Reformulation of Global Constraints

Nina Narodytska

Ph.D.

2011

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation, and linguistic expression is acknowledged.’

Signed .....

Date .....

Abstract

This dissertation is dedicated to the design and analysis of algorithms for global constraints. Global constraints are one of the major contributors to the success of modern constraint solvers in solving combinatorial and industrial problems. They serve as natural building blocks in the modelling stage providing the user with a rich declarative language for problem specification. Global constraints can also lead to dramatic reduction of the search space during problem solving because their propagation algorithms leverage global knowledge about the problem.

In this work we study a number of useful global constraints, including SEQUENCE, ALL-DIFFERENT, GCC, GRAMMAR and NVALUE. We propose reformulations of these constraints into logically equivalent problems on graphs or into logically equivalent sets of primitive constraints. We analyse existing filtering algorithms for these constraints and prove that our reformulations can simulate filtering algorithms for all these constraints with a slight complexity overhead in some cases. These reformulations have a number of advantages; in particular, they allow adding global constraints to a constraint solver without implementing special-purpose filtering algorithms, they guarantee maintaining standard consistency levels for propagation, and their time complexity is similar to or the same as the best known special-purpose filtering algorithms. Finally, we investigate limitations of our approach and show that for some constraints and their filtering algorithms a polynomial size reformulation is not possible.

Acknowledgements

I am grateful to my supervisor Toby Walsh, who guided my PhD project and supported me during these years. Toby's broad research interests, experience and collaboration network that he kindly shared with me were invaluable. Toby's exceptional research vision and intuition helped to develop and improve my research skills. His hard work inspired me to work hard myself. I am indebted to Toby for helping me to master computer science research techniques, develop the ability to study problems from different angles, think critically, and much more. His energy, cheerfulness and optimism helped and encouraged me during my studies.

I would also like to thank Toby for giving me an opportunity to work on various interesting topics in computer science outside my PhD topic, including areas of social choice, knowledge compilation, parametrised complexity and dealing with symmetries in *SAT*/*CSP*, and supporting me in doing several research and industrial internships. I am also grateful for involving me in paper reviewing since the early stages of my PhD, which helped expand my outlook and research interests.

I would like to thank my co-supervisor George Katsirelos. I appreciate the time we spent working together on many interesting problems. We had fruitful and productive discussions in Sydney and Paris, leading to many interesting results. It has been a great pleasure to work with him and I learnt a lot from him. Hope we continue our collaboration in future!

I would like to thank Maurice Pagnucco, my co-supervisor, postgraduate coordinator and now Head of School of Computer Science and Engineering for his support and help. Maurice's optimism and positive attitude were helpful in all kinds of situations during my PhD!

I would like to thank all members of the COMIC group for their support, interesting discussions and for the great time together in many places and countries. I am grateful to Christian Bessiere for his continuous help, advice and collaboration on global constraints decompositions. I thank Emmanuel Herbrard and Claude-Guy Quimper for showing me the meaning of working hard and being persistent in achieving results :). I would also like to

thank Claude-Guy Quimper for collaboration on global constraints decompositions and the SEQUENCE constraint.

I would like to thank Peter Stuckey and Sebastian Brand for collaboration on decompositions of the SEQUENCE constraint. I am also grateful to Peter for valuable discussions on various topics in constraint programming.

I would like to thank Michael Maher for collaboration on flow-based algorithms for the SEQUENCE constraints and Sebastian Maneth for the collaboration on the GRAMMAR constraint.

I would like to thank Lucas Bordeaux, Youssef Hamadi and Horst Samulowitz for inviting me to do an internship with Microsoft Research Lab in Cambridge. I was impressed by the broad range of their research interests and learnt a lot from them. The nice research environment at MSR along with free food were important contributors to our results ;).

I would like to thank Angelo Oddi and Amedeo Cesta for inviting me to do an internship with the Planing and Scheduling Team in CNR,Rome. I learnt a lot about scheduling problems and internals of *SAT* solvers during this internship. I really value the time I spent working with them. I also miss real Italian food which is quite different from Italian food in Sydney.

I want to thank Jaron Schaeffer who was my supervisor during an industrial internship at Google. We made a long journey from my limited knowledge in state-of-the-art web development to the production stage of my project. I really enjoyed my time there.

I would like to thank Mike Fellows for inviting me to visit Newcastle University and work on parametrised complexity problems for global constraints. My learning curve was very steep during this visit and I am grateful for the work we did together.

I would like to thank Laurent Simon and George Katsirelos for inviting me to visit LRI Lab in Paris and work on *SAT* encodings of bin-packing problems. I hope we will finish this work in near future!

I want to thank my CP mentors Gilles Pesant, Nicolas Beldiceanu, and Meinolf Sellmann for valuable discussions during the mentoring program at CP conferences.

I would like thank Jessica Davies and Lirong Xia who visited Sydney Lab for the interesting collaboration on coalition manipulation problems.

I want to thank all my colleagues at constraint programming groups across NICTA and CSE, UNSW whose knowledge and sense of humour have helped reduce the inevitable stress of being a graduate student.

I want to thank Michael Maher and Sebastian Brand for proofreading the thesis.

Finally, and most importantly, I thank my parents for looking after my animals, an American cocker-spaniel Eva and a blue budgie parrot Flint, who were in Ukraine during all these years.

x

Dedication

To Eva and Flint

Related publications

Journal papers

1. Lucas Bordeaux, George Katsirelos, Nina Narodytska, and Moshe Y. Vardi. The complexity of integer bound propagation. In *Journal of Artificial Intelligence Research*, number 40, pages 657–676, 2011.

2. George Katsirelos, Nina Narodytska, and Toby Walsh. The weighted GRAMMAR constraint. *Annals of Operations Research*, 184:179–207, 2011.

Conference papers

1. Michael Fellows, Tobias Friedrich, Danny Hermelin, Nina Narodytska, and Frances Rosamond. Constraint satisfaction problems: Convexity makes ALL-DIFFERENT constraints tractable. In *Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI'11)*, 2011.

2. Christian Bessiere, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. The ALL-DIFFERENT constraint with precedences. In *Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisations Problems (CPAIOR'11)*, 2011.

3. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decomposition of the NVALUE constraint. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*, volume 6308, pages 114–128. Springer, 2010.

4. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Propagating conjunctions of ALL-DIFFERENT constraints. In Maria Fox and David Poole, editors, *Proceedings of the 24th National Conference on Artificial intelligence (AAAI'10)*. AAAI Press, 2010.

5. George Katsirelos, Sebastian Maneth, Nina Narodytska, and Toby Walsh. Restricted global GRAMMAR constraints. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, volume 5732, pages 501–508. Springer, 2009.

6. George Katsirelos, Nina Narodytska, and Toby Walsh. Reformulating global GRAMMAR constraints. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR'09)*, pages 132–147, 2009.

7. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of ALL-DIFFERENT , Global Cardinality and related constraints. In *Proceedings of the 21th International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 419–424, 2009.

8. Christian Bessiere, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In *Proceedings of the 21th International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 412–418, 2009.

9. Michael Maher, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Flow-based propagators for the SEQUENCE and related global constraints. In Peter J. Stuckey, editor, *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08)*, pages 159–174. Springer, 2008.

10. George Katsirelos, Nina Narodytska, and Toby Walsh. The weighted GRAMMAR constraint. In *Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR'08)*, volume 5015, pages 323–327. Springer, 2008.

11. Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter Stuckey, and Toby Walsh. Encodings of the SEQUENCE constraint. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 210–224.

Other Publications

1. Michael Maher, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Flow-based propagators for the SEQUENCE and related global constraints (an extended

version of the CP'08 paper). In *ECAI Workshop on Modeling and Solving Problems with Constraints*, pages 54–62, Patra, Greece, 2008.

2. Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter Stuckey, and Toby Walsh. Encodings of the SEQUENCE constraint (an extended version of the CP'07 paper). Technical Report 010, 2007.

3. Nina Narodytska and Toby Walsh. The cyclic SEQUENCE constraint. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS'07), Doctoral Programme*, 2007.

4. Nina Narodytska and Toby Walsh. Encodings of the SEQUENCE constraint using the REGULAR constraint. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, Doctoral Programme*, 2007.

The work in the thesis was carried out in collaboration with a number of people. In some cases it is hard to separate my own results and ideas from the work of other people during the iterative process of trial and error in achieving these results.

I hereby declare a *significant contribution into theoretical results and ideas and all experimental results*.

Chapter 3. The results in this chapter were obtained in collaboration with Christian Bessiere, George Katsirelos and Toby Walsh. Sections 3.1– 3.3 are my own work.

Chapter 4. The results on decompositions of the SEQUENCE constraint were obtained in collaboration with Sebastian Brand, Claude-Guy Quimper, Peter Stuckey and Toby Walsh. Section 4.3.3 on the decomposition that is based on the REGULAR constraint, Section 4.3.6 on theoretical results on comparison of decompositions and Section 4.6 on experimental results are my own work.

The results on flow-based algorithms for the SEQUENCE constraint were obtained in collaboration with Micheal Maher, Claude-Guy Quimper and Toby Walsh. The idea of reformulation of the SEQUENCE and soft SEQUENCE constraints as integer linear programs with a totally unimodular matrix (Section 4.2.3 and Section 4.4.2), the hardness proof of the Multiple SEQUENCE constraint (Section 4.4.5) and experimental results (Section 4.6) are my own work.

Chapter 5. The results on reformulation of the GRAMMAR constraint into the REGULAR constraint were obtained in collaboration with George Katsirelos and Toby Walsh. The

idea of reformulation steps (Section 5.3.1–Section 5.3.4) is mine and experimental results (Section 5.3.7) are my own work.

The results on restrictions the context free grammars were obtained in collaboration with George Katsirelos, Sebastian Maneth and Toby Walsh. The impossibility result for restricted GRAMMAR constraints in Section 5.4.1 is my own work.

The results on weighted GRAMMAR constraint were obtained in collaboration with George Katsirelos and Toby Walsh. The domain consistency propagator for the weighted GRAMMAR constraint and experimental results (Section 5.5.6) are my own work.

Chapter 6. The results on reformulation of ALL-DIFFERENT and its generalizations were obtained in collaboration with Christian Bessiere, George Katsirelos, Claude-Guy Quimper and Toby Walsh. The partial sums decomposition of the ALL-DIFFERENT constraint (Section 6.4.3), theoretical results on the OVERLAPPINGALLDIFF constraint (Section 6.7.1–Section 6.7.2) and experimental results in Section 6.5 and 6.7.4 are my own work. These results correct an error in the 'Propagating conjunctions of ALL-DIFFERENT constraints'. I also reworked and simplified the proof of bounds consistency of the GCC constraint (Section 6.8.2). An extension of the Hall theorem to maximum matching (Section 6.9.1.3), the proof for bounds consistency decomposition of the ATLEASTNVALUE constraint (Section 6.9.2.3, which is different from an incomplete proof in the 'Decomposition of the NVALUE constraint' paper) and experimental results (Section 6.9.5) are my own work.

Chapter 7. The results on limitations of decompositions were obtained in collaboration with Christian Bessiere, George Katsirelos and Toby Walsh.

Contents

List of Figures

Acronyms

APSP	all pairs shortest path
BC	bounds consistency
CFG	context-free grammar
CNF	conjunctive normal form
CSP	constraint satisfaction problem
CYK	Cocke-Younger-Kasami
DAG	directed acyclic graph
DC	domain consistency
DFA	deterministic finite automaton
FLT	failed literal test
ILP	integer linear programming
LP	integer linear programming
NFA	non-deterministic finite automaton
PB	pseudo-Boolean
PDA	pushdown automaton
RC	range consistency
SAT	satisfiability formula
SBC	singleton bounds consistent
SCC	strongly connected component

SDC singleton domain consistent

SSSP single source shortest path

Chapter 1

Introduction

This thesis studies the design and analysis of algorithms for solving combinatorial optimisation problems. Such problems arise from the need to improve day-to-day operation of companies across all sectors of the economy. The ability to solve them efficiently is often critical to maintaining the competitiveness of services and goods produced by the company.

Consider, for instance, the process of producing of a hardware device. At the design stage, among other problems, one might need to find an optimum placement of components on the circuit board and design automated testing procedures to find possible device defects. At the manufacturing stage, we might have to create a shift schedule for a factory to produce devices and an optimum ordering of partially completed devices on the assembly line. Finally, when shipping the product to customers, one needs to solve vehicle routing problems with additional constraints.

Fast technological progress has lead to a dramatic growth in the size of industrial optimisation problems. Nowadays, many real-world problems are very difficult for a human to tackle. Therefore, computing resources and efficient algorithms play an important role in solving such problems.

The availability and affordability of computing resources increases every year. This trend is driven by hardware improvements, such as growing transistor densities and various architectural optimisations, as well as new infrastructure technologies, such as cloud computing. On the algorithmic side, the need to solve industrial optimisation problems fosters the development of software combinatorial optimisation toolkits [Bix11].

Some of the first widely used toolkits came from the operations research community. The development of the theory of *integer linear programming* (*ILP*) and the availability of efficient implementations of *ILP* algorithms made it possible to deal with large-scale problems in various domains, such as network design, shift scheduling, vehicle routing, and

graph drawing [KB85, RT08, JM96, EFK00]. To solve a problem, one has to express it as a system of linear inequality constraints and give this formal model to an *ILP* solver. The solver uses branch-and-bound techniques to find an optimal solution. From the theoretical perspective, the class of problems that *ILP* tackles is the class of NP-complete problems.

Another popular class of software tools for solving NP-complete problems are *Boolean satisfiability solvers* (SAT). A complete *SAT* solver performs exhaustive exploration of the search space using smart heuristics and learning of new implied information during the search. *SAT* solvers are used in software and hardware verification by Intel, IBM, and many other companies [KS03, AFF+05, Zar04, Zar05]. To use most *SAT* solvers, one needs to express the problems as a conjunction of disjunctions of Boolean variables. Some *SAT* solvers also allow an arbitrary Boolean formula as an input format.

Constraint solvers are a generalisation of Boolean solvers. They allow specifying the problem as a set of high-level constraints, including so-called global constraints, over finite domain variables. A constraint solver consists of two main parts: a modelling layer and a search engine. One aim of the constraint programming modelling layer is to provide the user with a set of global constraints to make the modelling process as simple as possible.

The richness of the constraint programming modelling layer contrasts with the very restricted constraint languages that are supported by *SAT* and *ILP* solvers. These toolkits use disjunctions of literals and linear inequality constraints, respectively, to represent a problem. Useful global constraints are found by identifying patterns that commonly occur in many problems. For example, finite state machines are often used to specify regulation rules in scheduling and rostering problems. This motivated the introduction of the REGULAR global constraint that allows modelling problem constraints in the form of an automaton [Pes04, BCP04]. Another common pattern occurs when a set of objects must be matched to another set of pairwise distinct objects. For instance, a set of machines have to perform different tasks at any time point or a student cannot take two exams on the same day. To model this pattern the ALL-DIFFERENT constraint was introduced [Reg94]. More useful constraint patterns are continually being identified, and so far about 300 are formally described in the global constraints catalogue [BCR05] that form a powerful modelling layer that allows a non-expert to specify their problem.

Besides being easy to use, the constraint programming modelling layer has the important advantage of preserving the structure of the problem. A constraint solver can leverage this structure-preserving encoding to solve the problem more efficiently. Since the structure of the model significantly affects the performance of the constraint solver, the user needs

some expertise in modelling their problem in the most efficient way by choosing the right constraints among hundreds of available global constraints.

Given a model of the problem a constraint solver search engine performs an exhaustive exploration of the search space trying to avoid going into unsatisfiable subtrees by means of inference algorithms, called *filtering algorithms* or *propagators* [1]. Filtering algorithms work locally for each individual constraint. Therefore, it is important to provide the best possible algorithm for each constraint to achieve good performance. Significant research effort is devoted to developing such algorithms for many useful global constraint [HK06].

The growing importance of combinatorial optimisation problems has motivated the development of numerous industrial ILP, Boolean, and constraint solvers in the last two decades. To name a few examples, Microsoft Research developed the Microsoft Solver Foundation, which includes all three techniques, and incorporated in the Microsoft Excel product [Mic11]. Intel uses a home-grown constraint solver for automatic test generation for hardware verification [Gut08, HCG09]. IBM is developing several constraint programming solvers, including complete and incomplete-search constraint solvers, and recently acquired the ILOG product line, which includes *ILP* and constraint solvers [IBM11]. Google recently developed a constraint solver for vehicle routing problems [Goo11]. Continued demand for combinatorial optimisation tools motivates the research community to create new efficient and effective algorithms to be incorporated into these tools.

This thesis addresses one of the important challenges in constraint programming, namely *constraint reformulation* [Fre97]. Constraint reformulation is a technique that allows us to substitute a constraint with a set of simpler constraints and preserve logical equivalence between the original constraint and the constraints in the reformulation. These simpler constraints can be primitive, bounded arity constraints or more expressive constraints on graphs, like network flow or shortest path constraint. On top of preserving logical equivalence, a reformulation can satisfy additional requirements, e.g. guarantee that a given amount of inference is achieved by constraints in the reformulation. Such reformulations can have a number of advantages. For example, they allow adding new global constraints to a constraint solver without implementing special-purpose filtering algorithms for these constraints. Another advantage is that reformulations can expose the internal state of filtering algorithms to the constraint solver. This, potentially, allows us to build better branching heuristics and obtain learning schemes, such as the c-learning scheme that was recently proposed by Moore [Moo11]. In addition, we can use reformulation to communicate in-

[1] We only consider complete search *CSP* solvers in this work.

formation between different constraints through the extra variables that are introduced in the reformulation process. In this work, we focus on reformulations of filtering algorithms for many important global constraints, which ensure that constraints in the reformulation achieve the same inference as the filtering algorithm for the original constraint would do.

The thesis defended in this dissertation is that:

> *Efficient propagators for many important global constraints can be developed by reformulating them using decompositions based on linear inequalities and network flows. However, we show that there are also theoretical limits on the efficiency of propagators and decompositions that can simulate them.*

Outline of the thesis. The rest of this thesis is structured as follows. Chapter 2 gives an overview of results from graph theory, integer linear programming, formal languages and constraint programming used in the rest of the thesis. Chapter 3 gives an overview of the theoretical framework of constraint decompositions introduced in the previous work and extends it to handle decompositions of constraint propagators. Chapter 4 proposes network-flow based filtering algorithms for the SEQUENCE constraint and its generalisations. It also proposes a number to decompositions of the filtering algorithm into a set of bounded arity constraints. Chapter 5 is dedicated to the GRAMMAR constraint and its generalisations. It investigates a wide class of restricted context free grammars and establishes a lower bound on the complexity of filtering algorithms for these grammars. It also investigates reformulations from GRAMMAR to REGULAR constraint and proposes a filtering algorithm and a decomposition for the weighted GRAMMAR constraint. Chapter 6 studies the ALL-DIFFERENT constraint and its generalisations and proposes a range of decompositions into bounded arity constraints. Chapter 7 explores theoretical limits on the efficiency of constraint propagator decompositions. In particular, we show that the polynomial time domain consistency filtering algorithm for the ALL-DIFFERENT constraint cannot be encoded into a polynomial size *SAT* formula so that unit propagation in a SAT solver achieves the same amount of inference. Chapter 8 summarises the main contributions of the thesis and outlines future research directions.

