomposed into two PF-intervals $Y^1_{pf} = pf\langle\emptyset, 1, 1, 7, 3\rangle$ and $Y^2_{pf} = pf\langle\{2\}, 3, 4, 7, 3\rangle$. Consider the binary disjoint constraint. In line 2, the algorithm invokes the $locate_succ$ routine for each PF-intervals in the decomposition of Y. In case Y^1_{pf}, the locate routine returns $pf\langle\emptyset, 2, 5, 7, 3\rangle$, whilst in case Y^2_{pf}, the locate routine returns $pf\langle\{1\}, 3, 6, 7, 3\rangle$. The PF-interval in the latter case is smaller, hence the new lower bound lies in that interval. That PF-interval is stored and used in the *construct* phase.

B.2.4 Locate for binary disjoint constraint

In the `amortized` schema, the algorithm relies on a constraint specific $locate_{succ}$ routine. We present such routine for the binary disjoint constraint. In short, the *locate* routine composes the decomposition and feasibility routine, it exploits the property that every consecutive PF-interval share many common structure. Hence, we only need to pay attention to the difference. Recall that decomposing a length-lex interval $ll\langle l, \bigtriangledown_c, n\rangle$ gives us a systematic list of PF-intervals. We, once again, state the decomposition here: A length-lex interval $\langle\{l_1, l_2, .., l_c\}, \bigtriangledown_c, n\rangle$ can be decomposed into at most $c + 1$ PF-intervals:

$$pf\langle l_{1..c-1}, l_c, l_c, n, c\rangle \tag{B.1}$$

$$pf\langle l_{1..i-1}, l_i + 1, n - c + i, n, c\rangle \quad i \in \{1..c\} \tag{B.2}$$

The first PF-interval is actually special case, it is a singleton which is the lower bound. Every consecutive pair of PF-intervals in the remaining list enjoys a systematic delta: 1. the prefixes is differ by one, 2. the lower bound of the first-set is always the maximum of prefix of the next PF-interval plus one, 3. the upper bound of the first-set is increased by one every time. Therefore, the *locate* routine can take advantage of this knowledge. Algorithm 22 implements Specification 15 for binary disjoint constraint. It *flattens* the feasibility routine for binary disjoint (Algorithm [**?**]), and basically carries out the same function. It first returns the PF-interval corresponding to the lower bound if it is a solution. Lines 4–28 is the core *locate* routine. It start by considering the PF-interval with the shortest prefix, which is also the largest with regard to the length-lex ordering. It performs a feasibility check. Then, it advances to the next PF-interval with a second shortest prefix, performs the feasibility check again, repeat until the last PF-interval. What it does is essentially equivalent to the decompose-then-check-feasibility algorithm under the `generic` schema. The key difference is that here we take the *change* between consecutive PF-intervals into account, and avoid the need to re-compute everything from scratch in every checks.

We move to the real story. We use $X_p f$ to mark the smallest PF-interval that contains a solution (lines 5,16,27,28). Similar to the feasibility routine for binary disjoint constraint, we need to separate into two case, $\check{f_X} \leq \check{f_Y}$ and $\check{f_X} > \check{f_Y}$. In the structural decomposition, the first case always happens before the second. Hence, we can use an index α (in line 5) the boundary between these two cases. The first case corresponds to lines 6–16, while the second case corresponds to lines 17–27. In the first case, $\check{f_X}$ is possibly smaller than the maximum element of P_Y, hence we have to take into account of it. $\check{f_X}$ is smaller infers that the prefix of X is also smaller, hence we can factor out the shorter prefix easily. We can see this problem as invoking Algorithm 9 by interchanging the role of X and Y. Hence, this part is almost equivalent to lines 12–17 in algorithm 9. We take a closer look at it. Line 7 forms the F_X and V_X as we did in the feasibility routine, notice that these are logical representation of ranges for the ease of demonstration and are not explicitly created. c'_X denotes the number of element required excluding the prefix. The variable $flag$ serves the same function too: after all test, if $flag$ remains true, it indicates the corresponding PF-interval contains a solution. Lines 9–10 is the only difference, we need to make sure both prefixes does overlap. Once they overlap, we know there couldn't be any solution afterwards, since we keep appending more elements to the prefix in the "locate" routine, and we can return the best index we found so far.

Line 11 checks if the remaining sub-universe has enough rooms to fulfill the cardinality requirement. Lines 12–14 checks if the first-set restriction could be satisfied. If the PF-interval passes all test, we can mark it as a potential solution (lines 15–16). For the second part (lines 18–27), they are essentially performing the same function.

Complexity Analysis

Lemma 11. Algorithm 22 ($locate_{succ}\langle\mathcal{D}\rangle(X_{ll}, Y_{pf})$)takes $O(c)$ time.

Proof. Line 2 invokes the feasibility routine for binary disjoint constraint for two PF-intervals, which takes $O(c)$. Notice we do not explicitly create $F_X, F'_X, V_X, F_Y, F'_Y, V_Y$, they are just logical representation of a range (with at most $O(c)$ "holes"), we can perform efficient operations on them with extra bookkeeping in $O(1)$ time (or a total amortized $O(c)$ at worst). Line 5 takes $O(c)$. α is bounded by c_X, hence loops in lines 6–16 and lines 18–27 iterates at most c_X times. Line 9 can be implemented by keeping an extra index for the last seen position in P_Y, that takes an overall amortized $O(c_Y)$ time. Line 10 can be done in an amortized $O(c_Y)$ time, thanks to the observation: $\min V_X$ is strictly monotonic increasing over the loop, hence after the first iteration (that requires $O(c_Y)$ to scan the whole P_Y), we only require to see how many elements in P_Y is lost due to the increase in $\min V_X$, elements that are lost won't go back to again due to monotonicity. Therefor, this line could be implement in an overall amortized $O(c_Y)$ time. Using similar arguments, all other lines (lines 12–14, 21–25) takes an amortized $O(c)$ time too. Therefore, algorithm 22 takes $O(c)$ time. □ □

Therefore, under the `amortized` schema, for binary disjoint constraint, we have $O(\alpha) = O(\beta) = O(\gamma)$ and therefore by Theorem 32, the bound-consistent algorithm takes $O(c^2)$ time.

Algorithm 22 $locate_{succ}\langle\mathcal{D}\rangle(X_{ll} = ll\langle l_X, \bigtriangledown_{|l_X|}, n_X\rangle, Y_{pf} = pf\langle P_Y, \check{f}_Y, \hat{f}_Y, n_Y, c_Y)\rangle$

Require: $n_X = n_Y$

$c_X = |l_X|$
if $hs\langle\mathcal{D}\rangle(pf\langle l_{X\,1,...,c_X-1}, l_{X\,c_X}, l_{X\,c_X}, n_X, c_X\rangle, Y_{pf})$ **then**
  return $pf\langle l_{X\,1,...,c_X-1}, l_{X\,c_X}, l_{X\,c_X}, n_X, c_X\rangle$
$F_Y, V_Y, c'_Y \leftarrow \{\check{f}_Y, ..., \hat{f}_Y\}, \{\check{f}_Y, ..., n_Y\}, c_Y - |P_Y|$
$X_{pf}, \alpha \leftarrow \emptyset, \arg\max_i\{0, 1 \leq i \leq c_X | l_{X\,i} < \check{f}_Y\}$
for $i \leftarrow 1$ to α **do**
  $F_X, V_X, c'_X \leftarrow \{l_{X\,i} + 1, ..., n_X - c_X + i\}, \{l_{X\,i} + 1, ..., n_X\}, c_X - i + 1$
  $flag = true$
  if $i > 1 \wedge l_{X\,i-1} \in P_Y$ **then**
    return min_i
  $flag \leftarrow flag \wedge (|V_X| \geq c'_X + c'_Y + |\{P_Y \cap V_X\}|)$
  $F'_X \leftarrow F_X \setminus P_Y$
  $flag \leftarrow flag \wedge (F'_X \neq \emptyset)$
  $flag \leftarrow flag \wedge (|F'_X \cup F_Y| \geq 2)$
  if $flag$ **then**
    $X_{pf} \leftarrow pf\langle l_{X\,1,..,i-1}, l_{X\,i+1} + 1, n - c + i, n, c\rangle$
$F'_Y \leftarrow F_Y \setminus l_{X\,1,...,\alpha-1}$
for $i \leftarrow \alpha + 1$ to c_X **do**
  $F_X, c'_X \leftarrow \{l_{X\,i} + 1, ..., n_X - c_X + i\}, c_X - i + 1$
  $flag = true$
  $flag \leftarrow flag \wedge (|V_Y| \geq c'_X + |\{P_X \cap V_Y\}| + c'_Y)$
  $F'_Y \leftarrow F'_Y \setminus \{l_{X\,i-1}\}$
  $flag \leftarrow flag \wedge (F_X \neq \emptyset)$
  $flag \leftarrow flag \wedge (F'_Y \neq \emptyset)$
  $flag \leftarrow flag \wedge (|F_X \cup F'_Y| \geq 2)$
  if $flag$ **then**
    $X_{pf} \leftarrow pf\langle l_{X\,1,..,i-1}, l_{X\,i+1} + 1, n - c + i, n, c\rangle$
return X_{pf}

Appendix C

Global Propagators for Subset-Bound Variables

C.1 Overview

This chapter reconsiders the deployment of synchronous optical networks (SONET), an optimization problem originally studied in the operation research community[60]. The SONET problem is defined in terms of a set of clients and a set of communication demands between pairs of clients who communicate through optical rings. The task is to allocate clients on (possibly multiple) rings, satisfying the bandwidth constraints on the rings and minimizing the equipment cost. This problem has been tackled previously using mixed integer programming (MIP)[60] and constraint programming (CP)[62, 57]. Much attention was devoted to variable branching heuristics and breaking ring symmetries (since all rings are identical). It was shown that sophisticated symmetry-breaking techniques dramatically reduce the computational times, both for MIP and CP formulations. The difficulty of finding good branching heuristics, which do not clash with symmetry breaking, was also mentioned.

This chapter takes another look at the problem and studies the possibility that the thrashing behavior experienced in earlier attempts is primarily due to lack of pruning. The key observation is that existing models mainly consist of binary constraints and lack a global perspective. Instead of focusing on symmetry breaking and branching heuristics, we study how to strengthen constraint propagation

		sb-domain	sbc-domain
NonEmptyIntersection	$\|X \cap Y\| \geq 1$	Polynomial	Polynomial (Thm. 34)
AllNonEmptyIntersection	$\forall i, \|X \cap Y_i\| \geq 1$	Polynomial (Thm. 35)	NP-hard (Thm. 37)
SubsetOfUnion	$\bigcup_i Y_i \supseteq X$	Polynomial (Thm. 40)	?
SubsetOfOpenUnion	$\bigcup_{i \in Y} X_i \supseteq s$	Polynomial (Thm. 44)	NP-hard (Thm. 45)

Figure C.1: Overview of Hardness of Complete Filtering Algorithms

by adding redundant global set-constraints. We propose two classes of redundant constraints and we investigate the complexity of these set constraints and the design of filtering algorithms. Like many other global constraints for set variables [55, 6], complete filtering algorithms are often intractable but we propose inference rules that can reduce the search space effectively. The considered set constraints, their complexity results, and some of the open questions, are summarized in Figure C.1. The technical results were evaluated experimentally on the standard SONET benchmarks. They indicate that the enhanced model, with static symmetry-breaking constraints and a static variable ordering, is many orders of magnitude faster than existing approaches.

This chapter is organized as follows. Section C.2 gives a formal description of the SONET problem and its CP model. Section C.3 recalls basic definitions about set domains and fixes the notation used in the paper. Sections C.4–C.8 constitute the core of the paper and study the various constraints used in the model. Section C.9 presents the experimental results and Section C.10 concludes the paper.

C.2 The SONET Problem

Problem Description The SONET problem [60] is a network topology design problem for optical fiber network, the goal is to find a topology that minimizes the cost such that all clients' traffic demands are met. An input instance is a weighted undirected demand graph $G = \langle N, E; d \rangle$, where each node $u \in N$ represents a client and weighted edges $(u, v) \in E$ correspond to traffic demands of a pair of clients. Demand $d(u, v)$ is always integral. Two clients can communicate only if both of them are installed on the same ring, which requires an expensive equipment called an add-drop multiplexer (ADM). A demand can be split into multiple rings. The input also specifies the maximum number of rings r, the maximum number of ADMs allowed on the same ring a, and the bandwidth capacity

of each ring c. A solution of the SONET problem is an assignment of rings to nodes and of capacity to demands such that 1) all demands of each client pairs are satisfied; 2) the ring traffic does not exceed the bandwidth capacity; 3) at most r rings are used; 4) at most a ADMs on each ring; and 5) the total number of ADMs used is minimized.

The Basic CP Model The core CP model [56, 62] include three types of variables: Set variable X_i represents the set of nodes assigned to ring i, set variable Y_u represents the set of rings assigned to node u, and integer variable $Z_{i,e}$ represents the amount of bandwidth assigned to demand pair e on ring i. The model is

$$\textit{minimize} \sum_{i \in R} |X_i| \quad \texttt{s.t.}$$

$$|Y_u \cap Y_v| \geq 1 \qquad \forall (u,v) \in E \tag{C.1}$$

$$Z_{i,(u,v)} > 0 \Rightarrow i \in (Y_u \cap Y_v) \qquad \forall i \in R, (u,v) \in E \tag{C.2}$$

$$\sum_{i \in R} Z_{i,e} = d(e) \qquad \forall e \in E \tag{C.3}$$

$$u \in X_i \Leftrightarrow i \in Y_u \qquad \forall i \in R, u \in N \tag{C.4}$$

$$|X_i| \leq a \qquad \forall i \in R \tag{C.5}$$

$$\sum_{e \in E} Z_{i,e} \leq c \qquad \forall i \in R \tag{C.6}$$

$$X_i \preceq X_j \qquad \forall i,j \in R : i < j \tag{C.7}$$

Constraint (C.1) ensures nodes of every demand pair lie on at least one common ring. Constraint (C.2) ensures that there is a flow for a demand pair on a particular ring i only if both client are on that ring. Constraint (C.3) guarantees that every demand is satisfied. Constraint (C.4) channels between the first two types of variables. Constraint (C.5) makes sure that there are at most a ADMs on each ring. Constraint (C.6) makes sure that the total traffic flow on each ring does not exceed the bandwidth capacity. Constraint (C.7) is a symmetry-breaking constraint that removes symmetric solutions caused by interchangeability of rings. Any total ordering on sets could be used for imposing the lexicographic constraint.

Extended Model Smith [62][Section 5] proposed a few implied constraints to detect infeasible assignments early in the search. For space reasons, we only show some of them which will be generalized by our redundant global constraints:

$$|X_i| \neq 1 \qquad \forall i \in R \tag{C.8}$$

$$|Y_u| \geq \lceil \frac{|\delta_u|}{a-1} \rceil \qquad \forall u \in N \tag{C.9}$$

$$Y_u = \{i\} \Rightarrow \delta_u \cup \{u\} \subseteq X_i \qquad \forall u \in N, i \in R \tag{C.10}$$

$$Y_u = \{i,j\} \Rightarrow \delta_u \cup \{u\} \subseteq X_i \cup X_j \qquad \forall u \in N, i,j \in R \tag{C.11}$$

In those constraints, δ_u denotes the neighbors of node u.

Our Extended Model We propose two constraints to boost propagation:

$$\bigcup_{i \in \delta_u} Y_i \supseteq Y_u \qquad \forall u \in N \tag{C.12}$$

$$\bigcup_{i \in Y_u} X_i \supseteq \delta_u \qquad \forall u \in N \tag{C.13}$$

The *subsetOfUnion* constraint (C.12) generalizes (C.8) and forces a node not to lie on rings with no contribution. The *subsetOfOpenUnion* constraint (C.13) generalizes (C.9), (C.10), and (C.11) and ensures that the rings of a node accommodate all its neighbors.

C.3 The Set Domains

Our algorithms consider both the traditional subset-bound domain and subset-bound with cardinality domain.

Definition 34. A subset-bound domain (sb-domain) $sb\langle R, P\rangle$ consists of a required set R and a possible set P, and represents the set of sets

$$sb\langle R, P\rangle \equiv \left\{ s \mid R \subseteq s \subseteq P \right\} \tag{C.14}$$

Definition 35. A subset-bound + cardinality domain (sbc-domain) $sbc\langle R, P, \check{c}, \hat{c}\rangle$ consists of a required set R and a possible set P, a minimum and maximum cardinalities $\check{c}$ and $\hat{c}$, and represents the set of sets

$$sbc\langle R, P, \check{c}, \hat{c}\rangle \equiv \left\{s \mid R \subseteq s \subseteq P \wedge \check{c} \leq |s| \leq \hat{c}\right\} \tag{C.15}$$

We now give the definition of bound consistency for these set domains.

Definition 36 (sbc-bound consistency)**.** A set constraint $\mathcal{C}(X_1, ..., X_m)$ (X_i are set variables using the sbc-domain) is said to be sbc-bound consistent if and only if $\forall 1 \leq i \leq m$,

$$\exists x_1 \in d(X_1), ..., x_m \in d(X_m) \text{ s.t. } C(x_1, ..., x_m) \tag{C.16}$$

$$\wedge \quad R_{X_i} = \bigcap_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} x_i \tag{C.17}$$

$$\wedge \quad P_{X_i} = \bigcup_{\forall 1 \leq j \leq m, x_j \in d(X_j) : C(x_1, ..., x_m)} x_i \tag{C.18}$$

$$\wedge \quad \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \text{ s.t. } |x_i| = c_{\check{X}_i} \wedge C(x_1, ..., x_m) \tag{C.19}$$

$$\wedge \quad \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \text{ s.t. } |x_i| = c_{\hat{X}_i} \wedge C(x_1, ..., x_m) \tag{C.20}$$

where $d(X_i) = sbc\langle R_{X_i}, P_{X_i}, c_{\check{X}_i}, c_{\hat{X}_i}\rangle$ denotes the domain of X_i.