Chapter 2

Background

In this chapter we introduce the formal background for the work we present in Chapters 3–7.

2.1 Graph Theory

In this section we recall several important problems in graph theory, like the shortest path and maximum network flow problems. These problems are central problems in graph theory, as many real world problems, e.g the distribution of the traffic in a road network or assignment of jobs to machines in production line scheduling, can be reduced to them. In constraint programming these problems also play an important role. Constraint propagators for many useful global constraints can be reduced to problems on graphs.

2.1.1 Basic definitions

The definitions in this section are based on [Sch86]. An *undirected graph* $G = (V, E)$ consists of two sets: a set of *vertices* (or *nodes*) $V = \{v_i | i = 1, \ldots, n\}$ and a set of *edges* $E = \{e_i | e_i = (v_i, v_j), v_i, v_j \in V\}$. Each edge $e_i = (v_i, v_j)$ represents an *unordered pair* of two vertices v_i and v_j that it connects. An example of a graph is shown in Figure 2.1. The graph contains 6 vertices $v_1, \ldots, v_6$ and 6 edges. If a vertex v_i is an endpoint of an edge e, then e is said to be *incident* on v_i. A vertex u is *adjacent* to a vertex v if they are joined by an edge. Two adjacent vertices are called *neighbours* . For example, vertices v_1 and v_4 are adjacent, hence, they are neighbours. No graph that we use in this thesis contains loops, i.e. an edge with a single endpoint, and multiple edges. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A *path* in a graph $G = (V, E)$ from v_0 to v_n is a sequence of distinct vertices and edges of the form $(v_0, e_0, v_1, \ldots, v_{n-1}, e_{n-1}, v_n)$, such that $e_i = (v_i, v_{i+1})$, $i = 0, \ldots, n-1$.

Figure 2.1 A graph with 6 vertices and 6 edges.

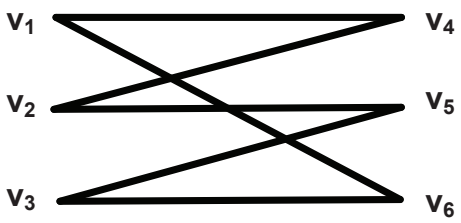
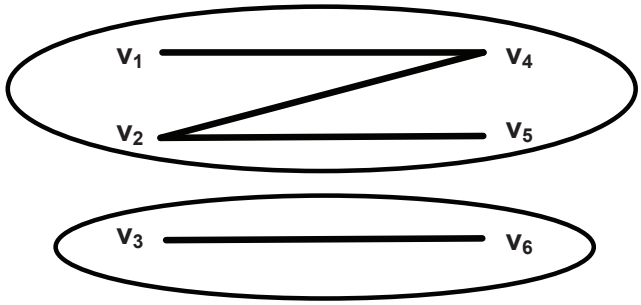


Figure 2.2 A graph with two connected components.



For example, the sequence $(v_3, (v_3, v_5), v_5, (v_5, v_2), v_2)$ is a path from the vertex v_3 to the vertex v_2 in Figure 2.1. We can unambiguously describe a path using only the sequence of vertices $(v_0, \ldots, v_n)$, e.g. (v_3, v_5, v_2), or using only the sequence of edges $(e_0, \ldots, e_{n-1})$, e.g. $((v_3, v_5), (v_5, v_2))$. If $v_0 = v_n$, the path is called *closed*. A closed path of length at least one is called a *circuit*. A graph is *connected* if there exists a path between any two vertices. A set of vertices connected by a path is called a *connected component* of the graph. A graph can have several connected components. An example of a graph with two connected components is at Figure 2.2. A graph without circuits is called a *forest*. A connected forest is called a *tree*.

A *directed graph* $G = (V, E)$ consists of a set vertices V and a set of ordered pairs E. The elements of $E = \{e_k \mid e_k = (v_i, v_j), v_i, v_j \in V\}$ are called *directed edges*. Each directed edge e_{ij} represents an *ordered pair* of two vertices, a head vertex v_i and the tail vertex v_j, that it joins. For simplicity, we use the same notations for a set of directed and undirected edges. The type of graph will be clear from the context. Figure 2.3 shows an example of a directed graph.

A *directed path* in a graph $G = (V, E)$ from v_0 to v_n is a sequence of vertices and directed edges of the form $(v_0, e_0, v_1, \ldots, v_{n-1}, e_{(n-1)}, v_n)$ such that $e_i = (v_i, v_{i+1})$, $i = 0, \ldots, n-1$. For example, $(s, (s, v_3), v_3, (v_3, v_6), v_6)$ is a path from s to v_6 in Figure 2.3. We can unambiguously describe a path using only the sequence of vertices $(v_0, \ldots, v_n)$ or using only the sequence of edges $e_0, \ldots, e_{(n-1)}$. If $v_0 = v_n$, the directed path is called

Figure 2.3 A network flow graph.

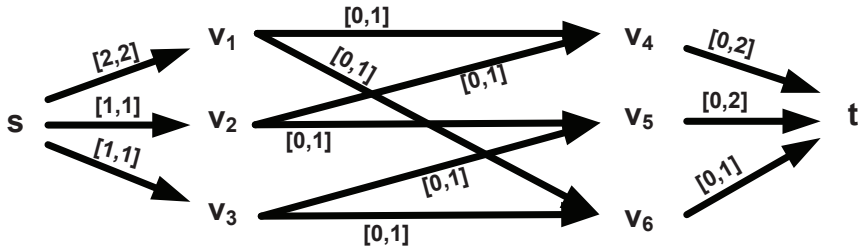
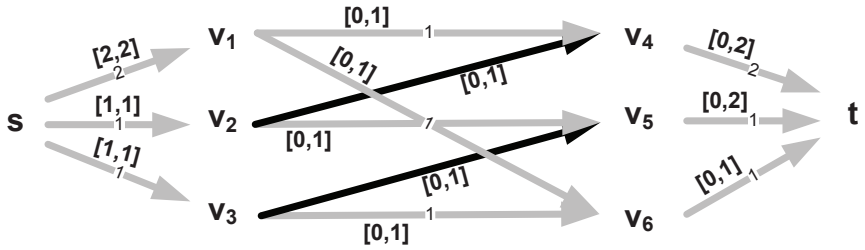


Figure 2.4 A network flow graph.



closed. A closed directed path of length at least one is called a *directed circuit* or *cycle*.

A directed graph $G = (V, E)$ is *strongly connected* if there exists a directed path between any two of its vertices. A maximal strongly connected subgraph is called a *strongly connected component* or SCC.

A *network flow graph* is a directed graph $G = (V, E)$ where each edge has limited capacity. The *capacity* of an edge e_i is an interval of values, $[l(e_i), u(e_i)]$. We distinguish two special types of vertices: *sources* and *sinks*. The source vertex s does not have incoming edges and a sink vertex t does not have outgoing edges. An example of a network flow graph is shown in Figure 2.3. For instance, the capacity of the edge from v_4 to the sink t is $[0, 2]$.

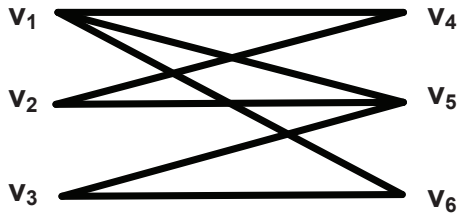
A *flow network* is a function that maps edges to integers $f : E \longmapsto N$ and satisfies the *flow conservation law* for all vertices except the source and the sink. We denote $f(e)$ the amount of flow that is going through an edge. The conservation law requires the amount of flow that enters each vertex v_i to be equal to the amount flow that leaves this vertex:

$$\sum_{e_{ji} \in E, j \neq i} f\left(e_{ji}\right) = \sum_{e_{ij} \in E, j \neq i} f\left(e_{ij}\right)$$

We denote by $v(f)$ the amount of the flow that comes out of the source node. Due to the conservation law the same amount of flow goes into the sink node. We call $v(f)$ the *flow value*. Figure 2.4 shows a flow in a network graph in grey colour. The value of this flow is 4.

A *weighted* undirected or directed graph $G = (V, E)$ is a graph where every edge has

Figure 2.5 A convex bipartite graph with 6 vertices and 6 edges.



a real weight attached to it. We denote $w(e_i)$ the *weight* or *cost* of an edge. The *weight* or *cost* of the path between vertices v_0 and v_n is the sum of the weights of all edges on this path. Cost of the flow through an edge is the product of the amount of flow, $f(e)$, and the weight of this edge. The *weight* or *cost* of the flow in a network graph, $w(f)$, is the sum of the flow costs of all edges that a flow goes through.

2.1.2 Restricted graphs

We consider several special cases of graphs that play an important role in constructing constraint propagators for global constraints, such as bipartite graphs, convex bipartite graphs and interval graphs.

Definition 2.1 *The graph $G = (V, E)$ is* bipartite *if V partitions into 2 classes, $V = A \cup B$ and $A \cap B = \emptyset$, such that every edge has ends in different classes.*

The graph presented at Figure 2.1 is a bipartite graph. Vertices of this graph can be partitioned into two sets $A = \{v_1, v_2, v_3\}$ and $B = \{v_4, v_5, v_6\}$ so that each edge has its endpoints in two different partitions.

Definition 2.2 *The bipartite graph $G = (V, E)$, $V = A \cup B$ is* convex bipartite *if vertices in the partition B can be ordered such that for all $v_i \in A$ the vertices adjacent to v_i are consecutive in this order.*

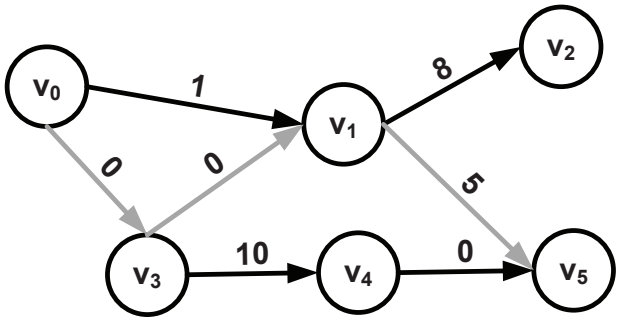
The graph presented at Figure 2.1 is not a convex bipartite graph. We can check that any ordering on the vertices v_4, v_5 and v_6 does not create a convex graph. Figure 2.5 shows an example of a convex bipartite graph.

Definition 2.3 *The graph $G = (V, E)$ is an* interval *graph if it could be interpreted as the intersection of a set of intervals on the real line. It has one vertex for each interval in the set, and an edge between every pair of vertices corresponding to intervals that intersect.*

Figure 2.6 shows an example of a set of three intervals and the corresponding interval graph.

Figure 2.6 An interval graph.

Figure 2.7 A weighted directed graph.



2.1.3 Problems on graphs

In this section we introduce several important problems in graph theory. We use these problems in Chapters 4 and 6, where we construct reformulations for SEQUENCE and ALL-DIFFERENT constraints and their generalisations.

Problem 2.1 (Shortest path) *Let $G = (V, E)$ be a weighted graph. The* shortest path *between two vertices s and t is a path of the minimum cost between s and t.*

Figure 2.7 shows a weighted directed graph. The shortest path between vertices v_0 and v_5 is highlighted in gray. The Bellman-Ford algorithm solves the single source shortest path problem (*SSSP*), which is a generalisation of the shortest path problem. *SSSP* finds shortest paths from a given source vertex to all vertices in the graph. The Bellman-Ford algorithm takes $O(|V||E|)$ time. In a weighted graph that does not contain negative cost edges solving *SSSP* takes $O(|V|^2)$ using Dijkstra's algorithm. Using efficient data structures the time complexity can be improved [CLRS01] for sparse graphs to $O(|E| \lg |V|)$ using binary min-heap or to $O(|V| \lg |V| + |E|)$ using Fibonacci heap implementations of the min-priority queue. Finally, if a graph does not contain cycles, so called directed acyclic graph (*DAG*), then solving *SSSP* takes $O(|E|)$ time [CLRS01].

Problem 2.2 (All pairs shortest path (*APSP*)) *Let $G = (V, E)$ be a weighted graph. The*

all pairs shortest path problem *is the problem of finding a shortest path between every pair of vertices in* G.

APSP can be solved in $O(|V|^2|E|)$ if we run the Bellman-Ford algorithm for each vertex. If all edges have non-negative cost, we can use Dijkstra's algorithm which reduces complexity to $O(|V|^3)$. Alternatively, we can use the Floyd-Warshall algorithm, which runs in $\Theta(|V|^3)$ time and works for graphs with negative cost edges. For sparse graphs, it is better to use Johnson's algorithm that runs in $O(|V|^2 \lg |V| + |V||E|)$ time [CLRS01].

Problem 2.3 (Maximum matching) *Let $G = (V, E)$ be an undirected graph. A matching is a set of edges $M \subseteq E$ such that every vertex is incident to at most one edge in the set. The* maximum matching *problem is the problem of finding a matching of the maximum size.*

Figure 2.8(a) shows a maximum matching in a graph. Edges in the matching are highlighted in gray color. In this thesis we only work with maximum matching in bipartite graphs. The Hopcroft-Karp algorithm finds a maximum matching in a bipartite graph in $O(|E|\sqrt{|V|})$ time.

One generalisation of the maximum matching problem in unweighted graphs is the problem of finding a minimum cost maximum matching on weighted graphs. The *cost* or *weight* of the matching is the sum of the weights of all edges in the matching.

Problem 2.4 (Minimum cost maximum matching) *Let $G = (V, E)$ be an undirected weighted graph. A* minimum cost maximum matching *is a maximum matching of the minimum cost.*

Figure 2.8(b) shows a minimum cost maximum matching in a graph. Edges in the minimum cost maximum matching are highlighted in gray. In this thesis we only consider the minimum cost maximum matching in bipartite graphs. This problem can be seen as a special case of the minimum cost maximum flow problem that we consider below.

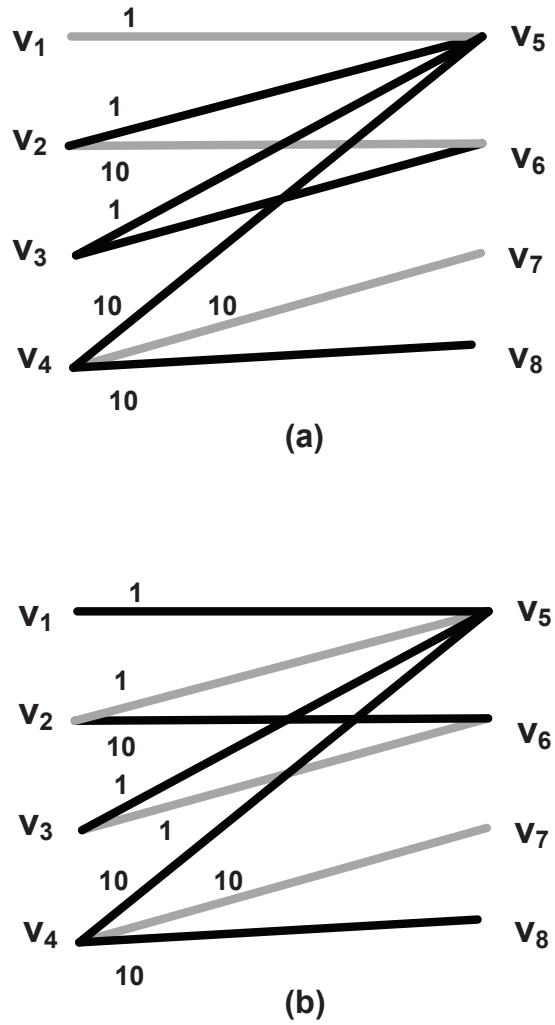
Another generalisation of the matching problem is the network flow problem. In the next section we consider this problem and its generalisation in details as constraint propagators that we propose in this thesis rely on the algorithms for network flow problems.

Problem 2.5 (Maximum flow) *Let $G = (V, E)$ be a directed network flow graph. The* feasible network flow *problem is to identify a flow f in a network $G = (V, E)$ satisfying the following constraints $\forall e_i \in E$ and $\forall v_i \in V \setminus \{s, t\}$:*

$$\sum_{e=(v_j,v_i)\in E, j\neq i} f(e) = \sum_{e=(v_i,v_j)\in E, j\neq i} f(e) \tag{2.1}$$

$$l(e) \leq f(e) \leq u(e) \tag{2.2}$$

Figure 2.8 (a)A maximum matching in a graph; (b) a minimum cost maximum matching.



The maximum network flow *is a feasible flow with the maximum flow value* $v(f)$.

Figure 2.4 shows a network graph with a flow of value 4. The maximum flow can be solved by the successive shortest path algorithm in $O(|V|^2|E|)$ time. There exist more efficient preflow-push algorithms that run in $O(|V|^2\sqrt{|E|})$ and $O(|V||E| + |V|^2 \lg |v(f)|)$ time, where $v(f)$ is the value of the maximum flow.

We introduce the optimisation version of the maximum flow problem, which is a generalisation of Problems 2.1–2.5.

Problem 2.6 (Minimum cost maximum flow) *Let* $G = (V, E)$ *be a directed network flow graph. The* minimum cost maximum network flow *is a maximum network flow with the minimum flow weight* $w(f)$.

Problem 2.6 is one of the most researched problems in computer science. This problem can be solved in polynomial time using a number of efficient algorithms [AMO93]. We

consider one of them in the next section that runs in $O(v(f))O(SP)$ time, where $O(SP)$ is the time complexity of a shortest path algorithm.

Finally, we introduce several NP-complete problems on general graphs. We point out that these problems can be solved in polynomial time if we consider restricted graphs, such as interval graphs or bipartite graphs.

Problem 2.7 (Maximum independent set) *Let $G = (V, E)$ be an undirected graph. An* independent set *is a set of vertices $V' \subseteq V$ such that for any two vertices $v_i, v_j \in V'$, v_i and v_j are not adjacent. A* maximum independent set *is an independent set of the maximum size. We denote $\alpha_I(G)$ the size of the maximum independent set.*

For example, the vertices v_1, v_2, v_3, v_7 and v_8 in Figure 2.8(a) form a maximum independent set.

Problem 2.8 (Maximum clique) *Let $G = (V, E)$ be an undirected graph. A* clique *is a set of vertices $V' \subseteq V$ such that for any two vertices $v_i, v_j \in V'$, v_i and v_j are adjacent. A* maximum clique *is a clique of the maximum size.*

We call a *maximal clique* a clique in a graph that is not contained in any other clique.

2.1.4 Network flow theory

In this section we consider the successive shortest path algorithm to find a minimum cost maximum flow in a graph [Sch86]. The algorithm uses the notion of a residual network.