The definition is similar for the subset-bound domain but it omits the cardinality rules. In the following, we use $bc_\theta\langle\mathcal{C}\rangle$ to denote a bound-consistency propagator (or complete filtering algorithm) for constraint $\mathcal{C}$ on a θ-domain. We call *free elements* the elements in the possible set that are not in required set and *empty spots* the maximum number of free elements that the set can include.

Example 60. Consider domain $sbc\langle\{1, 2\}, \{1, .., 6\}, 3, 5\rangle$. $\{3, 4, 5, 6\}$ are free elements and the domain has 3 empty spots since it can take at most 5 elements while 2 of them are already fixed by required set.

C.4 Non-Empty Intersection Constraint

Reference [62] does not specify how the constraint propagator for the non-empty intersection constraint($|X \cap Y| \geq 1$) is implemented. This section presents a sound and complete propagator for the sbc-domain.

First note that the sbc-domain gives stronger propagation than the sb-domain.

Theorem 33. Enforcing bound consistency on the conjunction of constraints

$$|X \cap Y| \geq 1 \wedge \check{c_X} \leq |X| \leq \hat{c_X} \wedge \check{c_Y} \leq |Y| \leq \hat{c_Y}$$

is strictly stronger for the sbc-domain than for the sb-domain.

Proof. Consider $X \in sb\langle\{1\}, \{1..5\}\rangle$, $Y \in sb\langle\{6\}, \{2, .., 6\}\rangle$, $\check{c_X} = \hat{c_X} = \check{c_Y} = \hat{c_Y} = 2$. For the sbc-domain, after enforcing bound consistency on each constraint, $X \in sbc\langle\{1\}, \{1, .., 4\}, 2, 2\rangle$ and $Y \in sbc\langle\{6\}, \{3, .., 6\}, 2, 2\rangle$. X and Y can each take two elements, one of which is fixed, and elements 2 in X and 5 in Y are removed. All 3 constraints are bound-consistent for the sb-domain. □ □

Algorithm 23 presents the filtering algorithm for the sbc-domain which relies on insights from the length-lex domain [36] and the *atmost* algorithm studied in [70]. For simplicity, it assumes the cardinality of both input variables are bounded, but it can easily be generalized to the unbounded case. It divides all elements in the universe into 9 different regions, according to how they belong in the domains. The algorithm mostly performs a case analysis of the number of empty spots in both domains. It essentially detects if the overlap region is too small (that contains only one element), in which case that element is inserted into the required set of both variables. On the other hand, if there are too few empty spots left and the variables have no fixed overlapping element, the variables cannot include elements not in the overlapping area.

Example 61. Let $X \in sbc\langle\{1\}, \{1, 2, 3, 4\}, 2, 2\rangle$ and $Y \in sbc\langle\{3, 5\}, \{3, 4, 5, 6\}, 3, 3\rangle$. There is a solution since $P_X P_Y = \{4\}$ and $P_X R_Y = \{3\}$ are both non-empty (lines 6–9). The only empty spot of X has to be used to accommodate the common element since the required element $\{1\}$ is not in the common region. As a consequence, it must require either 3 or 4 and element 2 which is not in the common region can be removed (lines 11–13).

Example 62. Let $X = \{1, 2\}$ and $Y \in sbc\langle\{3\}, \{2, 3, 4\}, 2, 2\rangle$. There is a solution since the overlapping is non-empty. Since there are only one choice in the common region, Y_{sbc} must take element 2 (lines 8–9).

Theorem 34. Algorithm 23 is sound and complete, and takes $O(n)$ time.

Algorithm 23 $bc_{sbc}\langle nonEmptyIntersection\rangle(X_{sbc} = sbc\langle R_X, P_X, c_X, c_X\rangle, Y_{sbc})$

Require: X_{sbc}, Y_{sbc} are both bound consistent

```
1:  P_X E_Y, E_X P_Y ← P_X \ (R_X ∪ P_Y), P_Y \ (R_Y ∪ P_X)
2:  P_X P_Y, R_X R_Y ← (P_X ∩ P_Y) \ (R_X ∪ R_Y), R_X ∩ R_Y
3:  R_X P_Y, P_X R_Y ← R_X ∩ (P_Y \ R_Y), (P_X \ R_X) ∩ R_Y
4:  if |R_X R_Y| > 0 then
5:     return true
6:  if |P_X P_Y| + |R_X P_Y| + |P_X R_Y| = 0 then
7:     return ⊥
8:  else if |P_X P_Y| + |R_X P_Y| + |P_X R_Y| = 1 then
9:     insert e into X_sbc, Y_sbc (where {e} = P_X ∪ P_Y)
10: else
11:    c'_X, c'_Y ← c_X − |R_X|, c_Y − |R_Y|
12:    if c'_X = 1 ∧ R_X P_Y = ∅ then
13:       exclude P_X E_Y from X_sbc
14:    if c'_Y = 1 ∧ P_X R_Y = ∅ then
15:       exclude E_X P_Y from Y_sbc
16: return true
```

Proof. Algorithm 23 assumes that both domains are bound-consistent initially. We also assume that the domains will remain the bound-consistent after any operation.[1]

The filtering algorithm reasons on the common region in which possible sets of two domains overlaps. It tries to construct a feasible assignment and determines whether an element should be included or excluded from the domain by seeking an alternative feasible assignment. Enforcing bound consistency in each domain guarantees that we are able to construct a set $x \in X_{sbc}$ and $y \in Y_{sbc}$. To enforce bound consistency, it is then sufficient to make sure that x and y overlaps.

There are 4 cases. First, the required set of both domains overlaps (lines 4–5), in which case any assignment would satisfy the intersection constraint. Second, the common region is empty (lines 6–7), meaning that there is no hope to construct a feasible assignment and the filtering algorithm fails. Third, the common region has exactly one element e (lines 8–9): This element must be inserted in the required of both domains and the filtering succeeds.

The last case occurs when the common region has more than one element and none of them are required in both sets. We consider inclusion and exclusion conditions separately. Only operations on X_{sbc} are discussed, since those on Y_{sbc} are symmetrical.

Inclusion: A free element is included in the required set only if it belongs to all solutions. We

[1]For example, given $X_{sbc} = sbc\langle\emptyset, \{1,2\}, 0, 1\rangle$, including element 1 in X_{sbc} will give $X_{sbc} = sbc\langle\{1\}, \{1\}, 1, 1\rangle$. The operation not adds only element 1 to the required set, but the cardinality lower bound and possible set are also updated accordingly to maintain bound consistency.

will show that no such element exists in this case. Consider any feasible assignment in which free element e is used only by X_{sbc}: By consistency of the domain, there is another free element e' in X_{sbc} and we can construct another feasible assignment by swapping e' with e. Consider now the case in which free element e is used by both X_{sbc} and Y_{sbc}. If e is not the only common element in the considered assignment, we can swap e with another free element in X_{sbc} and the assignment still satisfies the constraint. Otherwise, when e is the only common element, since the size of common region is greater than one, we can find another element e' in the common region. If e' belongs to Y_{sbc} in the current assignment, we can swap e and e'. Otherwise, we can swap e and e' in the current assignment to X_{sbc} and swap e' and a free element of Y_{sbc}. As a result, no free element in X_{sbc} appears in all solutiosn, which justify why lines 11–16 have no inclusion operation.

Exclusion: A free element is excluded from the possible set only if it does not belong to any solution. If there is no empty spot in X_{sbc}, there is no free element X_{sbc} and no element can be removed. If the number of empty spots $c'_{X_{sbc}}$ is at least two, one empty spot can be used by the common element and another can be used by any free element. All free elements then belong to at least one solution. As a result, it remains to consider the case where there are only one empty spot for X_{sbc}. The key idea is that, if the empty spot must be reserved for the common element, then we can remove all free elements not in the common region (lines 12–13). If $R_X P_Y$ is empty, any required element in X_{sbc} cannot be the common element, the empty spot must be reserved for the common element and any element not in the common region should be removed: these are possible elements for X_{sbc} which are not in the possible set of Y_{sbc}. Otherwise when $R_X P_Y$ is non-empty, elements in this set can served as the common element, the empty spot can be used by any free element, and hence all the free elements appear in at least one solution and cannot be removed. □ □

C.5 All Non-Empty Intersection Constraint

In SONET, a node u must share rings with all its neighbors. It naturally raises a question whether or not there exists a global constraint achieving more pruning. We define a new global constraint

$$allNonEmptyIntersect(X, \{Y_1, .., Y_n\}) \quad \equiv \quad (\forall 1 \leq i \leq n, |X \cap Y_i| \geq 1) \tag{C.21}$$

which allows us to rewrite (C.1) into

$$allNonEmptyIntersect(Y_u, \{Y_v | v \in \delta_u\}) \qquad \forall u \in N. \tag{C.22}$$

Theorem 35. $bc_{sb}\langle allNonEmptyIntersect(X, \{Y_1, .., Y_n\})\rangle$ is decomposable.

Proof. (sketch) From reference [6], $bc_{sb}(\forall i < j, |Y_i \cap Y_j| \geq 1)$ is decomposable. Our constraint is a special case of it which can be transformed to the general case by amending a dummy element to the possible set of each Y_i. □ □

Unfortunately, the result does not hold for the sbc-domain.