Definition 2.4 *Given a flow network $G = (V, E)$ and a flow f, the* residual network graph *of G induced by f is $G_f = (V, E_f)$, where for each edge $e = (v_i, v_j) \in E$ there exists*

- *an edge $e = (v_i, v_j) \in E_f$ if $f(e) < u(e)$ with capacities $[max(l(e) - f(e), 0), u(e) - f(e)]$ and the weight $w(e)$,*
- *an edge $e^{-1} = (v_j, v_i) \in E_f$ if $f(e) > l(e)$, with capacities $[0, max(f(e) - l(e), 0)]$ and the weight $-w(e)$,*

where e^{-1} denotes a reverse edge of e.

The residual graph shows how much extra flow we can push through an edge $e = (v_i, v_j)$ and how much flow we can remove from this edge while meeting the lower bound requirements for the flow through an edge. We also extend a set of edges in the residual graph with an edge from the sink t to the source s of the capacity $[0, \infty]$ and zero cost. On each iteration, the successive shortest path algorithm finds an edge $e = (v_i, v_j)$ such that

$f(e) < l(e)$ and finds a path P from v_j to v_i in the residual graph. If there is no such path then there is no feasible flow. Otherwise, it maximally increases the flow along the cycle formed by the edge (v_i, v_j) and the path P taking into account capacities of all edges along this path. We define a characteristic vector of a path P, s^P, as follows: $s^P(e) = 1$ if P contains e, $s^P(e) = -1$ if P traverses e^{-1} and $s^P(e) = 0$ in all other cases. In a similar way, we can define the characteristic vector of a cycle. Algorithm 2.1 shows a pseudocode for the successive shortest path algorithm [Hoe05].

Algorithm 2.1 Minimum-weight flow of value k in a graph $G = (V, E)$

procedure MINCOSTMAXFLOW(G, k)
  $f(e) = 0, e \in E$
  if $\exists e \in E$ s.t. $l(e) > 0$ **then**
    build a residual graph G_f extended with an edge (t, s) with capacities $[0, \infty]$ and a zero weight edge.
    while $\exists e = (v_i, v_j) \in E$ s.t. $f(e) < u(e)$ **do**
      compute a directed path from v_j to v_i with the minimum cost $w(P)$
      if P does not exist **then**
        return Failure.
      else
        define a cycle with P and the edge (v_i, v_j, s).
      reset $f = f + \varepsilon s^C$ where ε is maximal subject to $f + \varepsilon s^P \leq u$.
  if $v(f) > k$ **then**
    return True.
  else
    while $v(f) < k$ **do**
      compute a directed path from s to t in G_f minimising $w(P)$
      if P does not exist **then**
        return Failure.
      reset $f = f + \varepsilon s^C$ where ε is maximal subject to $l \leq f + \varepsilon s^P \leq u$
      and $v(f) + \varepsilon \leq k$

Figure 2.9 shows a network flow graph where all weights of edges are zero. We want to push 4 units of flow through the graph. Gray edges show that one unit of flow has been pushed through the graph. However, several edges have unsatisfied lower bound capacities. For instance, the edge (s, v_1) requires two units of flow and the current flow only

Figure 2.9 A network flow graph with one unit of flow.

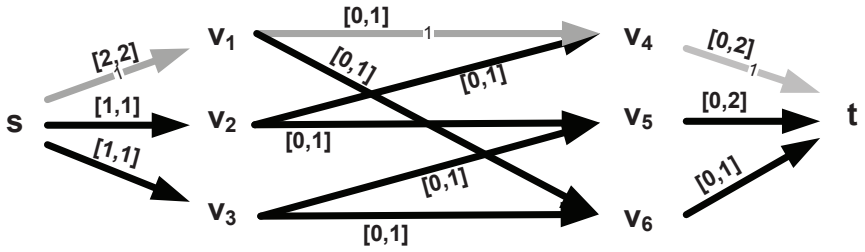
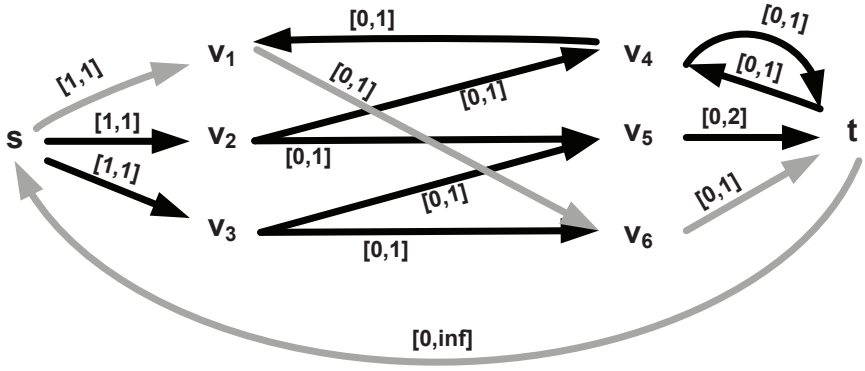
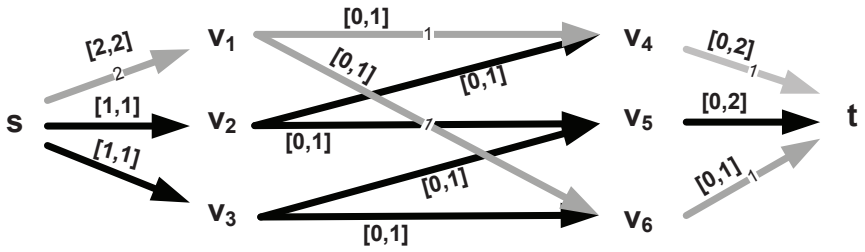


Figure 2.10 A residual graph of the network flow graph from Figure 2.9.



pushes one unit of flow through this edge. Hence, we construct a residual graph and find a path from v_1 to s that is highlighted in gray (Figure 2.10). Note that we can push only one unit of flow through this cycle. In this case it is enough to satisfy the lower bound capacity demand for the edge (s, v_1). Finally, we update the flow in the original graph to accommodate this unit of flow (Figure 2.11). The algorithm run in $O(v(f))O(SP)$, where $O(SP)$ is the time complexity of a shortest path algorithm. Note that the successive shortest path algorithm is pseudo-polynomial as its complexity depends on the size of the flow. However, the size of the flow in this thesis is always $O(|V|)$ which makes the algorithm sufficient for our purposes. The best polynomial time complexity to solve this problem is $O((|E| \log |V|)(|E| + |V| \log |V|))$ [AMO93].

Figure 2.11 A network flow graph with two units of flow.



To build a constraint propagator we often need to determine the cost of pushing an extra unit of flow through an edge or removing a unit of flow from an edge. We use the following theorems to answer these questions:

Theorem 2.1 *[AMO93] Let f be a min cost flow in a graph $G = (V, E)$ and $e = (v_i, v_j)$ be an edge such that $f(e) = 1$. Let P be a shortest path in the residual network $G_f = (V, E_f)$ from the vertex v_i to the vertex v_j. A minimum cost flow that* decreases by one *the flow through the edge e can be constructed as follows. We augment one unit of flow from vertex v_i to vertex v_j along the path p and decrease the amount of flow through e by one. The cost of the optimum flow f' is $w(f) - w(e) + w(p)$.*

Theorem 2.2 *[AMO93] Let f be a min cost flow in a graph $G = (V, E)$ and $e = (v_i, v_j)$ be an edge such that $f(e) = 0$. Let P be a shortest path in the residual network $G_f = (V, E_f)$ from the vertex v_j to the vertex v_i. A minimum cost flow that* increases by one *the flow through the edge e can be constructed as follows. We augment one unit of flow from vertex v_j to vertex v_i along the path p and increase the amount of flow through e by one. The cost of the optimum flow f' is $w(f) + w(e) + w(p)$.*

2.2 Integer Linear Programming

In this section we recall some integer linear programming theory. We mostly focus on tractable classes of integer linear programming (ILP) which can be reduced to the network flow problem, because we use these tractable classes to construct a propagator for the SEQUENCE constraint and its generalisations. We also consider the Fourier-Motzkin elimination procedure that we use to construct a decomposition of the SEQUENCE constraint.

The definitions in this section are based on [Sch86].

2.2.1 Basic definitions

A set of vectors in $\mathbb{R}^n$ is called a polyhedron if $P = \{x | Ax \leq b\}$. *Linear programming* (LP) is the problem of maximising or minimising a linear functional over a polyhedron which is defined as a set of linear inequality constraints. One of the standard forms of LP with n variables and m inequalities is:

$$
\begin{aligned}
&maximise && c_1x_1 + c_2x_2 + \ldots + c_nx_n \\
&subject\ to && a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1 \\
& && a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2 \\
& && \vdots \\
& && a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq b_m \\
& && x \geq 0
\end{aligned}
$$

We can rewrite this system in the matrix form:

$$maximise \quad cx \tag{2.3}$$

$$subject\ to \quad Ax \leq b \tag{2.4}$$

$$x \geq 0 \tag{2.5}$$

where $A \in \mathbb{R}^{m \times n}$ is a LP *matrix of coefficients*, $c \in \mathbb{R}^n$ is a cost vector, $b \in \mathbb{R}^m$ is a right-hand side vector and $x \in \mathbb{R}^n$ is a vector of n variables:

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ & & \vdots & \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix},$$

$x = \left(x_1, x_2 \ldots, x_n\right)^T$, $c = \left(c_1, c_2 \ldots, c_n\right)$ and $b = \left(b_1, b_2 \ldots, b_m\right)^T$.

One of the most important results in LP is von Neumann's Strong Duality Theorem:

Theorem 2.3 (Duality theorem of linear programming [Sch86]) *Let A be a coefficients matrix and let b and c be vectors then*

$$
\begin{aligned}
&maximise && cx \\
&subject\ to && Ax \leq b \\
& && x \geq 0
\end{aligned}
\quad \Leftrightarrow \quad
\begin{aligned}
&minimise && yb \\
&subject\ to && yA \geq c \\
& && y \geq 0
\end{aligned}
$$

provided that both sets of solutions are not empty.

A similar duality reformulation can be done for other standard forms with maximisation and minimisation. We call the original LP *primal* and the other LP program *dual*. In the most general form the LP duality equations are as follows:

$$
\begin{array}{lcc|lcc}
 & \textbf{Primal } LP & & & \textbf{Dual } LP & \\
maximise & dx + ey + fz & & minimise & au + bv + cw & \\
subject\ to & Ax + By + Cz \leq a & & subject\ to & uA + vD + wG \geq d & \\
 & Dx + Ey + Fz = b & \Longleftrightarrow & & uB + vE + wH = e & \\
 & Gx + Hy + Kz \geq c & & & uC + vF + wK \leq f & \\
 & x \geq 0 & & & u \geq 0 & \\
 & z \leq 0 & & & w \leq 0 &
\end{array}
$$

In the case of a maximisation problem (Equations (2.3)–(2.5)), any vector x that satisfies constraints (2.4) is called a *feasible* solution. If LP does not have feasible solutions then it is *infeasible*. A function $x \Rightarrow cx$ is called an *objective function* and the value cx is the *objective value* or *cost* of x. Finding a solution of LP with n variables that can be encoded with L bits can be done in $O(n^{3.5}L)$ pseudo-arithmetic operations on numbers with $O(L)$ digits. Note that the complexity is counted in the number of operations, and operations on the rationals can in principle expand the size of the numbers (repeated multiplications can blow-up the representation exponentially). However, for practical purposes, typical LP implementations prevent the blow-up of number representation by limiting the precision to b bits throughout the execution; solvability by Linear Programming is widely regarded as synonymous to strong tractability, and provably sub-exponential LP algorithms exist [MSW96]. If we extend Equation (2.5) with integrality requirement for variables, we obtain an *integer linear program* (ILP) . Integer linear programming belongs to the class of NP-complete problems.

In the next section we consider a special class of *ILP* that admit a polynomial time algorithm.

2.2.2 Tractable classes

A matrix A is *totally unimodular* if each sub-determinant of A is 0, $+1$ or -1.

Theorem 2.4 (Totally unimodular matrix [Sch86]) *Let A be a totally unimodular matrix and let b be an integral vector. Then the polyhedron $P := \{x | Ax \leq b\}$ is integral.*

In other words, if a ILP matrix A is totally unimodular, then we can solve the LP relaxation of the problem and find a feasible integral solution.

Corollary 2.1 *[Sch86] Let A be a totally unimodular matrix, and let b and c be integral vectors. Then both problems in the LP-duality equation*

$$\begin{array}{lll} maximise & cx & \\ subject\ to & Ax \leq b & \Leftrightarrow \\ & x \geq 0 & \end{array} \quad \begin{array}{ll} minimise & yb \\ subject\ to & yA \geq c \\ & y \geq 0 \end{array}$$

have an integral optimum solution.

There exists a polynomial time procedure to detect whether a matrix is totally unimodular [Tru90]. Moreover, there are a number of sufficient conditions for the unimodular matrices with entries 0,$+1$ or -1 [Sch86]. In this thesis, we are interested in subclasses of totally unimodular matrices that are called interval matrices. A is a *interval* matrix if its entries are 0 and 1 and each column of A has its 1's in consecutive locations (assuming some linear order of the rows of A). Interval matrices are totally unimodular. An interesting property of interval matrices is that an interval matrix can be transformed into a matrix such that has each column of the transformed matrix has exactly one $+1$ and exactly one -1 in each column and the rest of the entries are zeros. The transformation procedure is given in [AMO93]. We demonstrate it on an example. Consider the following ILP problem:

$$minimise \sum_{i=1}^{5} x_i \tag{2.6}$$

$$x_1 + x_2 + x_3 \leq 2, \tag{2.7}$$

$$x_2 + x_3 + x_4 \leq 3, \tag{2.8}$$

$$x_3 + x_4 + x_5 \leq 4, \tag{2.9}$$

where $x_i \in [0, 2]$.

First, we reformulate this problem into a problem with equalities by introducing slack variables y_i:

$$minimise \sum_{i=1}^{5} x_i$$

$$x_1 + x_2 + x_3 + y_1 = 2,$$

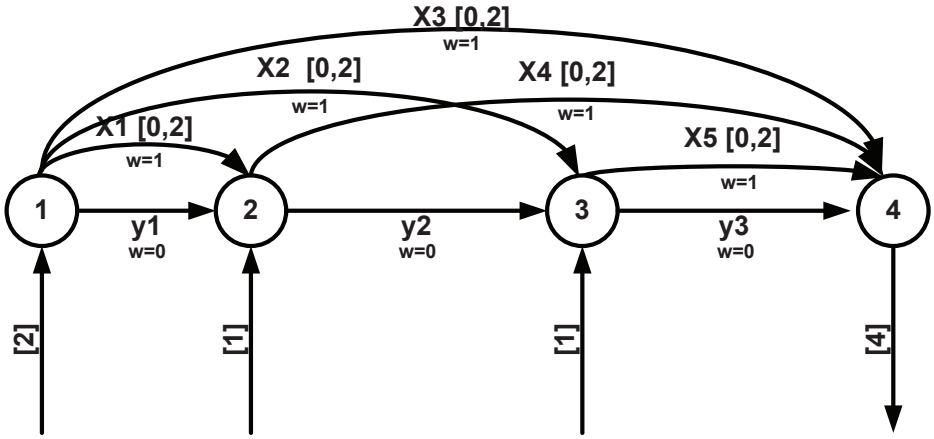
$$x_2 + x_3 + x_4 + y_2 = 3,$$

$$x_3 + x_4 + x_5 + y_3 = 4,$$

where $y_i \geq 0$. We can express it as an integer linear program in the matrix form:

$$\begin{pmatrix} 1&1&1&0&0&1&0&0 \\ 0&1&1&1&0&0&1&0 \\ 0&0&1&1&1&0&0&1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -3 \\ -4 \end{pmatrix} \tag{2.10}$$

Figure 2.12 A minimum cost flow problem that corresponds to the ILP problem.



Now we add a redundant constraint $0x + 0y = 0$ and get

$$\begin{pmatrix} 1&1&1&0&0&1&0&0 \\ 0&1&1&1&0&0&1&0 \\ 0&0&1&1&1&0&0&1 \\ 0&0&0&0&0&0&0&0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -3 \\ -4 \\ 0 \end{pmatrix} \tag{2.11}$$

Finally, we subtract ith from $i+1$th row, $i = 1, \ldots, 3$ and obtain the following equivalent ILP.

$$\begin{pmatrix} 1&1&1&0&0&1&0&0 \\ -1&0&0&1&0&-1&1&0 \\ 0&-1&0&0&1&0&-1&1 \\ 0&0&-1&-1&-1&0&0&-1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \\ -1 \\ 4 \end{pmatrix} \tag{2.12}$$

The resulting matrix can be transformed into a network flow problem if we consider the resulting matrix A as an incidence matrix of a graph. We introduce a vertex for each row and an edge for each column. If the value 1 is in ith row and the value -1 is in the kth row, then we draw a directed edge form the ith vertex to the kth vertex. The bounds restrictions on the variables x become capacity bounds on the corresponding edges. The right hand side vector defines the flow supply and demand for all vertices. If the value is negative then the vertex has a flow demand and otherwise it has a flow supply. Edges that correspond to the variables x have weights of one and the remaining edges have zero weights. The network flow that corresponds to the example is shown in Figure 2.12.

The following theorem gives another sufficient condition for transforming a ILP problem to a network flow problem.

Theorem 2.5 *[AMO93] Any linear program that contains (a) at most one $+1$ and at most one -1 in each column, or (b) at most one $+1$ and at most one -1 in each row, can be transformed into a minimum cost flow problem.*

2.2.3 Fourier-Motzkin elimination

In this section we consider a technique for solving a system of linear inequalities. The idea is to replace a system of linear inequalities over n variables with an equivalent system of linear inequalities over $n-1$ variables. Hence, on every step, the procedure eliminates one variable, but might increase the number of inequalities $O(n^2)$ times. So, this method is not polynomial for a general LP or ILP. We show how this method works on a system of m linear inequalities over n variables:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \quad \dots \quad + a_{1n}x_n \le b_1 \\ a_{21}x_1 + a_{22}x_2 + \quad \dots \quad + a_{2n}x_n \le b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \quad \dots \quad + a_{mn}x_n \le b_m \end{aligned}$$

We can assume that first column coefficients contain only $\{-1, 0, +1\}$ entries. This can be achieved by multiplying each inequality by a positive scalar. Then we reorder all inequalities and partition them in three groups:

$$x_1 + a^i x' \le b_i \qquad i = 1, \dots, m' \tag{2.13}$$

$$-x_1 + a^i x' \le b_i \quad i = m'+1, \dots, m'' \tag{2.14}$$

$$a^i x' \le b_i \quad i = m''+1, \dots, m, \tag{2.15}$$

where $x' = (x_2, \dots, x_n)$ and $a^1, \dots, a^m$ are the rows of A with first entries deleted. Equations (2.13)–(2.14) are equivalent to

$$max_{m'+1 \le j \le m''}(a^j x' - b_j) \quad \le x_1 \le \quad max_{1 \le j \le m'}(b_j - a^i x'). \tag{2.16}$$

Hence, we can eliminate the variable x_1 and obtain an equivalent system:

$$\begin{aligned} a^j x' - b_j \le \quad b_j - a^i x' \quad i = 1, \dots, m'; j = m'+1, \dots, m'' \\ a^i x' \le \qquad b_j \qquad i = m''+1, \dots, m. \end{aligned}$$

This system has $m'(m''-m') + m - m''$ constraints, and $n-1$ variables. Any solution of the new system can be extended to a solution of the original system if we select x_1 with respect to (2.16). We can apply this elimination step $n-1$ times. If the resulting system has a solution then it can be extended to the solution of a system.