Theorem 36. $bc_{sbc}\langle allNonEmptyIntersect(X, \{Y_1, .., Y_n\})\rangle$ is strictly stronger than enforcing BC on its decomposition (i.e. $\forall 1 \leq i \leq n,\ bc_{sbc}\langle |X \cap Y_i| \geq 1\rangle$).

Proof. Consider $allNonEmptyIntersect(X, \{Y_1, Y_2, Y_3\})$. $X \in sbc\langle\emptyset, \{1..6\}, 2, 2\rangle$, $Y_1 \in sbc\langle\emptyset, \{1, 2\}, 1, 1\rangle$, $Y_2 \in sbc\langle\emptyset, \{3, 4\}, 1, 1\rangle$, and $Y_3 \in sbc\langle\emptyset, \{5, 6\}, 1, 1\rangle$. It is bound consistency on each constraint in the decomposition. However, there is no solution since X can only takes two elements and the possible sets of Y_1, Y_2 and Y_3 are disjoint. □ □

Theorem 37. $bc_{sbc}\langle allNonEmptyIntersect(X, \{Y_1, .., Y_n\})\rangle$ is NP-hard.

Proof. Reduction from 3-SAT. Instance: Set of n literals and m clauses over the literals such that each clause contains exactly 3 literals. Question: Is there a satisfying truth assignment for all clauses?

We construct a set-CSP with three types of variables. The first type corresponds to literals: for each literal, we construct a set variable X_i with domain $sbc\langle\emptyset, \{i, \neg i\}, 1, 1\rangle$, values in the possible set corresponds to true and false. The second type corresponds to clauses: for every clause j $(x_p \vee \neg x_q \vee x_r)$, we introduce one set variable Y_j with domain $sbc\langle\emptyset, \{p, -q, r\}, 1, 3\rangle$. The third type contains just one set variable Z correspond to the assignment, its domain is $sbc\langle\emptyset, \{1, -1, .., n, -n\}, n, n\rangle$. The constraint is in the form,

$$allNonEmptyIntersect(Z, \{X_1, .., X_n, Y_1, .., Y_m\})$$

Set variables X_i guarantees that Z is valid assignment (i.e., for every i, it can only pick either i or $-i$, but not both). Y_j and Z overlap if and only if at least one of the literals is satisfied. The constraint has a solution if and only if the 3-SAT instance is satisfiable. Therefore, enforcing bound consistency is NP-hard. □ □

C.6 Subset of Union

This section considers constraint (C.12) which is an instance of

$$subsetOfUnion(X, \{Y_1, .., Y_m\}) \equiv \bigcup_{1 \leq i \leq m} Y_i \supseteq X \tag{C.23}$$

Constraint (C.12) is justified by the following reasoning for a node u and a ring i it belongs to: If i is not used by any of u's neighbors, u does not need to use i. As a result, the rings of node u must be a subset of the rings of its neighbors. We first propose two simple inference rules to perform deductions on this constraint.

Rule 4 (SubsetOfUnion : Element Not in Union)**.**

$$\frac{i \in P_X \wedge \forall 1 \leq j \leq m, i \notin P_{Y_j}}{subsetOfUnion(X, \{Y_1, .., Y_m\}) \longmapsto i \notin X \wedge subsetOfUnion(X, \{Y_1, .., Y_m\})}$$

Theorem 38. Rule 4 is sound.

Proof. Elements in X have to be supported by some Y_i. However, as none of the Y_i contain i, this element belongs to no solution. □ □

Rule 5 (SubsetOfUnion : Element Must Be in Union)**.**

$$\frac{i \in R_X \wedge i \in P_{Y_k} \wedge |\{i \in P_{Y_j} \mid 1 \leq j \leq m\}| = 1}{subsetOfUnion(X, \{Y_1, .., Y_m\}) \longmapsto i \in Y_k \wedge subsetOfUnion(X, \{Y_1, .., Y_m\})}$$

Theorem 39. Rule 5 is sound.

Proof. Since $i \in X$ in all solutions, at least one of the variables among $Y_1, .., Y_m$ contains i. Since Y_k is the only variable that contains i, it must contain i in all solutions. □ □

Two above rules are sufficient to enforce bound consistency on the sb-domain.

Theorem 40. $bc_{sb}\langle subsetOfUnion(X, \{Y_1, .., Y_m\})\rangle$ is equivalent to enforcing rule 4 and rule 5 until they reach the fixpoint.

Proof. Consider an element $e \in P_X$. It has a support or otherwise it would be removed by rule 4. It does not belong to all solutions since, given any feasible assignment to the constraint that contains e, removing e from X still leaves us with a feasible solution. Hence e does not belong to the required set. An element $e \in P_{Y_i}$ always has a support since adding e to any feasible assignment would not make it invalid. An element $e \in P_{Y_i}$ belongs to all solutions if it must be in the union and Y_i is the only variable that contains e (rule 5). □ □

It is an open issue to determine if bound consistency can be enforced in polynomial time on the sbc-domain.

Theorem 41. $bc_{sbc}\langle subsetOfUnion(X, \{Y_1, .., Y_m\})\rangle$ is strictly stronger than enforcing rule 4 and rule 5 until they reach the fix-point.

Proof. Consider the domains $X \in sbc\langle\emptyset, \{1, .., 6\}, 0, 2\rangle$, $Y_1 \in sbc\langle\emptyset, \{1, 2\}, 1, 1\rangle$, $Y_2 \in sbc\langle\emptyset, \{3, 4\}, 1, 1\rangle$ and $Y_3 \in sbc\langle\emptyset, \{1, .., 5\}, 2, 2\rangle$. Applying the domain reduction rules, the domain of X becomes $sbc\langle\emptyset, \{1, .., 5\}, 2, 2\rangle$. $5 \in P_{Y_3}$ has no solution since X has only two empty spots, one for $\{1, 2\}$ and the other for $\{3, 4\}$ as Y_1 and Y_2 are disjoint. The constraint is thus not bound consistent. □ □

C.7 Subset Of Open Union

The SONET model contains a dual set of variables. Variable Y_u represents the set of rings node u lies on and ring variable X_i represents the set of nodes on ring i. Variable Y_u indirectly specifies the set of nodes that u can communicate with. Such set should be a superset of δ_u. We propose a global constraint that enforce this relation:

$$subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \equiv \bigcup_{i \in Y} X_i \supseteq s \tag{C.24}$$

which is used in constraint (C.13) of the model.

Example 63. Suppose node 1 has 5 neighbors (i.e., $\delta_1 = \{2, .., 6\}$), each pair has a demand of one unit. There are 2 rings, each ring can accommodate atmost 2 ADMs. There is no solution since 2 rings can accommodate atmost 4 neighbors. Using 5 *nonEmptyIntersection* constraints cannot detect such failure.

Constraint $subsetOfOpenUnion$ is sometimes called an open constraint [69], since the scope of the constraint is defined by Y. Complete filtering is polynomial for the sb-domain but intractable for the sbc-domain.

Rule 6 (SubsetOfOpenUnion : Failure)**.**

$$\frac{\bigcup_{i \in P_Y} P_{X_i} \not\supseteq s}{subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \longmapsto \bot}$$

Theorem 42. Rule 6 is sound.

Proof. The possible set is the largest set which a set variable can take. If some element in s does not belong to any possible set in the possible scope, there is no solution. □ □

Rule 7 (SubsetOfOpenUnion: Required Elements)**.**

$$\frac{i \in P_Y \wedge e \in P_{X_i} \wedge e \in s \wedge |\{e \in P_{X_j} \mid j \in P_Y\}| = 1}{\begin{array}{l} subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \\ \longmapsto i \in Y \wedge e \in X_i \wedge subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \end{array}}$$

Theorem 43. Rule 7 is sound.

Proof. Similar to Theorem 39. □ □

Theorem 44. $bc_{sb}\langle subsetOfOpenUnion(s, Y, \{X_1, .., X_m\})\rangle$ is equivalent to enforcing rule 6 and rule 7 until they reach a fixpoint.

Proof. There is no feasible assignment if and only if the union of all possible X_i is not a superset of s (rule 6). Assume that there is a feasible solution. Consider an element $e \in P_Y$ or $e \in P_{X_i}$: It

must have a support since any feasible assignment would remain feasible after adding e to it. An element $e \in P_{X_i}$ which is also in s belongs to all solutions if it belongs to exactly one variable X_i. In such case, we include e in X_i and i in Y since X_i must be in the scope (rule 7). □ □

Theorem 45. $bc_{sbc}\langle subsetOfOpenUnion(s, Y, \{X_1, .., X_m\})\rangle$ is NP-hard.

Proof. Reduction from Dominating Set. The problem of dominating set is defined as follows. Input instance: A graph $G = \langle V, E\rangle$ and an integer $k \leq |V|$. Question: Does there exist a subset V' of V such that $|V'| \leq k$ and every node in $V \setminus V'$ is a neighbor of some nodes in V'?