As pointed out above, this procedure has an exponential worst case behaviour, as the number of constraints grows quadratically at each step.

Consider how the procedure works on the following four linear constraints:

$$x_1 + x_2 + x_3 \geq 2, \tag{2.17}$$

$$x_2 + x_3 + x_4 \leq 3, \tag{2.18}$$

$$x_1 + x_3 + x_4 \leq 4, \tag{2.19}$$

$$x_3 + x_4 \geq 4, \tag{2.20}$$

First, we eliminate the variable x_1.

$$\begin{aligned}
-x_1 - x_2 - x_3 &\leq -2,\\
x_1 + x_3 + x_4 &\leq 4,\\
x_2 + x_3 + x_4 &\leq 3,\\
x_3 + x_4 &\geq 4,\\
&\Updownarrow\\
2 - x_2 - x_3 &\leq 4 - x_3 - x_4,\\
x_2 + x_3 + x_4 &\leq 3,\\
x_3 + x_4 &\geq 4,\\
&\Updownarrow\\
-x_2 &\leq 2 - x_4,\\
x_2 + x_3 + x_4 &\leq 3,\\
x_3 + x_4 &\geq 4,
\end{aligned}$$

Second, we eliminate the variable x_2.

$$\begin{aligned}
-x_2 + x_4 &\leq 2,\\
x_2 + x_3 + x_4 &\leq 3,\\
x_3 + x_4 &\geq 4,\\
&\Updownarrow\\
x_4 - 2 &\leq 3 - x_3 - x_4,\\
x_3 + x_4 &\geq 4,\\
&\Updownarrow\\
2x_4 &\leq 5 - x_3,\\
x_3 + x_4 &\geq 4,
\end{aligned}$$

Third, we eliminate the variable x_3.

$$\begin{aligned} 2x_4 + x_3 &\leq 5, \\ -x_3 - x_4 &\leq -4, \\ &\Updownarrow \\ 4 - x_4 &\leq 5 - 2x_4 \end{aligned}$$

We obtain that $x_4 \leq 1$. Hence, we set x_4 to 1 which is one of the possible values. Then, x_3 is equal to 3, $x_2 = -1$ and $x_1 = 0$.

2.3 Formal languages

In this section we introduce some background in formal language theory. First, we consider regular and context-free grammars. Then we introduce several types of grammars that are in between these two grammars in the Chomsky hierarchy. We also consider the CYK algorithm for parsing a string.

The definitions in this section are based on [Roz97] and [HU79].

2.3.1 Basic definitions

An *alphabet* $\Sigma = \{a_1, \ldots, a_n\}$ is a finite, non empty set of *letters* or *symbols*. We use lower case letters to denote letters from the alphabet, e.g., $\Sigma_e = \{a, b, c\}$. A *string* or *word* is a finite sequence of symbols chosen from some alphabet. For example, $aabbc$ is a string from Σ_e. An *empty string* is the string with zero occurrences of symbols. The *length* of a string is the number of positions for symbols in the string. We define Σ^k to be the set of strings of length k. The set of all strings over an alphabet Σ is called Σ^*. Note that $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \ldots$. A *language* L over Σ is a set of words over Σ. A *universal language* over Σ is a set of all words over Σ^*. We denote $|L|$ the cardinality of a language. We can define basic operations over languages, such as union, intersection or complement.

The Hamming distance between two strings is the number of positions where these strings are different. For example, the Hamming distance between strings $abbc$ and $cabb$ is 3. *The edit distance* between two strings is the minimum number of deletion, insertion and substitution operations required to convert one string into the other. For example, the edit distance between $abbc$ and $cabb$ is 2 as we can delete the last symbol in the first string and the first symbol in the second string.

2.3.2 Grammars

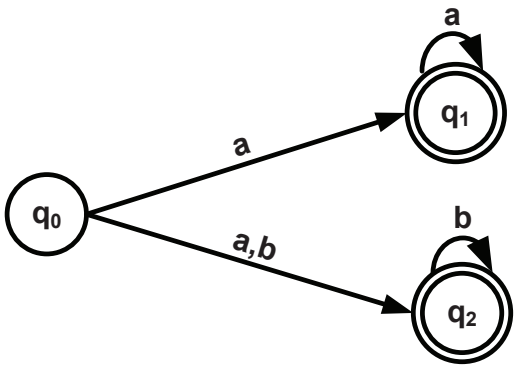
A *formal grammar* (or grammar) is a set of rules for forming strings in a language. A *phrase-structure grammar* is a quadruple $G = \langle \Sigma, H, P, S \rangle$, where Σ and H are disjoint alphabets, $S \in H$, and $P \subseteq V_G^* N V_G^* \times V_G^*$, for $V_G = N \cup T$. The elements of H are called *non-terminal* symbols, the elements of Σ are called *terminal* symbols, S is the *start symbol* , and P the set of *production rules* , $(\beta, \alpha) \in P$ that are written in the form $\beta \to \alpha$. The grammar is *context-free* (CFG) if each production $\beta \to \alpha$ has $\beta \in H$. A grammar that generates palindromes is an example of CFG, $G = (\{a, b\}, \{S\}, \{S \to aSa, S \to bSb, S \to a, S \to b, S \to \varepsilon\}, S)$. The grammar is *linear* if each production $\beta \to \alpha$ has $\beta \in H$ and $\alpha \in \Sigma^* \cup \Sigma^* H \Sigma^*$. The grammar is *right-linear* if each production $\beta \to \alpha$ has $\beta \in H$ and $\alpha \in \Sigma^* \cup \Sigma^* H$. The grammar is *left-linear* if each production $\beta \to \alpha$ has $\beta \in H$ and $\alpha \in \Sigma^* \cup H\Sigma^*$. The grammar is *regular* if each production $\beta \to \alpha$ has $\beta \in H$ and $\alpha \in \Sigma \cup \Sigma H$. A grammar that generates words a^*b^* is an example of regular grammar, $G = (\{a, b\}, \{S, S_1\}, \{S \to aS, S \to \varepsilon, S \to a, S \to S_1, S_1 \to bS_1, S_1 \to b, S_1 \to \varepsilon\}, S)$.

We do not consider grammars that are more expressive then context-free grammars in this work. Hence, for each rule $\beta \to \alpha$, $\beta \in H$. A grammar is *simple* if and only if for each pair of non-terminal and terminal there exists at most one production of the form $A \to a\alpha$, where $a \in \Sigma$, $\alpha \in (\Sigma \cup H)^*$. A grammar is *even linear* if and only if its productions are of the form $A \to \mathbf{a}B\mathbf{b}$, where $\mathbf{a}, \mathbf{b} \in \Sigma^+$, $A, B \in H$ and $|\mathbf{a}| = |\mathbf{b}|$.

A context-free grammar is in *Chomsky normal form* (CNF) if and only if all productions are of the form $A \to BC$ where B and C are non-terminals or $A \to a$ where a is a terminal. Any context-free grammar can be converted to one that is in Chomsky normal form with at most a linear increase in its size. For example, the grammar that generates palindromes can be converted into the following Chomsky normal form grammar, $G' = (\{a, b\}, \{S, S_{a_1}, S_{a_2}, S_{b_1}, S_{b_2}\}, \{S \to S_{a_1}S_{a_2}, S_{a_1} \to a, S_{a_2} \to SS_{a_1}, S_{a_2} \to a, S_{b_1} \to b, S \to S_{b_1}S_{b_2}, S_{b_2} \to SS_{b_1}, S_{b_2} \to b, S \to a, S \to b\}, S)$. A context free grammar in CNF is *acyclic*, G_a, if and only if there exists a partial order $\prec$ of the non-terminals, such that for every production $A_1 \to A_2A_3$, $A_1 \prec A_2$ and $A_1 \prec A_3$. A context-free grammar is in *Greibach normal form* if all productions are of the form $A \to a\alpha$ where a is a terminal and α is a sequence of non-terminals or $A \to a$ where a is a terminal. Any context-free grammar can be converted to one that is in Greibach normal form with at most a polynomial increase in its size.

The derivation relation $\Rightarrow_G$ induced by G is defined as follows: for any $u, v \in \Sigma^*$,

Figure 2.13 A non-deterministic finite automaton.



$uAv \Rightarrow_G u\alpha v$ if there exists a production $A \rightarrow \alpha$ in P. The transitive, reflexive closure of $\Rightarrow_G$ is denoted by $\Rightarrow_G^*$. A string in $s \in \Sigma^*$ is *generated* by G if $S \Rightarrow_G^* s$. For example, to generate a palindrome string aba using G' from the example above, we can use the following derivation: $S \Rightarrow S_{a_1}S_{a_2} \Rightarrow aS_{a_2} \Rightarrow aSS_{a_1} \Rightarrow aSa \Rightarrow aba$. The set of all strings generated by G is denoted $L(\mathcal{A})$. The membership question in the formal language theory is the following. Given a string w in Σ^*, decide whether or not w is in $L(G)$. A grammar G is *unambiguous* if and only if there exists a single derivation for each word in the language generated by the grammar G.

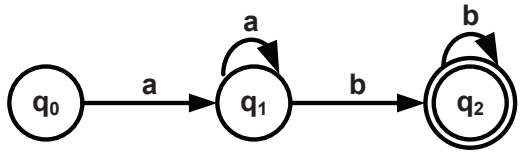
2.3.3 Automata

A non-deterministic finite automaton (*NFA*) $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$ consists of an alphabet Σ, a finite set of states Q, an initial state $q_0 \in Q$, a set of final states $F \subseteq Q$, and a transition function $\delta : (Q \times \Sigma) \rightarrow \mathcal{P}(Q)$, where $\mathcal{P}$ is the power set of Q. Figure 2.13 shows a non-deterministic finite automaton that has three states $\{q_0, q_1, q_2\}$. The set of final states F includes two states q_1 and q_2. The automaton is non-deterministic, because from the initial state q_0 on seeing the letter a, the automaton non-deterministically chooses the next state. We use a double circle to denote a final state.

The automaton $\mathcal{A}$ is deterministic (*DFA*) if $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. We extend δ to subsets Q' of Q and strings aw with $a \in \Sigma$ and $w \in \Sigma^*$ as follows: $\hat{\delta}(Q', aw) = \hat{\delta}(Q'', w)$, where $Q'' = \cup_{q \in Q'} \delta(q, a)$. A string $s \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if $\hat{\delta}(\{q_0\}, s) = (Q', \varepsilon)$ such that $Q' \cap F \neq \emptyset$, that is, if starting from the initial state q_0 we can reach one of the final states using the transition function δ. Figure 2.13 shows a deterministic finite automaton has a set of three states $\{q_0, q_1, q_2\}$. The set of final states F consists of the state q_2. The initial state is q_0.

The set of all strings accepted by $\mathcal{A}$ is denoted $L(\mathcal{A})$. Both *DFAs* and *NFAs* accept

Figure 2.14 A deterministic finite automaton.



precisely the regular languages.

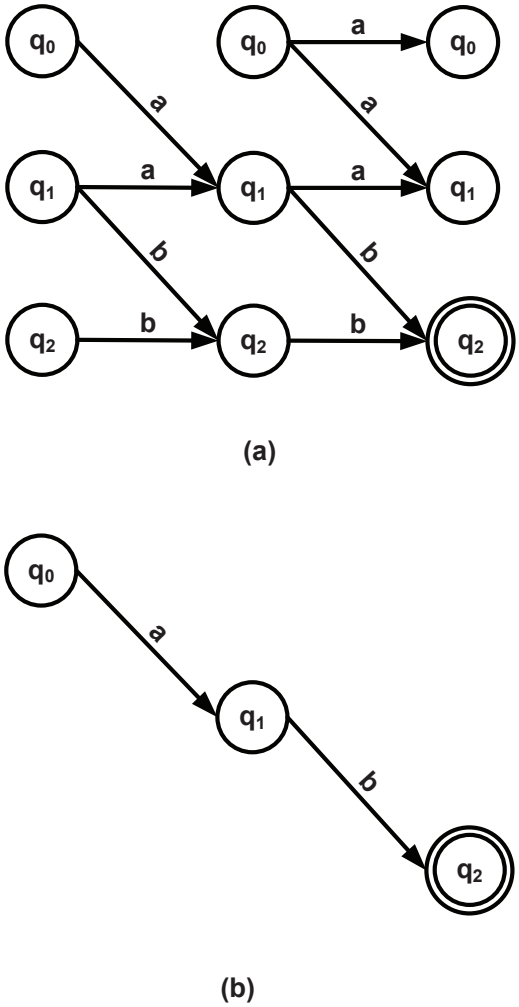
Given an automaton $\mathcal{A}$, we write $unfold_n(\mathcal{A})$ for the unfolded and layered form of $\mathcal{A}$ that just accepts words of length n that belong to the regular language. We construct the $unfold_n(\mathcal{A})$ automaton for the automaton $\mathcal{A}$ in the following way. Each layer of unfolded and layered automaton contains all states of the original automaton. We create a transition from state q_i at layer k to state q_j at layer $k+1$ labeled with a, $k = 1, \ldots, n$ if there is a transition $\delta(q_i, a) = q_j$ in the original automaton. We mark the initial state at layer 1 as initial state of the resulting automaton. We mark the set of final state at layer $n+1$ as final states of the resulting automaton. Figure 2.15(a) shows the $unfold_n(\mathcal{A})$ automaton of the automaton in Figure 2.14 where $n = 2$. We write $\min(\mathcal{A})$ for the canonical form of $\mathcal{A}$ with minimal number of states. Figure 2.15(b) shows the $\min(unfold_n(\mathcal{A}))$ for $unfold_n(\mathcal{A})$ from Figure 2.15(a) . We denote $f_n(\mathcal{A})$ the transformation of an automaton to automaton that excepts strings of n among the set of strings accepted by $\mathcal{A}$. We write $f_n(\mathcal{A}) \ll g_n(\mathcal{A})$ if and only if $f_n(\mathcal{A}) \leq g_n(\mathcal{A})$ for all n, and there exists $\mathcal{A}$ such that $\log \frac{g_n(\mathcal{A})}{f_n(\mathcal{A})} = \Omega(n)$. That is, $g_n(\mathcal{A})$ is never smaller than $f_n(\mathcal{A})$ and there are cases where it is exponentially larger.

A *non-deterministic pushdown automaton* (PDA) P over an alphabet Σ is a triple $\langle S, Q, \Sigma, T, \delta, Q_0, F \rangle$, where S is the initial stack of P, Q is a finite set of states, Σ is a set of input symbols, T is the set of stack symbols, δ is the transition function from $Q \times (\Sigma \cup \{\varepsilon\}) \times T$ (a triple of the current state, input symbol and the top of the stack) to $Q \times T^*$ (a pair of a new state and a sequence of stack symbols that are pushed on the stack), Q_0 is the start state, $F \subseteq Q$ a set of accepting states. A word w is accepted by *PDA* P if starting from the initial state Q_0 we can reach one of the final states using the transition function δ. We call all words accepted by PDA P the language generated by P, denoted $L(P)$.

Pushdown automata are equivalent to context-free grammars. For every context-free grammar G there exists a pushdown automaton P such that $L(G)$ is equivalent to the language accepted by P. The reverse also holds.

A pushdown automaton P over an alphabet Σ is *deterministic* if the set δ of transitions satisfies the following conditions for all $(y, q, z) \in Q \times (\Sigma \cup \{\varepsilon\}) \times T$: $Card(\delta(q, y, z)) \leq 1$

Figure 2.15 (a) An unfolded automaton of the automaton from Figure 2.14 that accepts words of length 2. (b) The minimised automaton.

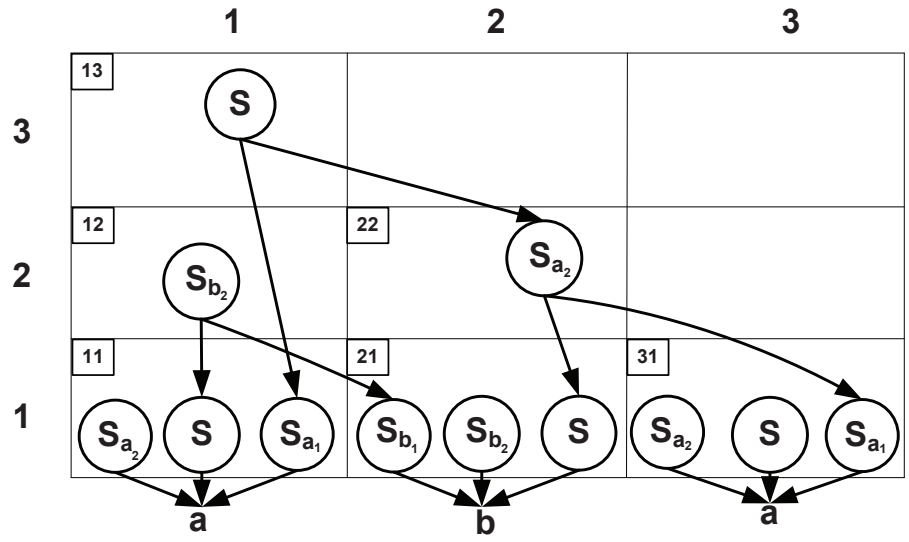


$\delta(q, \varepsilon, z) \neq 0 \Longrightarrow \delta(q, a, z) = 0, (a \in A)$. In other words, a deterministic *PDA* has at most one transition for the same combination of input state, symbol and the top stack symbol. A *deterministic context-free language* is a language recognised by a deterministic PDA.

2.3.4 Cocke-Younger-Kasami's parser

We consider a parsing algorithm that was proposed by Cocke, Younger and Kasami (*CYK*) to determine whether a string can be generated by a context free grammar $G = \langle \Sigma, H, P, S \rangle$. We assume that grammar is in CNF. Algorithm 2.2 shows the pseudocode for the CYK parser. The algorithm constructs all possible derivations for all possible substrings of the original string. Using the dynamic programming principle, it builds a derivation of a string of length k that starts at position i using two substrings of length $j < k$, $j = i, \ldots, k-1$. The first substring starts at position i and is of the length j and the second one starts at

Figure 2.16 A dynamic programming table that is constructed by Algorithm 2.2



position $i + j$ and finishes at position $i + k$. The time complexity of the algorithm is $O(n^3|G|)$.

Algorithm 2.2 Cocke-Younger-Kasami's algorithm

procedure CYK-ALGORITHM($G, w = (a_1, \ldots, a_n)$)

n = |w|;

for $i = 1$ **to** n **do**

$V[i, 1] = \{A | A \to a_i \in G\}$;

for $j = 2$ **to** n **do**

for $i = 1$ **to** $n - j + 1$ **do**

$V[i, j] = \emptyset$;

for $k = 1$ **to** $j - 1$ **do**

$V[i, j] = V[i, j] \cup \{A | A \to BC \in G, B \in V[i, k], C \in V[i + k, j - k]\}$;

if $S \notin V[1, n]$ **then**

return 0;

return 1;

Figure 2.16 shows a dynamic programming table that Algorithm 2.2 constructs on the string aba and the palindromes grammar G' from Section 2.3.2.

2.4 Constraint programming

2.4.1 Basic definitions

A constraint satisfaction problem (CSP) is a triple $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, where $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a set of finite domain variables, $\mathbf{D} = \{D(X_1), \ldots, D(X_n)\}$ is the set of domains of the variables and $\mathbf{C} = \{C_1, \ldots, C_m\}$ is a set of constraints.

Variables. The *domain* of a variable, $D(X)$, represents all possible values that this variable can take. We assume that values in a variable domain are ordered sets of integers. We write $lb(X)$ for the smallest value in $D(X)$ and $ub(X)$ for the greatest value. A domain of a variable can be represented as a set of possible values . In this case, we say that the variable has a *set representation* of its domain. For example, a domain $D(X) = \{1, 2, 3\}$ shows that the variable X can take one of the three possible values. The set representation requires a linear number of bits in the number of possible values to represent the domain. It can represent any variable domain state. Alternatively, a variable can be represented as an interval of values, $[lb(X), ub(X)]$. We call this representation the *interval* or *bounds* representation . For example, a domain $D(X) = [1, 2^{100}]$ shows that the variable X can take one of the 2^{100} possible values. The bounds representation requires a logarithmic number of bits in the number of values to represent the domain. It can represent only a quadratic number of different states of a variable domain.

We denote by $|D(X)|$ the size of the domain representation. In case of the set representation of domains, $|D(X)| = O(ub(X) - lb(X))$. In case of the bounds representation, the size of the domain is $O(log(lb(X)) + log(ub(X)))$. An *assignment* to a variable X_i is a mapping of X_i to a value $j \in D(X_i)$, it is also called literal, and written $X_i = j$. We write $\mathbf{D}(\mathbf{X})$ for sets of literals $\{X_i = j \mid X_i \in \mathbf{X} \wedge j \in D(X_i)\}$ and $\mathcal{P}(\mathbf{D})$ for the set of all such sets.

Constraints. A constraint $C \in \mathcal{C}$ is defined over a set of variables $X_1, \ldots, X_n$, that are called a *scope*, denoted $scope(C) \subseteq \mathbf{X}$. The number of variables in a constraint scope is called the *arity* of the constraint. A *constraint* is a relation over the variables in its scope. For example, $X \neq Y$, $D(X) = D(Y) = [0, 2]$ is a binary constraint . The variables X and Y are in the scope of the constraint. A subset of all possible assignments to the variables in $scope(C)$ that satisfy the relation are called *solutions* of C. We denote $|C(X_1, \ldots, X_k)|$ the size of the constraint. A constraint $C(X_1, \ldots, X_k)$ can be represented *extensionally* as a table of valid (or invalid) assignments to variables $X_1, \ldots, X_k$. The size of constraint in

this case equals to the size of the table. A constraint $C(X_1, \ldots, X_k)$ can be represented *intentionally* as a formula called the characteristic function of the constraint. For example, $X \neq Y$ is an intentional representation of the constraint. An extensional representation of the same constraint is a set of satisfying tuples $[(0,1),(0,2),(1,2),(2,0),(2,1)]$. The size of the constraint is the size of formula plus the size of domains representation $C(\mathbf{X}) = O(\sum_{i=1}^{k} |D(X_i)|)$. Constraints of arity 2 are called *binary* and, otherwise, they are called *non-binary* or *global constraints* .

Local consistencies. *Local consistency* is a property that characterises some necessary conditions on values (or assignments) to belong to a solution. Let Φ denote a consistency level. A constraint C is Φ *consistent* if and only if it satisfies the property Φ [Bes06]. We consider commonly used consistency levels. A *support* for the literal $X_i = v_j$ is an assignment containing $X_i = v_j$ that satisfies C. A variable X_i is *consistent* on C if and only if every value in $D(X_i)$ has support on C. A constraint C is *domain consistent* (*DC*) if and only if each variable in its scope is consistent on C. A *bound support* on C is a support, such that each variable X_i takes a value from the interval $[lb(X_i), ub(X_i)]$. A variable X_i is *bounds consistent* on C if and only if its lower and upper bounds values have bound support on C. A constraint C is *bounds consistent* (*BC*) if and only if all variables in its scope are bounds consistent . *Range consistency* (*RC*) is stronger than bounds consistency but is weaker than domain consistency. A variable X_i is *range consistent* on C if and only if its values have bound support on C. A constraint C is *range consistent* (*RC*) if and only if all its variables are $range\ consistent$. A *CSP* is *DC*/*RC*/*BC* if and only if each constraint is *DC*/*RC*/*BC*.

A *CSP* over variables $\mathbf{X}$ is *singleton domain consistent* (*SDC*) with respect to variables $\mathbf{X}$ if and only if for each variable $X_i \in \mathbf{X}$, we can assign any value in the domain of X_i and make the resulting subproblem domain consistent . Consider, for example, a problem with two constraints: $X_1 \neq X_2$, $X_1 + X_2 = X_3$, where $D(X_1) = \{0,1,2\}$, $D(X_2) = \{1,3\}$ and $D(X_3) = \{1,2,3\}$. All constraints are domain consistent, but enforcing SDC on these constraints removes the value 1 from the domain of X_1 and the value 2 from the domain of X_3. If we set $X_1 = 1$ or $X_3 = 2$ then the system of constraints is inconsistent. A *CSP* over variables $\mathbf{X}$ is *singleton bounds consistent* (*SBC*) with respect to variables $\mathbf{X}$ if and only if for each variable $X_i \in \mathbf{X}$, we can assign the upper (lower) bound value of X_i and make the resulting subproblem bounds consistent . In the previous example enforcing *SBC* does not prune any values. Similarly, we can define *SDC*/*SBC* for a subset of variables.

The procedure that enforces singleton domain or bounds consistency is called the *failed*

literal test (FLT) [Fre95]. We describe the failed literal test here as it is used independently to *SDC*/*SBC* in this thesis. For each variable-value pair (X_i, j) of an unset variable X_i, the failed literal test sets $X_i = j$, enforces *DC* or *BC*, checks whether the remaining problem is inconsistent so that one of the problem variables has an empty domain. If so, the value j is removed from domain of the variable X_i.

We will compare local consistency properties applied to sets of constraints, C_1 and C_2 which are logically equivalent. As in [DB97], we say that a local consistency property Φ on C_1 is as strong as Ψ on C_2 (written $\Phi(C_1) \preceq \Psi(C_2)$) iff, given any domains, if Φ holds on C_1 then Ψ holds on C_2; we say that Φ on C_1 is stronger than Ψ on C_2 (written $\Phi(C_1) \prec \Psi(C_2)$) iff Φ on C_1 is as strong as Ψ on C_2 but not vice verse; Φ on C_1 is equivalent to Ψ on C_2 (written $\Phi(c_1) \equiv \Psi(c_2)$) iff Φ on C_1 is as strong as Ψ on C_2 and vice verse; they are incomparable otherwise. (written $\Phi(c_1) \bowtie \Psi(c_2)$).

A constraint C is *monotone* if and only if there exists a total ordering $\prec$ of the domain values such that for any two values v, w, if $v \prec w$ then v is substitutable to w in any support for C [BHH+06b] .

Constraint propagators. A constraint C can have an inference algorithm or a *constraint propagator* . A constraint propagator is an algorithm which takes as input the domains of the variables in $scope(C)$ and returns *restrictions* of these domains. Constraint propagators are also called *filtering algorithms* or *propagation algorithms*. Following [SS04], we can formally define a propagation algorithm for a global constraint as a function:

Definition 2.5 (Propagator) *A propagator f for a constraint C is a polynomial time computable function $f : \mathcal{P}(\mathbf{D}) \rightarrow \mathcal{P}(\mathbf{D})$, such that*

1. *f is* monotone, *i.e.,* $\mathbf{D}'(\mathbf{X}) \subseteq \mathbf{D}(\mathbf{X}) \implies f(\mathbf{D}'(\mathbf{X})) \subseteq f(\mathbf{D}(\mathbf{X}))$,
2. *f is* contracting, *i.e.,* $f(\mathbf{D}(\mathbf{X})) \subseteq \mathbf{D}(\mathbf{X})$ *and*
3. *f is* idempotent, *i.e.,* $f(f(\mathbf{D}(\mathbf{X}))) = f(\mathbf{D}(\mathbf{X}))$.

If a literal $X_i = j$ is in $\mathbf{D}(\mathbf{X}) \setminus f(\mathbf{D}(\mathbf{X}))$ then $X_i = j$ does not belong to any solution of C given $\mathbf{D}(\mathbf{X})$. If f detects that C has no solutions under $\mathbf{D}(\mathbf{X})$ then $f(\mathbf{D}(\mathbf{X})) = \emptyset$.

A propagator for the constraint C achieves Φ consistency level if and only if after it finishes the constraint C is Φ-consistent. The weakest form of consistency the propagator can achieve is detection of disentailment. A propagator *detects disentailment* if when no possible assignment is a solution of C then $f(\mathbf{D}(\mathbf{X})) = \emptyset$. A consistency checker for a

constraint C is a function that returns 0 when it detects that no possible assignment is a solution of the constraint and 1 otherwise, rather than restricting domains.

Definition 2.6 (Consistency checker) *A consistency checker f for a constraint C is a polynomial time computable function $f : \mathcal{P}(\mathbf{D}) \rightarrow \{0,1\}$ such that f is monotone, i.e., $\mathbf{D}'(\mathbf{X}) \subseteq \mathbf{D}(\mathbf{X}) \implies f(\mathbf{D}'(\mathbf{X})) \leq f(\mathbf{D}(\mathbf{X}))$. If $f(\mathbf{D}(\mathbf{X})) = 0$ then no possible assignment under $\mathbf{D}(\mathbf{X})$ is a solution of C.*

A propagator enforces *domain consistency* if and only if after it finishes the constraint C is domain consistent. A propagator detects *domain disentailment* when it fails if and only if there exists no solution of C. A propagator enforces *bounds consistency* on C if and only if after it finishes the constraint C is bounds consistent. A propagator detects *bounds disentailment* when it fails if and only if there exists no solution of C such that each variable takes value between its lower and upper bounds. We consider a modification of a constraint propagator that detects domain or bounds disentailment that returns 0 when it detects disentailment and 1 otherwise. The propagator is called a consistency checker.

We can obtain a polynomial time *DC* consistency checker f_C of a constraint C from a polynomial time *DC* propagator f_P for C and vice verse [BHHW07]. Given the propagator f_P, the corresponding consistency checker f_C is defined as:

$$f_C(\mathbf{D}(\mathbf{X})) = \begin{cases} 0 & f_P(\mathbf{D}(\mathbf{X})) = \emptyset \\ 1 & otherwise \end{cases} \tag{2.21}$$

Conversely, given f_C, the propagator f_P is

$$f_P(\mathbf{D}(\mathbf{X})) = \mathbf{D}(\mathbf{X}) \setminus \{X_i = j \mid f_C(\mathbf{D}(\mathbf{X})|_{X_i=j}) = 0\} \tag{2.22}$$

where $\mathbf{D}(\mathbf{X})|_{X_i=j} = \mathbf{D}(\mathbf{X}) \setminus \{X_i = k | k \neq j\}$.

Note that Equation 2.22 is effectively running the failed literal test to enforce domain consistency. It assigns each variable a value and checks whether the remaining problem is inconsistent.

Solutions. A *solution* of CSP is an assignment of a value to each variable such that all constraints are satisfied. We denote the set of all solutions of a *CSP* as $sols(\bigwedge_{i=1}^{m} C_i)$. Projection of solutions on a subset of variables $\mathbf{X}' \subset \mathbf{X}$, $sols(\bigwedge_{i=1}^{m} C_i)[\mathbf{X}']$, is a set of solutions where values of variables $\mathbf{X} \setminus \mathbf{X}'$ are ignored.

Constraint graphs. A *constraint graph* of a *CSP* is a graph $G = (V, E)$ such that V corresponds to variables of the problem and E corresponds to constraints. Two nodes are

Figure 2.17 A constraint graph for the *CSP* $X \neq Y$ and $Y \neq Z$, $D(X) = D(Y) = D(Z) = [0, 2]$.

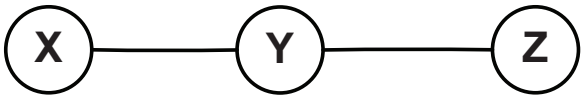
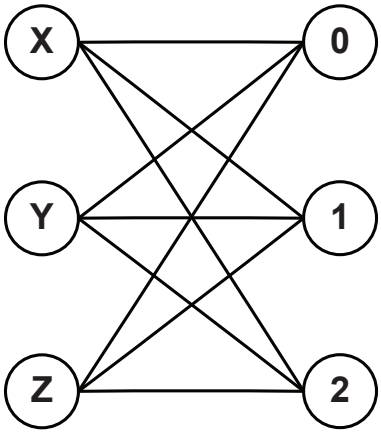


Figure 2.18 A variable-value graph for the *CSP* $X \neq Y$ and $Y \neq Z$, $D(X) = D(Y) = D(Z) = [0, 2]$.



connected by an edge if and only if both of them belong to a scope of some constraint. Consider a *CSP* with two constraints $X \neq Y$ and $Y \neq Z$, $D(X) = D(Y) = D(Z) = [0, 2]$. The constraint graph for this *CSP* is represented in Figure 2.17. A *variable-value graph (VG)* of a *CSP* is a bipartite graph $VG = (X \cup D(X), E)$, where $(X_i, j) \in E$ if and only if $j \in D(X_i)$. An example of variable-value graph for a *CSP* with two constraints is shown at Figure 2.18. A *variable-value graph with* $X_i = j$ *(*$VG_{X_i=j}$*)* is a variable-value graph of the *CSP* with $D(X_i) = j$. If the constraint graph does not have cycles, then enforcing Φ on the *CSP* is equivalent to enforcing Φ consistency on individual constraints. Such a constraint graph is called a Berge-acyclic graph [BFMY83].

2.4.2 Search

Constraint programming search algorithms can be divided into two main groups: local search algorithms and systematic search algorithms. The idea of local search algorithms is to select an initial variable-values combination, and change it, based on a local gradient function, to get closer to a solution on every step. These algorithms show good performance in practice, but they are incomplete as there is no guarantee that an existing solution will be found or that an inconsistent problem will be identified as such. The second group includes complete search algorithms, which systematically explore the search tree of a problem and

in each node of the tree filter inconsistent variables values. In this work we only consider the second group of search algorithms.

Complete constraint programming algorithms work by interleaving branching and inference. Inference is an iterative procedure which, at each iteration, invokes every constraint propagator to narrow the domains of its variables. This invocation happens when one of the variable domains is changed. The iteration stops when none of the constraint propagators can do any more changes. In this case, propagation has reached a *common fixpoint* for all constraints. No inference by the propagators is possible at fixpoint and the search algorithm performs branching by guessing the next variable assignment.

We illustrate the behaviour of search algorithms on the following example of a system of two constraints:

$$X + Y = 7,\ \ X + 1 \geq 2Y, \qquad \textit{with initial domains:} \quad X \in [0, 5],\ \ Y \in [0, 10]$$

The search algorithm starts with inference phase to compute a fixpoint. A possible trace of the fixpoint computation is the following. The lower bound of Y is initially 0 but from the constraint $X + Y = 7$ we deduce that Y cannot take values 10: if it does, then the sum is less than 7, *even if we fix X to its highest allowed value*. Therefore the intervals can be narrowed down to $X \in [0, 5], Y \in [\underline{2}, 10]$. Similarly:

from	$X + Y = 7$,	we deduce:	$X \in [0, 5], Y \in [2, \underline{7}]$;
from	$X + 1 \geq 2y$,	we deduce:	$X \in [\underline{3}, 5], Y \in [2, 7]$;
		and:	$X \in [3, 5], Y \in [2, \underline{3}]$;
back to	$X + Y = 7$,	we now deduce:	$X \in [\underline{4}, 5], Y \in [2, 3]$.

At this point we have reached a common fixpoint for both constraints, because we cannot deduce that the domains need to be narrowed any further. In this case the search algorithm performs branching based on some heuristic. Suppose we branch on the variable X and assign it to the first unassigned value 4. This assignment triggers an inference procedure. By the first constraint, we set the variable Y to 3. This gives a solution for this problem.

This iterative algorithm is guaranteed to compute the fixpoint in polynomial time if the domains of the variables are represented as a set of values and each constraint propagator is polynomial. Each constraint $C(X_1, \ldots, X_k)$ can be invoked at most $\sum_{i=1}^{k} D(X_i)$ times and a propagator takes a polynomial time.

2.4.3 Global constraints

A *global constraint* is a non-fixed arity constraint. An example of a global constraint is the ALL-DIFFERENT$(X_1, X_2, \ldots, X_n)$ constraint that holds if its variables take all distinct values. Most global constraints have dedicated filtering algorithms that performs shrinking of domains during the inference phase of the search (Section 2.4.1).

Time complexity down a branch of a search tree. Constraint solvers usually enforce a consistency level Φ for a global constraint after its variable domains change. Hence, a filtering algorithm for a constraint $C(\mathbf{X})$ can be invoked $O(|\mathbf{D}(\mathbf{X})|)$ times down a branch of the search tree, where $|\mathbf{D}(\mathbf{X})| = \sum_{X \in \mathbf{X}} |D(X)|$. Note that a constraint propagator is invoked on monotonically decreasing domains of variables in scopes of corresponding constraints down a branch. This monotonicity property allows us to build incremental propagators that reuse information computed higher in the search tree.

Incremental propagators for global constraints are usually significantly faster compared to non-incremental propagators. In particular, this is the case for domain consistency propagators for ALL-DIFFERENT, GCC and SEQUENCE constraints. However, if we compare the time complexity per single invocation of incremental and non-incremental propagators, we obtain the same worst case complexity estimate. Consider for example the ALL-DIFFERENT$[X_1, \ldots, X_n]$ constraint. The domain consistency propagator for this constraint is based on finding a maximum matching in the variable-value graph. If a constraint propagator is not incremental then it has to construct a graph and check whether a maximum matching exists at each invocation. Hence, its single computation costs $O(nd\sqrt{n})$ time, where d is the total number of values in the variable domains. The time complexity down a branch is $O(nd \times nd\sqrt{|n|})$ for this propagator. An incremental propagator for the ALL-DIFFERENT constraint also constructs a variable-value graph and checks the existence of a matching during the first invocation. Hence, the worst case time complexity of its invocation is the same as the time complexity of the non-incremental propagator. However, the constructed matching will be incrementally updated during the following invocations. Hence, any subsequent call takes $O(d)$ time. The time complexity down a branch is $O(nd\sqrt{|n|} + nd \times d)$ which is much more efficient than the time complexity of non-incremental propagator down a branch. For this reason, it is meaningful to compute the total cumulative cost of enforcing Φ consistency down an *entire branch of the search tree* so as to capture the incremental cost of propagation rather than the complexity of a single invocation of a constraint propagator. We define the *total cumulative cost* of enforcing Φ consistency down an entire branch of the search tree as $\sum_{i=1}^{k} Tf_{\Phi}^{i}(\mathbf{D}^{i}(\mathbf{X}), S^{i-1})$ for any

sequence $(\mathbf{D}^1(\mathbf{X}), \ldots, \mathbf{D}^k(\mathbf{X}))$, $\mathbf{D}^i(\mathbf{X}) \supseteq f_\Phi^i(\mathbf{D}^i(\mathbf{X})) \supset \mathbf{D}^{i+1}(\mathbf{X})$, $i = 1, \ldots, k$, where k is the number of invocations, S^{i-1} is the internal state of a propagator before the ith invocation of the propagator, $\mathbf{D}^i(\mathbf{X})$ are the domains of variables $\mathbf{X}$ and $Tf_\Phi^i(\mathbf{D}^i(\mathbf{X}), S^{i-1})$ is the time complexity of executing f_Φ on domains $\mathbf{D}^i(\mathbf{X})$ at ith invocation of the propagator given S^{i-1}.