Given an instance with a graph G and a constant k, we construct an instance of CSP that $s = V$, $Y \in \langle \emptyset, V, 0, k\rangle$ and, for every $i \in V$, $X_i = \delta_i^G \cup \{i\}$ (where δ_i^G denotes the neighborhood of node i in graph G). Intuitively, Y corresponds to a dominating set with size at most k, X_i is a vertex that can "dominate" at most all elements in its domain (which is also the neighbors in the originally graph). The constraint is consistent if and only if there exists a dominating set of size not more than k.

$\Rightarrow$ Given a dominating set V' in the original graph G, the constraint is consistent since we can construct a solution by setting $Y = V'$, every element in Y actually corresponds to a node in the dominating set. Since every node in $V \setminus V'$ is the neighbor or at least on node in V', every element in δ_u also belongs to the domain of some X_i ($i \in Y$).

$\Leftarrow$ Given a consistent assignment of Y and X_i for all $i \in Y$, all elements in δ_u are covered by some X_i and hence Y is the dominating set. □ □

Since the constraint is intractable, we present a number of inference rules particularly useful in practice. The first inference rule reasons about the cardinality of Y. The union of X_i must be a superset of s. Since Y determines the number of X_i in the union, we can get an upper bound on the union cardinality by reasoning on the maximal cardinalities of the X_i. If the upper bound is less than $|s|$, there is no solution. Otherwise, we obtain a lower bound of cardinality of Y.

Example 64. Suppose $X_1 = X_2 = X_3 \in sbc\langle\emptyset, \{1, .., 8\}, 0, 3\rangle$, $Y \in sbc\langle\emptyset, \{1, 2, 3\}, 2, 3\rangle$ and $s = \{1, .., 8\}$. Each of X_i has 3 empty spots. We need at least $\lceil 8/3\rceil = 3$ X_i to accommodate every element in s. It implies $|Y| > 2$.

Rule 8 (SubsetOfOpenUnion : Lower Bound of $|Y|$)**.**

$$\frac{\max_{t\in d(Y):|t|=c_{\check{Y}}} \sum_{i\in t}(c_{\hat{X}_i} - |R_{X_i} \setminus s|) < |s|}{\begin{array}{l} subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \\ \longmapsto |Y| > c_{\check{Y}} \wedge subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \end{array}}$$

Theorem 46. Rule 8 is sound.

Proof. Any feasible assignment to the constraint satisfies $\bigcup_{i\in y}(x_i \cap s) \supseteq s$. Consider the set $x_i \cap s$. x_i is in $d(X_i) = sbc\langle R_{X_i}, P_{X_i}, c_{\check{X}_i}, c_{\hat{X}_i}\rangle$. We divide it into two parts: First, the elements in $R_{X_i} \cap s$ are fixed. Second, x_i can choose $c_{\hat{X}_i} - |R_{X_i}|$ elements freely from the set $P_{X_i} \setminus R_{X_i}$. The cardinality of the set $x_i \cap s$ is the sum of two parts and can be bounded from above

$$(c_{\hat{X}_i} - |R_{X_i}|) + |R_{X_i} \cap s| = c_{\hat{X}_i} - |R_{X_i} \setminus s| \geq |x_i \cap s|$$

Therefore we obtain the following inequality,

$$\sum_{i\in y}(c_{\hat{X}_i} - |R_{X_i} \setminus s|) \geq \sum_{i\in y} |x_i \cap s| \geq |\bigcup_{i\in y}(x_i \cap s)| \geq |s| \tag{C.25}$$

Cardinalities of y that do not meet this condition belong to no solution. □ □

A similar reasoning on the cardinalities of Y can remove elements of Y that corresponds to small X_i.

Example 65. Suppose $X_1 = X_2 \in sbc\langle\emptyset, \{1, .., 6\}, 0, 3\rangle$, $X_3 \in sbc\langle\emptyset, \{1, .., 6\}, 0, 2\rangle$, $Y \in sbc\langle\emptyset, \{1, 2, 3\}, 2, 2\rangle$ and $s = \{1, .., 6\}$. We need to choose two sets among X_1, X_2 and X_3. If X_3 is chosen, it provides 2 empty spots and we need 4 more spots. However, neither X_1 nor X_2 is big enough to provide 4 empty spots. It implies that Y cannot take X_3.

Rule 9 (SubsetOfOpenUnion : Pruning Elements of Y)**.**

$$\frac{\max_{t\in d(Y):i\in t} \sum_{j\in t}(c_{\hat{X}_j} - |R_{X_j} \setminus s|) < |s| \wedge i \in P_Y}{\begin{array}{l} subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \\ \longmapsto i \notin Y \wedge subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \end{array}}$$

Theorem 47. Rule 9 is sound.

Proof. Expression (C.25) gives a upper bound of empty spots that X_i can provide. If all possible values of Y containing element i do not provide enough empty spots to accommodate all elements in s, X_i is too small and $i \notin Y$. □ □

C.8 Combination of *subsetOfOpenUnion* and *channeling*

This section explores the combination of the *subsetOfOpenUnion* and channeling constraints. Indeed, in the SONET model, the X_i and Y_u are primal and dual variables channeled using the constraint: $i \in Y_u \Leftrightarrow u \in X_i$. In other words, when Y_u takes element i, one spot in X_i is used to accommodate u. Exploiting this information enables us to derive stronger inference rules.

The first inference rule assumes that Y is bound and reduces the open constraints to a global cardinality constraint. It generalizes the last two constraints (C.10) and (C.11) in Smith's extended model which apply when $1 \leq |Y_u| \leq 2$.

Definition 37 (Global lower-bounded cardinality constraint)**.** We define a specialized global cardinality constraint, where only the lower bound is specified. $GCC_{lb}(\{X_1, .., X_m\}, [l_1, .., l_n]) \equiv \forall 1 \leq j \leq n, |\{j \in X_i | 1 \leq i \leq m\}| \geq l_j$

Example 66. Suppose node 1 has 3 neighbors, $Y_1 = \{1, 2\}$. X_1 and X_2 must contain $\{1\}$ and each element in {2,3,4} has to be taken at least once. It is equivalent to $GCC_{lb}(\{X_1, X_2\}, [2, 1, 1, 1])$. By a simple counting argument, there is no solution.

Rule 10 (SubsetOfOpenUnion and Channeling : Global Cardinality)**.**

$$\frac{Y_u = y \text{ (Y_u is bounded)}}{\begin{array}{l} subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \longmapsto GCC_{lb}(\{X_i | i \in Y_u\}, [l_1, .., l_n]) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \qquad \text{where } l_u = |Y_u|,\ l_i = 1 \text{ if } i \in s \text{ and otherwise } l_i = 0 \end{array}}$$

Theorem 48. Rule 10 is sound.

Proof. When Y_u is bounded, the scope for the union is fixed. The union constraint requires that the union of set has to be a superset of s and hence each element of s has to be taken at least once.

The channeling constraint requires each variable X_i contains element u and, as Y_u defines the scope, element u has to be taken exactly $|Y_u|$ times. It reduces to a GCC_{lb}. □ □

Moreover, it is possible to strengthen the earlier cardinality-based inference rules to include the channeling information.

Example 67. Suppose $u = 1$, $s = \{2, .., 7\}$, $X_1 \in sbc\langle\{\}, \{1..8\}, 0, 3\rangle$, $X_2 \in sbc\langle\{1, 8\}, \{1..8\}, 2, 4\rangle$, $X_3 \in sbc\langle\{2, 8\}, \{1..8\}, 2, 3\rangle$, and $Y_1 \in sbc\langle\{\}, \{1, 2, 3\}, 2, 3\rangle$. Y_1 determines the scope of the union constraint. Suppose $1 \in Y_1$, X_1 is in the scope and by channeling constraint we have $X_1 \in sbc\langle\{1\}, \{1..8\}, 1, 3\rangle$. X_1 now has at most 2 empty spots for elements in s, as its cardinality upper bound is 3 and one spot is used by element 1. On the other hand, suppose $2 \in Y_1$ and hence X_2 is in the scope, it provides 2 empty spots too. Lastly, suppose $3 \in Y_1$, X_3 is in the scope and can provide 2 spots for elements in s (which includes element 2 which is already required). Therefore, each of X_i provides at most 2 empty spots to accommodate elements in s, which implies that there is no solution if $|Y_1| = 2$, we can post $|Y_1| > 2$.

Rule 11 (SubsetOfOpenUnion and Channeling : Lower Bound of $|Y|$)**.**

$$\frac{u \notin s \wedge \max_{t \in d(Y):|t|=c\check{}_Y} \sum_{i \in t} (c\hat{}_{X_i} - |R_{X_i} \setminus s| - (R_{X_i} \not\ni u)) < |s|}{\begin{array}{l} subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \longmapsto |Y| > c\check{}_Y \wedge subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \end{array}}$$

Theorem 49. Rule 11 is sound.