In this thesis, we consider the complexity of algorithms that enforce ϕ down a branch. In order to make an accurate comparison of incremental and non-incremental propagators, we compute the complexity of non-incremental propagators down a branch using the total cumulative cost.

Next we discuss the relation between variable domain representations and the time complexity of constraint propagators. As we pointed out in Section 2.4.1 there are two common integer variables representations: set representation and bounds representation. The set representation requires a linear number of bits in the number of possible values to represent the domain. It can represent any variable domain state and is used for constraint propagators that enforce domain consistency and range consistency. The bounds representation requires a logarithmic number of bits in the number of values to represent the domain. It can represent only a quadratic number of different states of a variable domain and is used for constraint propagators that enforce bounds consistency. It should be noted that the bound representation is exponentially more succinct compared to the domain representation. This potentially allows us to construct a bounds consistency propagator for a constraint that is exponentially faster compared to a domain consistency propagator for this constraint. For example, a bounds consistency propagator for the ALL-DIFFERENT constraint runs in $O(n)$ time while a trivial lower bound on the domain consistency propagator is $O(nd)$. However, this advantage of the bound representation disappears of we consider complexity of a propagator down a branch of the search tree.

For example, if a constraint solver prunes bounds of a variable by one value at a time then it ends up traversing the entire range of values in a domain as the following example shows.

Example 2.1 *Consider a set of constraints* $(X < Y)$, $(Y < X)$ *and* ALL-DIFFERENT$(X, Y, \mathbf{Z})$, *with* $D(X) = D(Y) = D(Z_i) = [0, 2^n]$, $i = 1, \ldots, n$. *We assume that the solver enforces bounds consistency on each constraint. We also assume that it invokes constraint on a variable bound change and that all constraints have the same priority. Reasoning about this decomposition takes an exponential time in most modern constraint solvers because finding a common fixpoint for this set of constraints takes* $O(2^n)$

Table 2.1: Runtimes in seconds to detect inconsistency in Example 2.1 by four modern solvers.

n	ILOG Solver [Ilo03]	GeCode [GeC11b]	Mistral [Heb11]	Choco [Cho11]
20	0.19	0.06	0.030	0.56
21	0.37	0.122	0.066	1.11
22	0.74	0.240	0.13	2.25
23	1.51	0.482	0.27	4.20
24	2.95	1.016	0.54	8.78
25	6.01	2.072	1.1	17.8

time. Table 2.1 shows the results of our experiments to solve this problem using several modern constraint solvers.

Note that the bounds consistency propagator for ALL-DIFFERENT$(X, Y, \mathbf{Z})$ *is called by the solver* $O(2^n)$ *times. Therefore, the time complexity of enforcing bounds consistency on the* ALL-DIFFERENT *constraint is* $O(n^2 2^n)$ *down a branch of the search tree.* $\diamond$

Example 2.1 demonstrates that using the bounds domain representation can 'hide' the complexity of the set representation. In the worst case a constraint solver traverses the entire domains of variables with bounds representation by inspecting one value at the time down a branch of a search tree. Note that Example 2.1 demonstrates a worst case behaviour of a solver, while it does not prove that it takes an exponential time to find a common fixpoint.

Unfortunately, the slow convergence of a constraint solver cannot be improved by using a smarter algorithms to find the common fixpoint. It was shown in [BKNV11] that it is NP-complete to find the common bounds consistent fixpoint for a polynomial set of linear inequality constraints even with just two variables per inequality. Therefore, the number of domain values is inevitably a factor in the worst case if we compute complexity down a branch of the search tree. Hence, as we compute complexity results down a branch of the search tree, bounds representation does not give us any computational worst case complexity advantage compared to the sets representation.

Fixed arity constraints. We defined a global constraint as a non-fixed arity constraint that describes any relation on a set of variables. Other constraints are usually called *fixed arity* or *bounded arity* constraints. Examples of such constraints are arithmetic constrains, like $X = Y + Z$, logical constraints, like $X \vee Y$, or the TABLE$(X_1, \ldots, X_k)$ constraint for a fixed k. Note that any fixed arity constraints can be represented as a TABLE constraint

by enumerating all allowed tuples. If variable domains are represented as a set of values then enforcing domain consistency on a table constraint is trivially polynomial. Another important property of fixed arity constraints is that they can be encoded in *SAT* so that unit propagation (which we discuss in the next section) achieves domain consistency [BHW03]. All decompositions into primitive constraints that we propose in this work are decompositions into bounded arity constraints or constraints that can be decomposed into bounded arity constraints without hindering propagation, like linear arithmetic constraints with unary coefficient.

Linear arithmetic constraints with unary coefficients. Finally, we consider the linear arithmetic constraint, $\sum_{i=1}^{n} X_i \leq b$. This constraint is a unbounded arity constraint. However, it can be decomposed into a set of fixed arity constraints without hindering propagation in the following way.

We split a linear constraint of the form $X_1+X_2+,\ldots,+X_{n-2}+X_{n-1}+X_n$ into ternary constraints as follows. We introduce an set of auxiliary variables, Y_j, $j = 1,\ldots,n-1$ and a set of ternary constraints so that $Y_1 = X_1 + X_2$, $Y_2 = Y_1 + X_3$, ..., $Y_{n-1} = Y_{n-2} + X_n$ and $Y_{n-1} \leq b$. This decomposition is Berge-acyclic and does not hinder propagation.

2.4.4 Boolean satisfiability

The Boolean satisfiability problem (*SAT*) is a special case of the *CSP* where variables are Boolean. For each Boolean variable x_i there exist two *literals* x_i and $\overline{x_i}$. Constraints in conjunctive normal form (CNF) are disjunctions of literals, called *clauses* and sometimes written simply as tuples of literals.

Unit propagation *forces* a literal to TRUE if it appears in a clause where all other literals are FALSE and continues until a fixpoint is reached. If all literals in a clause are made FALSE, we say that the empty clause is produced. A stronger form of inference is the *failed literal test* [Fre95] which is the same procedure as described in Section 2.4.1. The only difference is that there are only two values to test. For each literal l of an unset variable x, the failed literal test sets l to TRUE, performs unit propagation, checks whether the empty clause was produced and retracts l and its consequences. If the empty clause was produced, l is set to FALSE.

A *CSP* instance can be encoded as a *SAT* instance. The most widely used mapping of *CSP* variables to Boolean variables is the *direct encoding* . Each *CSP* variable X_i with domain $D(X_i)$ is encoded in *SAT* as a set of propositions $x_{i,j}$, $X_i \in \mathbf{X}$, $j \in D(X_i)$ such that $X_i \neq j \iff \overline{x}_{i,j}$. The property that each *CSP* variable has at most one value is enforced

by the set of clauses $(\overline{x}_{i,j}, \overline{x}_{i,k})$ for all $k \in D(X_i), k \neq j$ and the property that each *CSP* variable has at least one value is enforced by the set of clauses $\bigvee_{j \in D(X_i)} x_{i,j}$. We denote this propositional representation of $\mathbf{D}(\mathbf{X})$ as $\mathbf{D}^{sat}(\mathbf{X})$.

Another commonly used encoding of *CSP* variable domains in to *SAT* is *linear encoding*, which uses only a linear number of clauses. It should be pointed out that this encoding was independently proposed in [GN04] (*the ladder encoding*), [AM04] (*the regular encoding*), [TTKB06] (*the ordered encoding*) and [OSC07] (*the linear encoding*). We use the name *linear encoding* because it reflects the main advantage of this encoding – the number of clauses is linear in this encoding.

Each *CSP* variable X_i with domain $D(X_i)$ is encoded in *SAT* as a set of propositions $x_{i,j}$ as in the direct encoding and a set of propositions $y_{i,j}$ $X_i \in \mathbf{X}, j \in D(X_i)$ such that $X_i \leq j \iff y_{i,j}$. The following set of clauses ensures that it is a proper encoding of domains:

$$\overline{y}_{i,j} \vee y_{i,j+1} \quad lb(D(X_i)) \leq j < ub(D(X_i)) - 1 \tag{2.23}$$

$$\overline{x}_{i,j} \vee y_{i,j} \quad lb(D(X_i)) \leq j < ub(D(X_i)) \tag{2.24}$$

$$\overline{x}_{i,j} \vee \overline{y}_{i,j-1} \quad lb(D(X_i)) < j \leq ub(D(X_i)) \tag{2.25}$$

$$x_{i,lb(D(X_i))} \vee \overline{y}_{i,lb(D(X_i))} \tag{2.26}$$

$$x_{i,j} \vee \overline{y}_{i,j} \vee y_{i,j-1} \quad lb(D(X_i)) < j < ub(D(X_i)) \tag{2.27}$$

$$x_{i,ub(D(X_i))} \vee y_{i,ub(D(X_i)-1)} \tag{2.28}$$

In this work we use both direct and linear encodings in our decompositions.

Chapter 3

Decompositions of global constraints

In this chapter we recall the notion of constraint decomposition into a constraint language. Our definitions are based on the theoretical framework of Bessiere and Hentenryck [BH03]. We adjust this framework to handle decompositions of constraint propagators. We also introduce our assumptions about the size of variable domains and constraints representations. Note that our constraint reformulations into network flows naturally fit into this framework if we fix the constraint language to a single network flow constraint.

Global constraints are one of the key components of the success of constraint programming [HK06]. Their expressiveness allows capturing the structure of the problem at the modelling stage and their powerful propagation algorithms permit efficient reasoning during the search. To achieve the most efficient propagation, filtering algorithms are designed for each constraint individually. Moreover, some constraints have several propagation algorithms that achieve different consistency levels. The variety of filtering algorithms for global constraints allows us to achieve the best performance for each problem. However, it is hard for constraint solvers developers to keep up with all the recent developments in global constraints and their propagation algorithms. Modern constraint solvers often provide only a small set of global constraints that appear in the global constraints catalogue [BCR05] as implementing all existing constraints and filtering algorithms is a very time consuming task.

There are three ways for a constraint solver to provide support for a global constraint without implementing a dedicated propagator for this constraint. The first approach is to provide a constraint only at the modelling language level. This constraint is logically decomposed into a set of primitive or basic constraints that are available in every constraint solver, e.g. the ALL-DIFFERENT constraint can be decomposed into a set of binary inequality constraints. The logical constraint decomposition usually does not provide any guarantees on the consistency level that will be achieved by the decomposition. Therefore, the

user can use the global constraint to formalise his problem but cannot specify a propagation algorithm for this constraint.

The second approach is to provide full support for the global constraint by decomposing it into a set of primitive constraints in such a way that primitive constraints achieve the same amount of propagation as a specialised filtering algorithm for the original constraint at their fixpoint. For instance, the AtMostSeq constraint can be decomposed into a set of AtMost constraints without hindering domain consistency of propagation. In this case, a user specifies a global constraint and a consistency level that they want to achieve. Transparently to the user, this constraint is reformulated into a set of constraints that achieve the same inference as the propagation algorithm would do. Both these techniques are widely used in constraint solvers and modelling languages, such as the MiniZinc modelling language [G1211], Ilog [ILO05] or Sugar [Sug11] constraint solvers.

The third approach is to ask the user to enumerate the set of allowed solutions of the global constraint as a truth table, so-called the Table constraint. However, the size of the table might be exponential, $O(d^n)$, where d is the number of variable values and n is the number of variables. This makes this approach infeasible; therefore we do not consider it further in this work.

In this chapter we formalise the first two approaches. Historically, the first approach was more popular in constraint programming [Lau78]. The notion of reformulation of a global constraint as a logical decomposition into constraints from a given language was formalised in the theoretical work of Bessiere and Van Hentenryck [BH03]. To investigate the decomposability property of a constraint the authors formalised key notions in constraint logical reformulation: a constraint language and constraint rewriting scheme. The constraint language specifies the set of constraints and variable domains that can be used in decomposition. The constraint rewriting scheme defines a decomposition of a constraint into a set of constraints from the constraint language. Following this work, we also define a notion of a constraint language and a logical decomposition of constraint into a constraint language. However, we significantly narrow down the class of constraint languages to a set of constraints that are practically used in constraint reformulation.

The second direction starts from the work by Ian Gent on decomposing binary constraints [Gen02]. Gent used a set of clauses to decompose a binary constraint and proved that unit propagation on these clauses achieves domain consistency on the binary constraint. Subsequent research in this direction was focused on decompositions of n-ary constraints that preserve a consistency level. However, it is difficult to come up with a general way

to construct a polynomial size decomposition even into a set of clauses like in the case of binary constraints. Hence, a few special cases have been investigated. A number of decompositions are based on results by Dechter [Dec03]. For instance, a global constraint might be decomposed into a Berge acyclic graphs of simpler constraints, in a way that does not hinder propagation. This approach was for example used to obtain a ternary decomposition of the REGULAR constraint . Other results use the monotonicity property of some constraints, such as the ATMOSTSEQ or ATLEASTSEQ constraints [BHH+06b].

In the general case, decompositions for global constraints are constructed for each constraint individually. In this work we propose several new reformulations of global constraints that are not based on principles of Berge-acyclicity and monotonicity and do not hinder propagation. The idea of our approach is to decompose a global constraint into a set of fixed arity arithmetic constraints that mimics the corresponding filtering algorithm for this constraint. To formalise these reformulation techniques we introduce the notion of a constraint propagator language and a decomposition of a constraint propagator in a constraint propagator language. These notions generalise previous work on logical decomposition of constraints and allow us to capture the way of constructing constraint reformulation commonly used in modern constraint solvers. This problem was partially addressed in the work by Bessiere and Hentenryck [BH03]. However, their framework is not flexible enough to describe constraint propagator reformulations that we use in this work. In our decompositions, we allow different propagation algorithms for each constraint in a language. Moreover, in this work we investigate decomposability properties of constraints propagators rather than constraints themselves, so we extend their definitions.

3.1 Basic assumptions

In this section we define our main assumptions about representation of variable domains and constraints.

First we choose a variable domain representation that we use in this work. There exist two common representations of finite domain integer variables. The first representation is the set representation of variable domains. The second is the interval (or bounds) representation of variable domains (Section 2.4.1). Both of these representations are used in modern constraint solvers and the bound representation is exponentially more succinct compared to the domain representation. However, as we discussed in Section 2.4.3, if we compute complexity results down a branch of the search tree, bounds representation does not give us any computational worst case complexity advantage compared to the sets representation.

Hence, we make an assumption about domain representation:

Assumption 3.1 (Domains) *The domain of a variable is represented as a* ***set of integer values****.*

Assumption 3.1 implies that the size of a domain representation is linear in the number of possible values.

Next, we restrict the class of constraints that we consider in this work. We exclude constraints that require an exponential time in the size of variable domain representation to read a constraint or to check a solution. An example of such constraints are table constraints over unbounded number of variables. We also rule out constraints that are NP-complete to find a solution, as we can encode any *CSP* with a single constraint like this. These two restrictions lead us to the second assumption.

Assumption 3.2 (Constraints) *We consider only a constraint $C(X)$ and a consistency level Φ such that*

- *the size of the constraint is polynomial in the size of domain representation of X.*
- *there exists a constraint checker that detects Φ-disentailment and that has a polynomial time and space complexity in the size of the domain representation of X.*

Note that the existence of a polynomial time consistency checker guarantees the existence of a polynomial time propagator (Section 2.4.1).

Assumptions 3.1 and 3.2 hold in the rest of the work.

3.2 Logical decomposition of a constraint

We define a logically equivalent reformulation of a constraint into a set of constraints from a constraint language. We start from a definition of a constraint language that we take from [BH03]. We rewrite this definition to take into account Assumptions 3.1 and 3.2.

Definition 3.1 (Constraint Language(modified [BH03])) *A constraint language $\mathcal{L}$ is an infinite set of constraints $\{C_i\}$ that satisfy Assumption 3.2 over finite domain variables whose domains are represented as sets of integers.*

We introduce the notion of a logical decomposition of a constraint into a constraint language which is again a simplification of the language rewriting scheme that takes into account Assumptions 3.1 and 3.2.

Definition 3.2 (Projection of solutions) *Let $C(\mathbf{X}, \mathbf{Y})$ be a global constraint over variables $\mathbf{X}$ and $\mathbf{Y}$ and $sol(\mathbf{X}, \mathbf{Y})$ be the set of solutions of the constraint. The projection of solutions of C, $sols(C(\mathbf{X}, \mathbf{Y}))$, to variables $\mathbf{X}$, is a set of solutions where values of variables $\mathbf{Y}$ are ignored. We denote such a projection $sols(C(\mathbf{X}, \mathbf{Y}))[\mathbf{X}]$.*

Definition 3.3 (Logical decomposition of a global constraint) *Let $C(\mathbf{X})$ be a global constraint and $\mathcal{L}$ be a constraint language. Let $\mathbf{Y}$ be a set of auxiliary variables. A set of constraints $\mathcal{C} = \{C_i(\mathbf{X}, \mathbf{Y})\}$,$i = 1, \ldots, m$, $\mathcal{C} \subseteq \mathcal{L}$, is a* logical decomposition *of the global constraint $C(\mathbf{X})$ if and only if $sols(C(\mathbf{X})) = sols(\bigwedge_{i=1}^{m} C_i(\mathbf{X}, \mathbf{Y}))[\mathbf{X}]$.*

Note if $|D(\mathbf{Y})|$ and m are polynomial in $|D(\mathbf{X})|$ then together with Assumption 3.2 this guarantees that the size of the decomposition, $\sum_{i=1}^{m} |C_i(\mathbf{X}, \mathbf{Y})|$ is polynomial in the size of $D(\mathbf{X})$.

Definition 3.3 represents the main idea behind logical constraint reformulation. Instead of using a global constraint we replace it with a set of constraints from a given constraint language that are logically equivalent. Note that the constraints in the decomposition have original or possibly auxiliary variables in their scopes. This is essential for some constraints as they cannot be decomposed without using some auxiliary variables. For example, the parity constraint cannot be polynomially expressed in the language of clauses without introducing extra variables [DM02]. Moreover, introducing new variables can increase the strength of propagation that we can achieve. For example, to obtain a decomposition of the REGULAR constraint into a language of ternary table constraints, we introduce extra variables to obtain a decomposition that achieves domain consistency. To the best of our knowledge, there is no decomposition of the REGULAR constraint into this language that does not require extra variables. Note that if our language is the language of clauses over Boolean variables then we can logically decompose any global constraint in this language as we can reduce any *CSP* to $3SAT$.