Proof. For any feasible assignment to the constraint $subsetOfOpenUnion(s, y_u, [x_1, ..., x_m])$, and the channeling constraint, it satisfies the condition,

$$\bigcup_{i \in y_u} (x_i \cap s) \supseteq s \wedge \bigwedge_{i \in y_u} (x_i \ni u)$$

The left part is equivalent to the proof of Theorem 46. The right part is the channeling constraint. Hence, when u is not in X_i, we need to reserve one spot for it. The condition now becomes,

$$c\hat{}_{X_i} - |R_{X_i} \setminus s| - (R_{X_i} \not\ni u) \geq |x_i \cap s|$$

$(e \notin R_{X_i})$ is a boolean function that returns 1 when the condition is true and otherwise 0. When u is not in the required set of X_i, we need to reserve one spot for it. The rest is same as the proof for Theorem 46. □ □

Rule 12 (SubsetOfOpenUnion and Channeling : Pruning Y)**.**

$$\frac{u \notin s \wedge \max_{t \in d(Y_u): i \in t} \sum_{j \in t} (c\hat{X}_j - |R_{X_j} \setminus s| - (R_{X_j} \not\ni u)) < |s| \wedge i \in P_{Y_u}}{\begin{array}{l} subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \longmapsto i \notin Y_u \wedge subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \end{array}}$$

Theorem 50. Rule 12 is sound.

Proof. The proof is similar to Theorem 49. If every possible assignment of Y that includes element i cannot satisfy the cardinality requirement, we can safely remove the element from the domain. □

□

Additional Pruning in Special Cases There are some additional inferences available when the cardinality of Y_u is 1.

Example 68. Consider the sonet problem with 5 nodes. Node 1 is adjacent with node 2,3, and 4. Node 2 is adjacent with node 1,3, and 5. Assume that the cardinality of node 2 is 1 (i.e., $|Y_2| = 1$), meaning that node 2 is on exactly one ring. This ring contains all the nodes adjacent to node 2 (i.e., 1,3, and 5). In particular, node 1 is now forced to lie on the same ring as node 5 (which is not one of its neighbors). Therefore, from the perspective of node 1, node 5 can be consider a "neighbor" and this information can result in more pruning.

Such reasoning is modeled using the following constraint which includes "new" neighbors:

$$subsetOfOpenUnion(s'_u, Y_u, \{X_1, .., X_m\})$$

where $s'_u = adj(u) \cup \bigcup_{v \in adj(u) \wedge |Y_v| = 1} (adj(v) \setminus \{u\})$. It is posted on the fly when the cardinality of neighbor node is bound to 1.

C.9 Experimental Evaluation

We now describe the experimental evaluation of our approach. We start by describing earlier results on MIP and CP models. We then present our search procedure and the computational results. Finally, we describe the impact of various factors, including the branching heuristics and the proposed global constraints.

The MIP Formulation The problem was first solved with a MIP solver [60]. The input was preprocessed before the search and some variables were pre-assigned. Valid inequalities were added during the search in order to tighten the model representation. Several variable-ordering heuristics, mainly based on the neighborhood and demand of nodes, were devised and tested. Several symmetry-breaking constraints were evaluated too. Table 1 in [59] indicates minuscule differences in performance among different symmetry-breaking constraints.

CP Formulations Smith [62] introduced a four-stage search procedure in her CP program: First decide the objective value, then decide how many rings each node lies on (label the cardinality of Y_u), then decide which rings each node lies on (label the elements of Y_u), and finally decide how much bandwidth is assigned to demand pairs on each ring. A few variable-branching heuristics were examined, with a dynamic ordering giving the best results. Symmetry-breaking techniques were also investigated. To avoid clashing with variable ordering, SBDS (symmetry breaking during search) was used. SBDS was very effective on the SONET problems, although it generated a huge number of no-good constraints, inducing a significant overhead to the system. Recall also that Smith's model included a few simple redundant constraints reasoning on the cardinality of node variables (Y_u). Please refer to Section 5 in [62] for a detailed discussion.

Another CP model was proposed in [57] and it broke symmetries by adding lexicographic constraints on set variables. With the additional lexicographic component, the solver obtained a tighter approximation of the set-variable domains. The lexicographical information was used not only for breaking symmetries, but also for cardinality reasoning. This method provided a much simpler mechanism to remove symmetries. However, as mentioned by the authors, different components of the set domain (the membership component, the cardinality restriction, and the lexicographical

bound) did not interact effectively.

The Comet Model Figures C.2, C.3 and C.4 give the model in the Comet language. The decision variables are equivalent to Smith's model, `rings[ri]` is the set of nodes assigned to ring `ri`, `nodes[ni]` is the set of rings assigned to node `ni`, `flows` denotes the amount of bandwidth of each demand allocated to each ring, `nodeCards[ni]` is an auxiliary variable indicates the cardinality of node `ni`, and `objective` is total number of ADMs used and is the value to minimize. Lines 8–15 preprocess the graph: `dem[ni]` denotes the total demand of node `ni`, `deg[ni]` its degree, and `adj[ni]` its neighborhood. The model has three parts. The first part (lines 17–34) captures the basic constraints used by all earlier CP models. The second part (lines 37–40) contains the two redundant constraints introduced in this paper. The third part (lines 43–65) is a set of implied constraints used in Smith's model.

Our Search Procedure Our CP algorithm *Boosting* implements all the constraints presented in this paper and uses a static four-stage search inspired by Smith's heuristics [62]. Figure C.4 illustrates the search procedure in COMET. Note that the searches adds redundant constraints on the fly (lines 76–81) as mentioned earlier. The algorithm first branches on the objective value (line 69), starting from the minimum value and increasing the value by one at a time from the infeasible region (line 68). The first feasible solution is thus optimal. Then the search decides the cardinality of nodes (lines 71–74). Third, the search decides the value of nodes (lines 83–89). Last, the algorithm decides the flow assigned to each pair of nodes on a ring (lines 91–95). Proposition 2 in [60] shows that there is an integral solution as long as all the demands are integral and the algorithm only needs to branch on integers. In each stage, variables are labeled in the order given by the instance being solved.

The last paragraph of Section C.8 describes some additional pruning for some special cases. It exploits the observation that, when the cardinality of a node is 1, all its neighbors must lie on the same ring. As a result, if a node's neighbor belongs to only one ring, all neighbors of the node's neighbor becomes the node's neighbor. Lines 76—81 take this into account and post redundant constraints after the cardinality of nodes is bound.

```
1 Solver<CP> cp();
2 var<CP>{set{int}} rings[Rings](cp, Nodes);
3 var<CP>{set{int}} nodes[Nodes](cp, Rings);
4 var<CP>{int} flows[Rings,Edges](cp, 0..c);
5 var<CP>{int} nodeCards[n in Nodes] = nodes[n].getCardinalityVariable();
6 var<CP>{int} objective(cp,0..r*a);
7
8 int dem[n in Nodes] = sum(ei in Edges: u[ei]==n || v[ei]==n) d[ei];
9 int deg[n in Nodes] = sum(ei in Edges: u[ei]==n || v[ei]==n) 1;
10 int g[ni in Nodes, nj in Nodes] = 0;
11 forall (ei in Edges) {
12   g[u[ei],v[ei]] = 1;
13   g[v[ei],u[ei]] = 1;
14 }
15 set{int} adj[ni in Nodes] = collect(nj in Nodes: g[ni,nj] > 0) nj;
```

Figure C.2: The Initialization for the Sonet Problem.

Benchmarks and Implementations The benchmarks include all the large capacitated instances from [60]. Small and medium instances take negligible time and are omitted. Our algorithm was evaluated on an Intel Core 2 Duo 2.6GHz laptop with 4Gb of memory. The MIP model [60] used CPLEX on a Sun Ultra 10 Workstation. Smith's algorithm [62] used ILOG Solver on one 1.7GHz processor. Hybrid[57] was run using the Eclipse constraint solver on a Pentium 4 2GHz processor, with a timeout of 3000 seconds.

Comparison of the Approaches Table C.1 reports the CPU time and number of backtracks (bt) required for each approach to prove the optimality of each instance. Our *Boosting* algorithm is, on average, more than 3400 times faster than the MIP and Hybrid approaches and visits several orders on magnitude fewer nodes. *Boosting* is more than 14 times faster than the SBDS approach when the machines are scaled and produces significantly higher speedups on the most difficult instances (e.g., instance 9). The SBDS method performs fewer backtracks in 9 out of 15 instances, because it eliminates symmetric subtrees earlier than our static symmetry-breaking constraint. However, even when the CPU speed is scaled, none of 15 instances are solved by SBDS faster than *Boosting*. This is explained by the huge number of symmetry-breaking constraints added during search. The empirical results confirm the strength of the light-weight and effective propagation algorithms proposed in this paper. While earlier attempts focused on branching heuristics and sophisticated symmetry-breaking techniques, the results demonstrate that effective filtering algorithms are key in obtaining strong performance on this problem. The remaining experimental results give empirical evidence justifying

```
solve<cp>{
  // basic constraints
  forall(r in Rings)
    cp.post(rings[r].getCardinalityVariable() <= a);
  cp.post(channeling(rings,nodes));
  forall(e in Edges)
    cp.post(atleast1(nodes[u[e]], nodes[v[e]]));
  cp.post(sum(r in Rings) rings[r].getCardinalityVariable()==objective);
  cp.post(sum(n in Nodes) nodeCards[n] == objective);
  forall(ri in Rings, rj in Rings: ri < rj)
    cp.post(lexleq(all(n in Nodes) rings[rj].getRequired(n),
                   all(n in Nodes) rings[ri].getRequired(n)));
  forall(e in Edges)
    cp.post(sum(ri in Rings) flows[ri,e] == d[e]);
  forall(r in Rings,e in Edges)
    cp.post((flows[r,e] > 0) =>
            (isRequired(rings[r],u[e]) && isRequired(rings[r],v[e])));
  forall(r in Rings)
    cp.post(sum(e in Edges) flows[r,e] <= c);

  // redundant constraints
  forall(ni in Nodes)
    cp.post(subsetOfUnion(nodes[ni],all(nj in Nodes:g[ni,nj]==1) nodes[nj]));
  forall(n in Nodes)
    cp.post(subsetOfOpenUnionWithChanneling(adj[n],nodes[n],n,rings));

  // redundant constraints (From Barbara Smith's paper)
  forall(r in Rings)
    cp.post(rings[r].getCardinalityVariable() != 1 );
  forall(n in Nodes) {
    cp.post(nodeCards[n] >= ceil((float) deg[n]/(a-1)));
    cp.post(nodeCards[n] >= ceil((float) dem[n]/c));
  }
  forall(ni in Nodes, nj in Nodes : ni < nj) {
    int c1 = sum(nk in Nodes: g[ni,nk] > 0 || g[nj,nk] > 0) 1;
    if (g[ni,nj] > 0) {
      if ((deg[ni] < a && deg[nj] < a) && c1 >= a+1)
        cp.post(nodeCards[ni] + nodeCards[nj] >= 3);
      if ((deg[ni] >= a || deg[nj] >= a) && c1 >= 2*a)
        cp.post(nodeCards[ni] + nodeCards[nj] >= 4);
    }
    else {
      if (deg[ni] < a && deg[nj] < a && c1 >= a-1)
        forall(nk in Nodes: ni!=nk && nj!=nk && g[ni,nk]>0 && g[nj,nk]>0) {
          int c2 = sum(nl in Nodes:(g[ni,nl]>0 || g[nj,nl]>0 || g[nk,nl]>0)) 1;
          if (c2 > 2*a-1)
            cp.post((nodeCards[ni]==1 && nodeCards[nj]==1) => (nodeCards[nk]>=3));
        }
    }
  }
}
```

Figure C.3: COMET Model for the Sonet Problem.

```
using {
  tryall<cp>(obj in n..r*a){
    cp.post(objective == obj);

    forall (ni in Nodes: !nodeCards[ni].bound())
      tryall<cp>(v in nodeCards[ni].getMin()..nodeCards[ni].getMax():
                    nodeCards[ni].memberOf(v))
        cp.post(nodeCards[ni] == v);

    forall (ni in Nodes) {
      set{int} s = adj[ni].copy();
      forall(u in adj[ni]: nodeCards[u].getValue() == 1)
        forall(v in adj[u]: v != ni) s.insert(v);
      cp.post(subsetOfOpenUnionWithChanneling(s,nodes[ni],ni,rings));
    }

    forall (ni in Nodes)
      forall (ri in Rings: !nodes[ni].isRequired(ri) &&
                           !nodes[ni].isExcluded(ri)) {
        try<cp>
           cp.requires(nodes[ni], ri); | cp.excludes(nodes[ni],ri);
        if (nodes[ni].bound()) break;
      }

    forall(ri in Rings, ei in Edges: !flows[ri,ei].bound() )
      while (!flows[ri,ei].bound()) {
        int l = flows[ri,ei].getMin();
        try<cp> cp.post(flows[ri,ei] == l); | cp.post(flows[ri,ei] > l);
      }
  }
}
```

Figure C.4: COMET Search Procedure for the Sonet Problem.

		MIP		Hybrid		SBDS		**Boosting**	
		Sun Ultra 10		P4, 2GHz		P(M), 1.7GHz		C2D 2.4GHz	
#	Opt	Nodes	Time	bt	Time	bt	Time	Fails	Time
1	22	5844	209.54	532065	2248.68	990	0.95	**755**	**0.09**
2	20	1654	89.23			451	0.65	**77**	**0.01**
3	22	4696	151.54	65039	227.71	**417**	0.62	781	**0.12**
4	23	50167	1814	476205	1767.82	**1419**	1.52	2585	**0.19**
5	22	36487	1358.83			**922**	0.7	1765	**0.18**
6	22	9001	343.54			**306**	0.29	519	**0.07**
7	22	13966	568.96	270310	1163.94	**982**	1.15	4395	**0.73**
8	20	441	23.38	11688	54.73	**34**	0.09	44	**0.01**
9	23	25504	701.71			35359	45.13	**5117**	**0.65**
10	24	8501	375.48			**4620**	6.75	5092	**0.84**
11	22	5015	316.77			352	0.54	**238**	**0.04**
12	22	6025	213.4			**1038**	1.09	2639	**0.31**
13	21	2052	65.06	255590	1300.91	**105**	0.14	710	**0.11**
14	23	61115	2337.29			1487	1.66	**1064**	**0.13**
15	23	100629	4324.19			13662	19.59	**1981**	**0.28**
	avg	22073.13	859.528			4142.93	5.39	**1850.80**	**0.25**

Table C.1: SONET: Experimental Results on Large Capacitated Instances.

this observation.

C.9.1 The Impact of Branching Heuristics

We now study the impact of the branching heuristics and evaluate various variable orderings for the static labeling procedure of *Boosting*. Various variable orderings were studied in [60, 62]. Most of them are based on the node demands and degrees. Our experiments considered four different heuristics: minimum-degree-first, maximum-degree-first, minimum-demand-first, and maximum-demand-first. To avoid a clash between the variable heuristics and the symmetry-breaking constraint, the lexicographic constraint uses the same static order as the branching heuristic. Table C.2 reports the average number of backtracks and time to solve all 15 instances, where row *Given* is the node ordering from the instance data. The results show that, with the exception of the *max-demand* heuristic, all variable orderings produce very similar number of backtracks and runtime performance. Moreover, the *max-demand* heuristic is still orders of magnitude faster than earlier attempts. This indicates that the variable ordering is not particularly significant when stronger filtering algorithms are available.

	avg Fails	avg Time
Given	1850.80	0.25
Min-Degree	1721.6	0.21
Max-Degree	2368.2	0.30
Min-Demand	1758.27	0.21
Max-Demand	2901.53	0.36

Table C.2: SONET: The Impact of Branching Heuristics

C.9.2 The Impact of Redundant Constraints

We conclude the experimental section by analyzing the impact of each redundant constraint. Our study simply enumerated and evaluated all combinations. The results are presented in Table C.3, where ✔ indicates that the corresponding constraint was used in the model. For cases where the sbc-domain propagator *nonEmptyIntersection* is absent, an sb-domain implementation is used instead. The table reports the average number of backtracks and the CPU time. Using all three redundant constraints (first row) gives the smallest search tree. $NotEmptyIntersection$ is the most cost-effective constraint, models using it takes the least CPU time. The models in which $subsetOfUnion$ constraint is absent (e.g. third row) achieves the same solving time as the complete model, with some more backtrackings. It suggests that constraint $subsetOfUnion$ brings the least contribution to the efficiency. Removing $subsetOfOpenUnion$ dampens the search the most, doubling the number of backtracks. Thrashing is caused when both binary intersection constraints and $subsetOfOpenUnion$ are removed (sixth row), the resulting algorithm being almost 10 times slower and visiting 11 times more nodes than the complete model. The worst performance is the last row, which essentially corresponds to Smith's model with a static symmetry-breaking constraint and a static labeling heuristic. Overall, these results suggest that, on the SONET application, the performance of the algorithm is strongly correlated to the strength of constraint propagation. The variable heuristics and the symmetry-breaking technique have marginal impact on the performance.

C.10 Conclusion

This chapter reconsiders the SONET problem. While earlier attempts focused on symmetry breaking and the design of effective search strategies, this paper took an orthogonal view and aimed at

NonEmptyIntersection $\lvert X \cap Y \rvert \geq 1$	SubsetOfUnion $\bigcup_i Y_i \supseteq X$	SubsetOfOpenUnion $\bigcup_{i \in Y} X_i \supseteq s$	avg Fails	avg Time
✔	✔	✔	1850.80	0.25
✔	✔		4569.80	0.21
✔		✔	1896.40	0.25
	✔	✔	2248.93	0.36
✔			4926.07	0.21
	✔		28956.93	1.52
		✔	2330.00	0.36
			35602.00	1.81

Table C.3: SONET: The Impact of Redundant Constraints

boosting constraint propagation by studying a variety of global constraints arising in the SONET application. From a modeling standpoint, the main contribution was to isolate two classes of redundant constraints that provide a global view to the solver. From a technical standpoint, the scientific contributions included novel hardness proofs, propagation algorithms, and filtering rules. The technical contributions were evaluated on a simple and static model that runs a few orders of magnitude faster than earlier attempts. Experimental results also demonstrated the minor impact of variable orderings, once advanced constraint propagation is used. More generally, these results indicate the significant benefits of constraint programming for this application and the value of developing effective constraint propagation over sets.

Bibliography

[1] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2009.