As we pointed out above, a logically equivalent decomposition of a global constraint is an important concept if we want to rewrite a constraint. However, in practice, we are interested in decompositions of global constraints that achieve a certain consistency level on the original constraint C. Most of the previous work on constraint decomposition consider decompositions that provide guarantees with respect to a given consistency level [QW07, Gen02, BHW03]. Therefore, we extend the notion of constraint language and constraint decomposition to include these type of decompositions.

3.3 Decomposition of a constraint propagator

In this section we formalise the notion of decomposition of *a constraint propagator* as opposed to decomposition of a constraint. The idea of decomposing a constraint propagator is similar to the idea of logical decomposition of constraints. Informally, instead of achieving consistency level Φ on a global constraint $C(\mathbf{X})$ we achieve consistency level Φ_i for constraints C_i, $i = 1, \ldots, m$. If each of the constraints C_i, $i = 1, \ldots, m$ is Φ_i-consistent we are guarantied to achieve the same amount of pruning on variables $\mathbf{X}$ as we would get by enforcing Φ on C.

We introduce the notion of a constraint propagator language which is an extension of the notion of a constraint language.

Definition 3.4 (Constraint Propagator Language) *A constraint propagators language $\mathcal{LP}$ is an infinite set of pairs $\{\langle C_i(\mathbf{X}), P_{\Phi_i}\rangle\}$, where $C_i(\mathbf{X})$ is a constraint that satisfies Assumption 3.2 and P_{Φ_i} is a filtering algorithm that enforces consistency level Φ_i on C_i.*

Using the notion of constraint propagator language we introduce a notion of a decomposition of a global constraint propagator.

Definition 3.5 (Decomposition of a global constraint propagator) *Let $C(\mathbf{X})$ be a global constraint and P_Φ be a propagator that enforces consistency level Φ on C. A set of pairs $\{\langle C_i(\mathbf{X}, \mathbf{Y}), P_{\Phi_i}\rangle\} \subseteq \mathcal{LP}$, $i = 1, \ldots, m$ is* a Φ-decomposition *of the constraint $C(\mathbf{X})$ if and only if C_i is Φ_i-consistent, $i = 1, \ldots, m$ implies that C is Φ-consistent.*

Note that if $|D(\mathbf{Y})|$ and m are polynomial in $|D(\mathbf{X})|$ then together with Assumption 3.2 this guarantees that enforcing P_{Φ_i} on $C_i(\mathbf{X}, \mathbf{Y})$ is polynomial in time and space in $|D(\mathbf{X})|$. On top of this, Assumption 3.1 ensures that finding a common fixpoint for the set of constraints in the decomposition is polynomial in $|D(\mathbf{X})|$, e.g. [BKNV11].

Note that for our theoretical results we assume that auxiliary variables $\mathbf{Y}$ in the scope of the constraint in the decomposition $C_i(\mathbf{X}, \mathbf{Y})$, $i = 1, \ldots, m$, can only be modified by the corresponding propagator P_{Φ_i}. This means that we assume that a constraint solver does not branch on these variables. The reason for this is that branching on auxiliary variables corresponds to changing a global constraint definition. Hence, this makes it non-trivial to derive any theoretical guaranties for our decomposition.

Consider, for example, the linear constraint C, $X_1 + X_2 + X_3 = 5$ with a bounds consistency propagator P_{BC}. The constraint propagator language consists of ternary sums constraints, like $Z_1 = Z_2 + Z_3$. We enforce bounds consistency on ternary sums constraints.

To decompose C, we introduce an auxiliary variable Y and obtain the following pairs of constraints and their propagators: $\langle X_1 + X_2 = Y, P_{BC}\rangle$ and $\langle Y + X_3 = 5, P_{BC}\rangle$. This decomposition does not hinder propagation as the constraint graph is Berge-acyclic. Suppose the constraint solver branches on variable Y, e.g $Y < 2$. In this case, the constraint solver implicitly enforces an additional constraint on the first two terms in the original constraint. In particular, it requires that $X_1 + X_2 < 1$. The decomposition becomes logically stronger compared to the original constraint. In this work we do not investigate *theoretical* consequences of branching or modifying auxiliary variables **Y** externally to the constraints in the decomposition by the constraint solver. Note that we only impose this restriction to simplify the derivation of theoretical results. In practice, the constraint solver can branch on these variables and use them for learning. This only makes the reformulation stronger and more useful.

It should be noted that Definition 3.5 is again inspired by definitions of strong globalities in a constraint language that were introduced in [BH03]. Formally, the main difference is that Definition 3.5 allows enforcing any consistency level Φ_i on constraint C_i in a decomposition rather than enforcing consistency level Φ on all constraints. Conceptually, we make the constraint propagator of the constraint rather than the constraint C itself the subject of decomposition.

3.4 CNF Decomposition of a constraint propagator

Finally, we focus on a restricted but very useful subclass of decompositions of constraint propagators – *CNF* decompositions. Definition 3.5 already provides the notion of a decomposition of a constraint propagator. However, we need to adjust this definition as we have to encode integer domain variables into Boolean variables. In this work we use the direct encoding (Section 2.4.4) of integer variables into Boolean variables. There are a number of advantages in using this encoding. First, the direct encoding can represent arbitrary state of a variable domain during search. Second, the direct encoding gives easy access to the state of variable domains for other constraints in the decomposition. Third, we can easily perform a polynomial time and space transformation of the constraint programming decompositions of constraint propagators that we propose into *CNF* decompositions of constraint propagators. There exist two commonly used alternative encodings. The linear encoding of variable domains(Section 2.4.4), that can represent any interval domains but cannot represent domains with holes. This encoding is more restrictive compared to the direct encoding. An advantage of the linear encoding is that it requires a linear number of variables and

clauses while the direct encoding uses a quadratic number of clauses. An important similarity is that sizes of the direct and linear encodings are polynomial in the number of variables and their possible values. Another popular encoding is a logarithmic encoding of variable domains and its variations [Gav07]. These encodings are exponentially more succinct compared to the direct encoding as the size of these encodings are logarithmic in the number of possible variables values. The main disadvantage for us is that with a logarithmic number of Boolean variables we cannot represent all possible states of variables bounds during search [Gav07]. Hence, we cannot express bounds consistency propagators behaviour using these encodings.

Definition 3.6 (CNF Decomposition of a propagator) *A CNF decomposition of a propagation algorithm P_Φ for a global constraint $C(\mathbf{X})$ is a formula in CNF C_P over Boolean variables $\mathbf{x} \cup \mathbf{y}$ such that*

- input variables $\mathbf{x}$ *are the propositional representation of $D(\mathbf{X})$ using the direct encoding and $\mathbf{y}$ is a set of auxiliary variables whose size is polynomial in $|\mathbf{X}|$.*
- $x_{i,j}$ *is set to* FALSE *by unit propagation if and only if $X_i = j \notin P_\Phi(\mathbf{D}(\mathbf{X}))$.*
- *Unit propagation on C_P produces the empty clause when $P_\Phi(D(\mathbf{X})) = \emptyset$.*

Note that if $|y|$ and the number of clauses are polynomial in $|D(\mathbf{X})|$ then the CNF decomposition is polynomial in the size of $D(\mathbf{X})$.

Note that, we use only information about variables that are set to FALSE to obtain information about the current state of domains in Definition 3.6. The reason for this is that we want to ensure the monotonicity property of a propagator over Boolean variables x. In practice, a CNF decomposition of a propagator may depend on variables that are set to TRUE and have clauses that contain negative literals in a CNF decomposition. However, we can always obtain the same information using positive literals. So, we can modify clauses of the CNF decomposition substituting negative literals with the disjunction of positive literals. For instance, consider variable X_2 with domain $\{1, 2, 3\}$ and clause $(x_{1,1}, \overline{x}_{2,2}, \overline{y})$ in C_P. The literal $x_{2,2}$ can make this clause unit. The direct encoding of $D(X_2)$ includes a at least clause $(x_{2,1}, x_{2,2}, x_{2,3})$ and at most clauses $(\overline{x}_{2,1}, \overline{x}_{2,2})$, $(\overline{x}_{2,3}, \overline{x}_{2,2})$ and $(\overline{x}_{2,1}, \overline{x}_{2,3})$. Note that literal $x_{2,2}$ is TRUE if and only if literals $x_{2,1}$ and $x_{2,3}$ are FALSE. Therefore, the literal $\overline{x}_{2,2}$ can be replaced with the disjunction $(x_{2,1}, x_{2,3})$ and the clause $(x_{1,1}, \overline{x}_{2,2}, \overline{y})$ is transformed to the clause $(x_{1,1}, x_{2,1}, x_{2,3}, \overline{y})$.

Chapter 4

The SEQUENCE constraint

4.1 Introduction

In modelling real-world and combinatorial problems we often need to restrict the number of occurrences of some values in an interval. For example, in staff scheduling problems we may want to specify that an employee can have at most two days off during a week, in car sequencing problems we may want to constrain the number of cars with sun roof produced within a day or in job-shop scheduling problems we may want to restrict the number of machines using a resource at any time point. To encode this type of constraints the AMONG constraint was introduced [BC94]. AMONG states that between l and u of k variables take values in a given set S.

Definition 4.1 AMONG$(l, u, [X_1, \ldots, X_k], S)$ *holds if* $l \leq |\{i \mid X_i \in S\}| \leq u$.

In some problems we might need to express a generalisation of this constraint. For example, we want to specify that every employee has at least 2 days off in any 7 day period or at most 1 in 3 cars along the production line has a sun-roof. In this case we want to slide a restriction on the number of occurrences of some values along a sequence of variables. To express this restriction the SEQUENCE constraint was introduced. The SEQUENCE constraint restricts the number of values taken from a given set S in any sequence of k variables.

Definition 4.2 SEQUENCE$(l, u, k, [X_1, \ldots, X_n], S)$ *holds if for* $1 \leq i \leq n - k + 1$, AMONG$(l, u, [X_i, \ldots, X_{i+k-1}], S)$ *holds.*

The SEQUENCE constraint can be seen as a conjunction of sliding AMONG$(X_i, \ldots, X_{i+k-1})$ constraints. We say that a variable subsequence of length k is a *sliding window* and k is *the size* of the sliding window.

There are several interesting special cases and generalisations of the SEQUENCE constraint. The ATMOSTSEQ and ATLEASTSEQ constraints are two practically useful special cases of the SEQUENCE constraint. If $l = 0$, SEQUENCE is reduced to an ATMOSTSEQ constraint. If $u = k$, SEQUENCE is an ATLEASTSEQ constraint. These constraints occur for example in modelling car sequencing as such problems typically only place upper bounds on occurrences (e.g. at most 1 in 3 cars can have the sun-roof). An interesting property of the ATLEASTSEQ constraint is that its decomposition into a set of ATLEAST constraints does not hinder propagation [BHH+06b]. The same property holds for the ATMOSTSEQ constraint.

To model some problems, we may want to have windows with different sizes or positions of constrained variables instead of sliding a window of fixed size. For example, the window size in a rostering problem may depend on whether it includes a weekend or not. As a second example, we might not want a window to start on Sunday. An extension of the SEQUENCE constraint proposed in [HPRS] is that each AMONG constraint can have different parameters (start position, l, u, and k). More precisely,

Definition 4.3 GEN-SEQUENCE$(\vec{p_1}, \ldots, \vec{p_m}, [X_1, X_2, \ldots, X_n], S)$ *holds if* AMONG$(l_i, u_i, k_i, [X_{s_i}, \ldots, X_{s_i+k_i-1}], S)$ *for* $1 \leq i \leq m$ *where* $\vec{p_i} = \langle l_i, u_i, k_i, s_i \rangle$.

In modelling over-constrained problems it is useful to have a soft form of the SEQUENCE constraint. The ROADEF 2005 challenge [SCNA08], which was proposed and sponsored by Renault, puts forward a violation measure for the SEQUENCE constraint which takes into account by how much each AMONG constraint is violated. We therefore consider the soft global SEQUENCE constraint that introduces a violation variable T and bounds the total number of violations of individual AMONG constraints:

Definition 4.4 SOFTSEQUENCE$(l, u, k, T, [X_1, \ldots, X_n], S)$ *holds if*

$$T \geq \sum_{i=1}^{n-k+1} \max(l - \sum_{j=0}^{k-1} X_{i+j} \in S, \sum_{j=0}^{k-1} X_{i+j} \in S - u, 0) \tag{4.1}$$

The SLIDINGSUM constraint [BCR05] is another generalisation of the SEQUENCE constraint that restricts the sum of variables rather than the number of occurrences of values in a set S. We further extend the constraint to allow arbitrary windows .

Definition 4.5 SLIDINGSUM $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}])$ *holds if* $l_i \leq \sum_{j=s_i}^{s_i+k_i-1} X_i \leq u_i$ *holds for* $1 \leq i \leq n - k + 1$ *where* $\vec{p_i} = \langle l_i, u_i, k_i, s_i \rangle$.

Similarly to the soft SEQUENCE constraint we can define the soft SLIDINGSUM $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}], T)$ constraint:

Definition 4.6 SLIDINGSUM $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}])$ *holds if* $T \geq \sum_{i=1}^{m} \max(l_i - \sum_{j=s_i}^{s_i+k_i-1} X_j, \sum_{j=s_i}^{s_i+k_i-1} X_j - u_i, 0)$

Finally, we consider the Multiple SEQUENCE constraint, where we have multiple SEQUENCE constraints applied to the same sequence of variables. For instance, we might insist that at most 1 in 3 cars have the sun roof option and simultaneously that at most 2 in 5 of those cars have electric windows.

Definition 4.7 *The Multiple* SEQUENCE $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}])$ *holds if* SEQUENCE$(l_i, u_i, k_i, [X_1, \ldots, X_n], S_i)$ *holds for* $1 \leq i \leq n - k + 1$ *where* $\vec{p_i} = \langle l_i, u_i, k_i, s_i, S_i \rangle$.

The AMONG and SEQUENCE constraints are defined over integer domain variables. However, they can be encoded by channelling into Boolean variables. We use $Y_i \leftrightarrow (X_i \in S)$ and $l \leq \sum_{i=1}^{k} Y_i \leq u$ to do the encoding. Since the constraint graph of this encoding is Berge-acyclic, this does not hinder propagation. Consequently, we will simplify notation and consider AMONG, SEQUENCE, GEN-SEQUENCE, the Soft SEQUENCE on Boolean variables and fix the set S to contain the single value $\{1\}$ in these cases.

In this chapter we make the following contributions:

- propose new domain consistency algorithms for the SEQUENCE constraint (Section 4.2.3).
- propose new domain consistency algorithms for the GEN-SEQUENCE constraint based on all pairs shortest path algorithm (Section 4.2.4 and Section 4.4.1).
- propose a first polynomial domain consistency algorithm for SLIDINGSUM (Section 4.4.4).
- propose a first polynomial domain consistency algorithm for the soft SEQUENCE constraint and the soft GEN-SEQUENCE constraint (Section 4.4.2 and Section 4.4.3).
- introduce and propose a first polynomial domain consistency algorithm for the soft SLIDINGSUM constraint (Section 4.4.4).

- propose several encodings of the SEQUENCE constraint into a set of primitive constraints and theoretically analyse their effectiveness and efficiency (Sections 4.3.2–4.3.6).

- experimentally evaluate proposed algorithms and encodings of the SEQUENCE constraint on some random and nurse scheduling problems. We identify conditions for applicability of each algorithm or encoding to these problems (Section 4.6).

4.2 Filtering algorithms for the SEQUENCE constraint

In this section we present several algorithms that enforce domain consistency on the SEQUENCE constraint. We start with two domain consistency algorithms that were proposed by van Hoeve *et al.* [HPRS]. The first algorithm enforces domain consistency in cubic time, while the second algorithm is exponential in the length of the sliding window. Next, we present two new domain consistency algorithms for the SEQUENCE constraint and the GEN-SEQUENCE constraint that improve upon the existing algorithms by a linear factor in both cases. Let us first introduce the running example.

Example 4.1 (Running example (SEQUENCE)) *Suppose regulations restrict the number of days off to be between* 1 *and* 2 *in any three consecutive days in a valid schedule. The scheduling period is six days. To encode this rule we introduce six Boolean variables,* $X_1, \ldots, X_6$*, one for each day in the schedule. If a worker has the* i*th day off then* $X_i = 1$*, otherwise* $X_i = 0$*.* SEQUENCE$(l, u, k, [X_1, \ldots, X_6])$*,* $l = 1$*,* $u = 2$ *and* $k = 3$ *encodes the regulation rule.* $\diamond$

4.2.1 Cumulative sums based algorithm ($HPRS$)

The first polynomial time domain consistency algorithm for the SEQUENCE constraint was proposed by van Hoeve *et al.* [HPRS]. The propagator performs the failed literal test (Section 2.4.1) using an algorithm (*the repair procedure*) that detects domain disentailment for the SEQUENCE constraint.

An array of cumulative sums y of length $n + 1$ is introduced, in order to detect disentailment of the constraint. A value y_i shows the number of ones taken by the first i variables X. The algorithm initially assigns values y as $y_i = \sum_{k=1}^{i} \min(D(X_k))$ for $i = 1$ to n and $y_0 = 0$. Then it keeps repairing the initial assignment of y's until they satisfy two sets of constraints:

$$y_{i+1} - y_i \in D(X_{i+1}), \quad i = 0, \ldots, n-1 \tag{4.2}$$

$$l \leq y_{i+k} - y_i \leq u, \quad i = 0, \ldots, n-k. \tag{4.3}$$

If the repair procedure finds an assignment to cumulative sums y that satisfies all constraints than the SEQUENCE constraint is satisfiable and the solution of the constraint can be obtained as

$$X_i = y_{i+1} - y_i, i = 0, \ldots, n. \tag{4.4}$$

It is easy to see that this assignment of variables X is a solution of the SEQUENCE constraint. This assignment is also the lexicographically smallest assignment of the variables y. The repair procedure runs in $O(n^2)$ time. The failed literal test adds another linear factor to the complexity resulting in $O(n^3)$ [HPRS]. The algorithm can be made incremental [HPRS09] so the total time complexity down a branch of the search tree is $O(n^3)$.