[2] Francisco Azevedo. Cardinal: A finite sets constraint solver. *Constraints*, 12(1):93–129, 2007.

[3] Nicolas Barnier and Pascal Brisset. Solving the kirkmans schoolgirl problem in a few seconds. In *CP-2002*, pages 477–491. Springer-Verlag, 2002.

[4] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. The parameterized complexity of global constraints. In Fox and Gomes [20], pages 235–240.

[5] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The complexity of global constraints. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 112–117, 2004.

[6] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In Wallace [71], pages 138–152.

[7] Christian Bessière and Jean-Charles Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 1999.

[8] David Cohen, editor. *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*. Springer, 2010.

[9] Charles J. Colbourn, Jeffrey H. Dinitz (Eds.), Jeffrey H. Dinitz, Leo Chouinard Ii, Robert Jajcay, and S. S. Magliveras. The crc handbook of combinatorial designs, 1995.

[10] Carlos Cotta, Iván Dotú, Antonio J. Fernández, and Pascal Van Hentenryck. Scheduling social golfers with memetic evolutionary programming. In Francisco Almeida, María J. Blesa Aguilera, Christian Blum, J. Marcos Moreno-Vega, Melquíades Pérez Pérez, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics*, volume 4030 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2006.

[11] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.

[12] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 1998.

[13] Gregoire Dooms, Luc Mercier, Pascal Van Hentenryck, Willem-Jan van Hoeve, and Laurent Michel. Length-lex open constraints. Technical Report. Brown University, 2007.

[14] Iván Dotú and Pascal Van Hentenryck. Scheduling social tournaments locally. *AI Commun.*, 20(3):151–162, 2007.

[15] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness ii: On completeness for w[1]. *Theor. Comput. Sci.*, 141(1&2):109–131, 1995.

[16] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Walsh [72], pages 93–107.

[17] P Flener, A Frisch, B Hnich, Z Kiziltan, I Miguel, J Pearson, and T Walsh. Breaking row and column symmetries in matrix models. In *in: Proceedings of the Eight International Conference on Principles and Practice of Constraint Programming*, pages 462–476. Springer-Verlag, 2002.

[18] Pierre Flener, Justin Pearson, Meinolf Sellmann, Pascal Van Hentenryck, and Magnus Ågren. Dynamic structural symmetry breaking for constraint satisfaction problems. *Constraints*, 14(4):506–538, 2009.

[19] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Walsh [72], pages 77–92.

[20] Dieter Fox and Carla P. Gomes, editors. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008.

[21] A M Frisch, B Hnich, Z Kiziltan, I Miguel, and T Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, (170):834, 2006.

[22] Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2002.

[23] G. Gange, V. Lagoon, and P.J. Stuckey. Fast set bounds propagation using a bdd-sat hybrid. *JAIR*, 2010.

[24] Graeme Gange, Peter J. Stuckey, and Vitaly Lagoon. Fast set bounds propagation using a bdd-sat hybrid. *J. Artif. Intell. Res. (JAIR)*, 38:307–338, 2010.

[25] Ian P. Gent, editor. *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732. Springer, 2009.

[26] Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In *SLP*, pages 339–358, 1994.

[27] Carmen Gervet. Conjunto: Constraint propagation over set constraints with finite set domain variables. In *ICLP*, page 733, 1994.

[28] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[29] Carmen Gervet and Pascal Van Hentenryck. Length-lex ordering for set csps. In *AAAI*. AAAI Press, 2006.

[30] Andrew Grayland, Ian Miguel, and Colva M. Roney-Dougal. Snake lex: An alternative to double lex. In Gent [25], pages 391–399.

[31] Warwick Harvey. Social Golfer Problem. csplib prob010. `http://www.csplib.org/prob/prob010/index.html`, 2011. [Online; accessed 23-Mar-2011].

[32] Warwick Harvey and Thorsten Winterer. Solving the molr and social golfers problems. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 2005.

[33] Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Set bounds and (split) set domain propagation using robdds. In Geoffrey I. Webb and Xinghuo Yu, editors, *Australian Conference on Artificial Intelligence*, volume 3339 of *Lecture Notes in Computer Science*, pages 706–717. Springer, 2004.

[34] Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using robdds. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[35] Peter Hawkins and Peter J. Stuckey. A hybrid bdd and sat finite domain constraint solver. In Pascal Van Hentenryck, editor, *PADL*, volume 3819 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2006.

[36] Pascal Van Hentenryck, Justin Yip, Carmen Gervet, and Grégoire Dooms. Bound consistency for binary length-lex set constraints. In Fox and Gomes [20], pages 375–380.

[37] Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Combining symmetry breaking with other constraints: Lexicographic ordering with sums. In *AMAI*, 2004.

[38] Brahim Hnich, Steven David Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[39] Sophie Huczynska, Paul McKay, Ian Miguel, and Peter Nightingale. Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In Gent [25], pages 50–64.

[40] Stasys Jukna. Extremal combinatorics, 2001.

[41] George Katsirelos, Nina Narodytska, and Toby Walsh. Combining symmetry breaking and global constraints, 2009.

[42] George Katsirelos, Nina Narodytska, and Toby Walsh. On the complexity and completeness of static constraints for breaking row and column symmetry. In Cohen [8], pages 305–320.

[43] Zeynep Kiziltan. Symmetry breaking ordering constraints. *Phd Thesis. Uppsala University*, 2004.

[44] Y C Law and J H M Lee. Breaking value symmetries in matrix models using channeling constraints. In *In Proceedings of the 20th Annual ACM Symposium on Applied Computing*, pages 375–380, 2005.

[45] Y. C. Law and J. H. M. Lee. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints*, 11:2006, 2006.

[46] Yat Chiu Law and Jimmy Ho-Man Lee. Global constraints for integer and set value precedence. In Wallace [71], pages 362–376.

[47] Yahia Lebbah and Olivier Lhomme. Accelerating filtering techniques for numeric csps. *Artif. Intell.*, 139(1):109–132, 2002.

[48] Yuri Malitsky, Meinolf Sellmann, and Willem Jan van Hoeve. Length-lex bounds consistency for knapsack constraints. In Peter J. Stuckey, editor, *CP*, volume 5202, pages 266–281. Springer, 2008.

[49] Kim Marriott and Peter Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[50] Luc Mercier and Pascal Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1):143–153, 2008.

[51] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artif. Intell.*, 129(1-2):133–163, 2001.

[52] Steven Prestwich. Balanced incomplete block design as satisfiability. In *In Proceedings of the 12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.

[53] J-F Puget. Pecos a high level constraint programming language. In *Proc. of Spicis*, 1992.

[54] Jean-Francois Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.

[55] Andrew Sadler and Carmen Gervet. Global reasoning on sets. In *In Proceedings of Workshop on Modelling and Problem Formulation (FORMUL01). held alongside CP-01*, 2001.

[56] Andrew Sadler and Carmen Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In Wallace [71], pages 604–618.

[57] Andrew Sadler and Carmen Gervet. Enhancing set constraint solvers with lexicographic bounds. *J. Heuristics*, 14(1):23–67, 2008.

[58] Meinolf Sellmann. On decomposing knapsack constraints for length-lex bounds consistency. In *CP'09*, pages 762–770, 2009.

[59] Hanif D. Sherali and J. Cole Smith. Improving discrete model representations via symmetry considerations. *Manage. Sci.*, 47(10):1396–1407, 2001.

[60] Hanif D. Sherali, Jonathan Cole Smith, and Youngho Lee. Enhanced model representations for an intra-ring synchronous optical network design problem allowing demand splitting. *INFORMS Journal on Computing*, 12(4):284–298, 2000.

[61] Barbara M. Smith. Reducing symmetry in a combinatorial design problem. pages 351–359, 2001.

[62] Barbara M. Smith. Symmetry and search in a network design problem. In Roman Barták and Michela Milano, editors, *CPAIOR*, volume 3524, pages 336–350. Springer, 2005.

[63] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation.* Doctoral dissertation, Saarland University, 2009.

[64] Guido Tack, Christian Schulte, and Gert Smolka. Generating propagators for finite set constraints. In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 575–589. Springer, 2006.

[65] Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals OR*, 118(1-4):73–84, 2003.

[66] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization.* The MIT Press, Cambridge, Mass., 1997.

[67] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, USA, 1989.

[68] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[69] Willem Jan van Hoeve and Jean-Charles Régin. Open constraints in a closed world. In J. Christopher Beck and Barbara M. Smith, editors, *CPAIOR*, volume 3990, pages 244–257. Springer, 2006.

[70] Willem Jan van Hoeve and Ashish Sabharwal. Filtering atmost1 on pairs of set variables. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015, pages 382–386. Springer, 2008.

[71] Mark Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258. Springer, 2004.

[72] Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001.

[73] Justin Yip and Pascal Van Hentenryck. Evaluation of length-lex set variables. In Gent [25], pages 817–832.

[74] Justin Yip and Pascal Van Hentenryck. Exponential propagation for set variables. In Cohen [8], pages 499–513.

[75] Justin Yip, Pascal Van Hentenryck, and Carmen Gervet. Boosting set constraint propagation for network design. *CPAIOR*, 2010.

Index