Example 4.2 *Consider how the algorithm works on the running example 4.1. Suppose we want to check whether the variable-value pair* $X_2 = 1$ *has a support. We fix* X_2 *to* 1 *and compute the values of cumulative sums* y_i*. We obtain that* $y_0 = 0$*,* $y_1 = \min(D(X_1)) = 0$*,* $y_2 = \min(D(X_1))+\min(D(X_2)) = 1$*,* $y_3 = \min(D(X_1))+\min(D(X_2))+\min(D(X_3)) = 1$ *and so on. Hence,* $y_0 = y_1 = 0$ *and* $y_2 = \ldots = y_6 = 1$*. This assignment does not satisfy constraints* (4.2)-(4.3)*. Therefore, the repair procedure finds the lexicographically smallest assignment to the values* y_i*. In this case,* y*'s assignment is* $0, 0, 0, 1, 1, 1, 2$*. The corresponding assignment of the variables* $X = [0, 0, 1, 0, 0, 1]$ *is indeed a support for* $X_2 = 1$*.* $\diamond$

4.2.2 REGULAR based algorithm (RE)

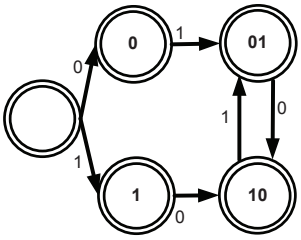
The second domain consistency algorithm for the SEQUENCE constraint was also proposed by van Hoeve *et al.* [HPRS]. The algorithm is based on the reformulation of the SEQUENCE constraint into the REGULAR constraint . To perform reformulation we need to construct an automaton $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$, where $\Sigma = \{0, 1\}$, that accepts strings that are exactly the solutions of the SEQUENCE constraint.

The states of $\mathcal{A}$ represent all the valid sequences of at most k values and the transitions between them simulate one shift to the right along the sequence. Hence, each state q, $q \in Q$ of the automaton $\mathcal{A}$ encodes a *valid* sequence of values, so that $q = \langle d_1, \ldots, d_p \rangle$, $p \leq k$ and $l \leq \sum_{j=1}^{p} d_j \leq u$. The initial state q_0 represents the empty sequence of values. Any state

is an accepting state, so that $Q = F$. A transition of the automaton from a state q to q' on seeing symbol v exists if and only if $q = \langle d_1, \ldots, d_k \rangle$ and $q' = \langle d_2, \ldots, d_k, v \rangle$, $v \in \{0, 1\}$.

The constructed automaton accepts only solutions of the SEQUENCE constraint. The time complexity of the algorithm is $O(n2^k)$ which is exponential in the size of the sliding window.

Figure 4.1 The REGULAR encoding of the SEQUENCE$(1, 1, 2, [X_1, \ldots, X_6])$ constraint.



Example 4.3 *Consider the encoding of the* SEQUENCE$(l, u, k, [X_1, \ldots, X_6])$, $l = 1$, $u = 1$, $k = 2$ *constraint into the* REGULAR *constraint. Automaton* $\mathcal{A}$ *that represents the* SEQUENCE *constraint is shown at Figure 4.1. For example, the initial state corresponds to the empty sequence of values and has two transitions to sequences of length* 1. *Note that* $\mathcal{A}$ *contains only valid states. Hence, there is no state that corresponds to the sequence* $\langle 11 \rangle$. $\diamond$

4.2.3 Network flow-based problem (primal model) (FB)

In this section we present a new filtering algorithm that enforces domain consistency on the SEQUENCE constraint first presented in [MNQW08]. This algorithm is based on a reformulation of the SEQUENCE constraint into a network flow problem. This reformulation allows us to use well studied results from graph theory and integer linear programming to construct an efficient propagator which is the fastest known propagator for the constraint to date for unbounded length of the sliding window.

Reformulation as an integer linear program. We encode the SEQUENCE constraint into a network flow by means of a linear program (LP). The SEQUENCE constraint can be seen as a conjunction of sliding AMONG constraints. In turn, the AMONG$(l, u, [X_1, \ldots, X_k])$ constraint over Boolean variables is logically equivalent to two linear inequalities:

$$l \leq \sum_{i=1}^{k} X_i$$

and

$$\sum_{i=1}^{k} X_i \leq u.$$

Therefore, we can reformulate the SEQUENCE constraint directly as an integer linear program. We use our running example 4.1 to demonstrate the reformulation:

$$\begin{aligned} l \leq \quad & X_1 + X_2 + X_3 \quad \leq u, \\ l \leq \quad & X_2 + X_3 + X_4 \quad \leq u, \\ l \leq \quad & X_3 + X_4 + X_5 \quad \leq u, \\ l \leq \quad & X_4 + X_5 + X_6 \quad \leq u \end{aligned}$$

where $X_i \in \{0, 1\}$.

By introducing surplus/slack variables, Y_i and Z_i, we convert this to a set of equalities in the standard from:

$$\begin{aligned} X_1 + X_2 + X_3 - Y_1 &= l, & X_1 + X_2 + X_3 + Z_1 &= u, \\ X_2 + X_3 + X_4 - Y_2 &= l, & X_2 + X_3 + X_4 + Z_2 &= u, \\ X_3 + X_4 + X_5 - Y_3 &= l, & X_3 + X_4 + X_5 + Z_3 &= u, \\ X_4 + X_5 + X_6 - Y_4 &= l, & X_4 + X_5 + X_6 + Z_3 &= u \end{aligned}$$

where $Y_i, Z_i \geq 0$. In matrix form, this is:

$$\begin{pmatrix} 1&1&1&0&0&0&-1&0&0&0&0&0&0&0 \\ 1&1&1&0&0&0&0&1&0&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&-1&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&0&1&0&0&0&0 \\ 0&0&1&1&1&0&0&0&0&-1&0&0&0&0 \\ 0&0&1&1&1&0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&1&1&0&0&0&0&0&0&-1&0 \\ 0&0&0&1&1&1&0&0&0&0&0&0&0&1 \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_6 \\ Y_1 \\ Z_1 \\ \vdots \\ Y_4 \\ Z_4 \end{pmatrix} = \begin{pmatrix} l \\ u \\ l \\ u \\ l \\ u \\ l \\ u \end{pmatrix}$$

This matrix has the *consecutive ones* property for columns: each column has a block of consecutive 1's or -1's and the remaining elements are 0's. This means that the matrix of coefficients is *totally unimodular* [Sch86] and, moreover, this *ILP* can be reformulated as a network flow problem in a graph by Theorem 2.5.

From integer linear program to a network flow problem. The consecutive ones property guarantees that there exists an equivalent network flow problem. To identify this problem we can use the method of Veinott and Wagner [WM62] (see Section 2.2.2). This method allows us to construct a graph such that feasible flows in this graph uniquely map to solutions of the SEQUENCE constraint. First we add a zero last row to the original matrix and subtract the ith row from $i + 1$th row for $i = 1$ to $2n$. These operations do not change the set of solutions. This gives:

$$A\vec{X} = \vec{b}, \tag{4.5}$$

where

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix},$$

$$\vec{X} = \Big(X_1, \ldots, X_6, Y_1, Z_1, \ldots, Y_4, Z_4\Big)^T,$$

$$\vec{b} = \Big(l, u-l, l-u, u-l, l-u, u-l, l-u, u-l, -u\Big)^T$$

This system of equations describes a network flow problem on graph $\mathrm{G} = (V, E)$ by Theorem 2.5. Each row in the matrix corresponds to a node in V and each column corresponds to an edge in E. Down each column, there is a single row i equal to 1 and a single row j equal to -1 corresponding to an edge $(i, j) \in E$ in the graph. We include a source node s and a sink node t in V. Let b be the vector on the right hand side of the equation. If b_i is positive, then there is an edge $(s, i) \in E$ that carries exactly b_i amount of flow. If b_i is negative, there is an edge $(i, t) \in E$ that caries exactly $|b_i|$ amount of flow. The bounds on the variables, which are not expressed in the matrix, are represented as bounds on the capacity of the corresponding edges.

Theorem 4.1 *Consider a* SEQUENCE$(l, u, k, [X_1, \ldots, X_n], S)$ *constraint. There exists an equivalent network flow graph* $G = (V, E)$ *such that there is a one-to-one correspondence between solutions of the constraint and feasible flows in the network. This graph consists of* $5n - 4k + 5$ *edges,* $2n - 2k + 5$ *vertices, a maximum edge capacity of* u*, and an amount of flow to send equal to* $f = (n - k)(u - l) + u$*.*

Proof: The existence of the network flow graph follows from the equivalence between solutions of *ILP* and solutions of the SEQUENCE constraint, the correctness of the Veinott-Wagner procedure and Theorem 2.5.

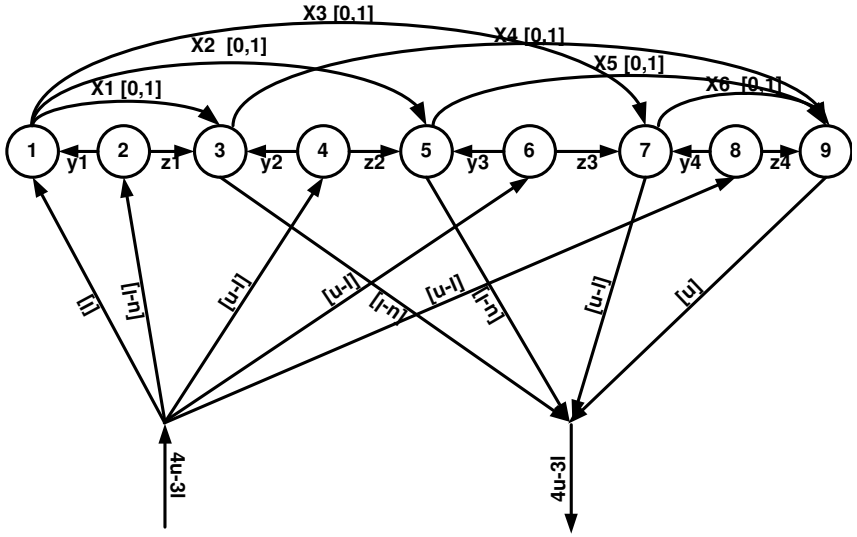
The number of vertices in the graph is equal to the number of rows in the matrix A which is $2(n - k + 1) + 1$ plus the source and the sink. This gives $2n - 2k + 5$ vertices. The number of edges equals the number of columns plus the number of edges that connect the source and the sink to other vertices. There are $n + 2(n - k + 1)$ of the former one and $2(n - k + 1)$ of the latter. By construction, the maximum edge capacity is u. The total amount of flow is

$$\sum_{i=1, b_i>0}^{n+2(n-k+1)} b_i = \sum_{i=1}^{(n-k+1)} (u - l) + l = (n - k)(u - l) - u.$$

$\diamond$

The time complexity of finding a maximum flow of value $v(f)$ is $O(|E|v(f))$ using the Ford-Fulkerson algorithm [CLRS01]. Faster algorithms exist for this problem. For example, Goldberg and Rao's algorithm finds a maximum flow in $O(min(|V|^{2/3}, |E|^{1/2})|E| \log(|V|^2/|E| + 2) \log C)$ time where C is the maximum capacity upper bound for an edge [GR98]. In our case, this gives $O(n^{3/2} \log n \log u)$ time complexity.

Figure 4.2 A flow graph for SEQUENCE$(l, u, 3, [X_1, \ldots, X_6])$



The graph for the set of equations in the example is given in Figure 4.2. A flow of value $4u - 3l$ in the graph corresponds to a solution. If a feasible flow sends a unit flow through the edge labelled with X_i then $X_i = 1$ in the solution; otherwise $X_i = 0$. Each even numbered vertex $2i$ represents a window. The way the incoming flow is shared between Y_j and Z_j reflects how many variables X_i in the j'th window are equal to 1. Odd numbered vertices represent transitions from one window to the next (except for the first and last vertices, which represent transitions between a window and nothing). An incoming X edge represents the variable omitted in the transition to the next window, while an outgoing X edge represents the added variable.

Theorem 4.1 suggests a straightforward algorithm to achieve domain consistency on the SEQUENCE constraint. We use Theorem 4.1 to construct an algorithm that detects domain disentailment for the constraint and run the failed literal test on top of this algorithm to achieve domain consistency. This gives *DC* propagator that runs in $O(n \times n^{3/2} \log n \log u)$. Next we show that this complexity can be significantly improved.

Enforcing domain consistency. Based on Theorem 4.1 we propose a network flow-based *DC* filtering algorithm for the SEQUENCE constraint. Algorithm 4.1 shows the pseu-

Algorithm 4.1 *DC* propagator for the SEQUENCE

```
1:  procedure PROPAGATORSEQUENCE(D(X_1), ..., D(X_n))
2:      Construct G(V, E).
3:      Find a feasible flow f.
4:      if ∄ a feasible flow f in G(V, E) then
5:          return Failure.
6:      Find the residual graph G_f = (V, E_f).
7:      Find the strongly connected components in G_f.
8:      for k ← 1 to n do
9:          Let e_{X_k} = (V_i, V_j) be the edge that corresponds to the variable X_k in G.
10:         if V_i and V_j are not in the same strongly connected component then
11:             if f(e_{X_k}) = 1 then
12:                 D(X_k) = D(X_k) \ {0}
13:             else
14:                 D(X_k) = D(X_k) \ {1}
15:     return True.
```

docode for this algorithm.

Theorem 4.2 *Algorithm 4.1 enforces domain consistency of the* SEQUENCE *constraint in* $O(n^{3/2} \log n \log u)$ *time.*

Proof: By Theorem 4.1 a support for value 1 (0) in $D(X_i)$, $i = 1, \ldots, n$ exists if and only if there exists a feasible flow that sends (does not send) a unit flow through the edge labelled with X_i.

A feasible flow f in the graph $G = (V, E)$ gives a support for one of values of each variable (line 4). Consider a variable X_i. If $f(e_{X_i})$ pushes a unit of flow through the edge e_{X_i} then the value 1 is supported otherwise the value 0 is supported. Suppose that $f(e_{X_i}) = 1$. (The case $f(e_{X_i}) = 0$ is similar). To find support for $X_i = 0$ we have to find a feasible flow f' such that $f'(e_{X_i}) = 0$. Hence, by Theorem 2.1, such flow exists if and only if we can find a path in the residual graph G_f from V_i to V_j. This path exists if and only if the vertices V_i and V_j belong to the same strongly connected component (line 10).

Complexity argument: A feasible flow can be found in $O(n^{3/2} \log n \log u)$ time (line 4). Strongly connected components can be found in $O(|E|) = O(n)$, because the number of edges in the flow graph for the SEQUENCE constraint is linear in n by Theorem 4.1 (line 7). The loop 8-14 takes $O(n)$ time. Hence, the total time complexity is $O(n^{3/2} \log n \log u)$. $\diamond$

Algorithm 4.2 Incremental *DC* propagator for the SEQUENCE

procedure PROPAGATORSEQUENCEINC($X_k = v, f, D(X_1), \ldots, D(X_n)$)

f is a feasible flow

X_k is the assigned variable

Let $e_{X_k} = (V_i, V_j)$ be the edge that corresponds to the variable X_k in G.

if $X_k = v$ and $f(e_{X_k}) = v$, $v \in \{0, 1\}$ **then**

return True.

else

if $X_k = 1$ and $\exists$ a path from V_i to V_j in $G_f = (V, E_f)$ **then**

Push a unit of flow through e_{X_k} and update f.

else

return Failure.

if $X_k = 0$ and $\exists$ a path from V_j to V_i in $G_f = (V, E_f)$ **then**

remove a unit of flow through e_{X_k} and update f.

else

return Failure.

Find the strongly connected components in G_f .

for $i \leftarrow 1$ **to** n **do**

Let $e_{X_i} = (V_i, V_j)$ be the edge that corresponds to the variable X_i in G.

if V_i and V_j are not the same strongly connected component **then**

if $f(e) = 1$ **then**

$D(X_i) = D(X_i) \setminus \{0\}$.

if $f(e) = 0$ **then**

$D(X_i) = D(X_i) \setminus \{1\}$.

Example 4.4 *Consider our running example. We assume that the first two variables are fixed to* 1, $X_1 = X_2 = 1$. *A feasible flow in the corresponding graph is shown in Figure 4.3 (dashed edges). The residual graph is presented in Figure 4.4. The residual graph contains two strongly connected components . The first component includes the first and the second vertex (gray vertices). The second component contains the remaining vertices (patterned with vertical lines). Note that the ends of the arc that corresponds to the variable* X_3 *belong to different connected components. Therefore, there is no way to construct a feasible flow through this edge to support the value* 1. *Hence,* 1 *is not supported for* X_3. *Indeed, if the first two variables are fixed to* 1, *the third variable has to take* 0 *to satisfy the first* AMONG$(1, 2, [X_1, X_2, X_3])$ *constraint.* $\diamond$

Figure 4.3 A flow graph for SEQUENCE$(1, 2, 3, [1, 1, X_3, X_4, X_5, X_6])$

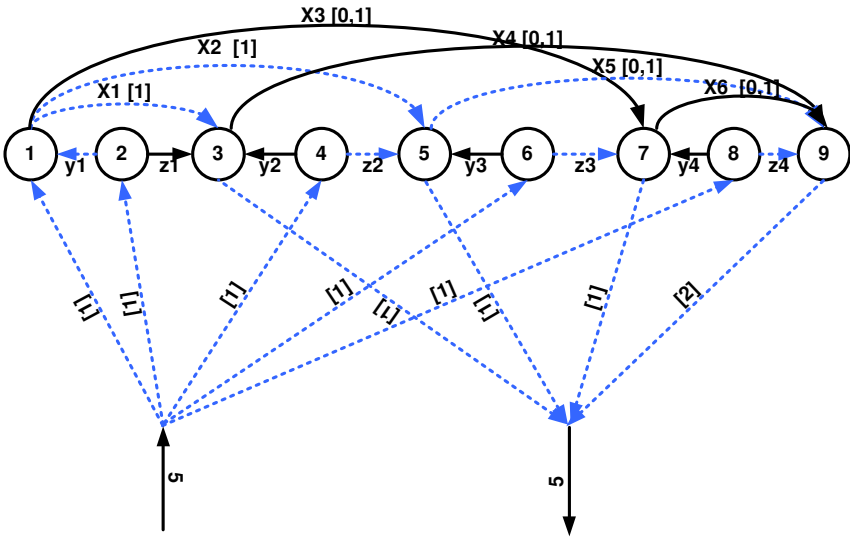
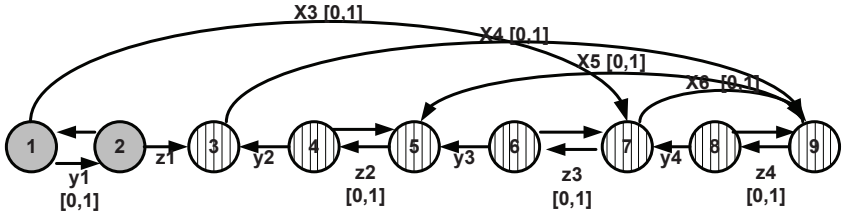
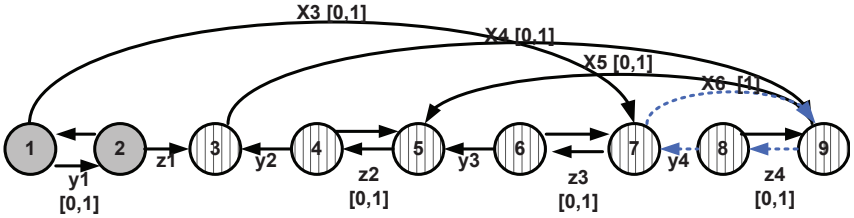


Figure 4.4 The residual graph for the flow at Figure 4.3



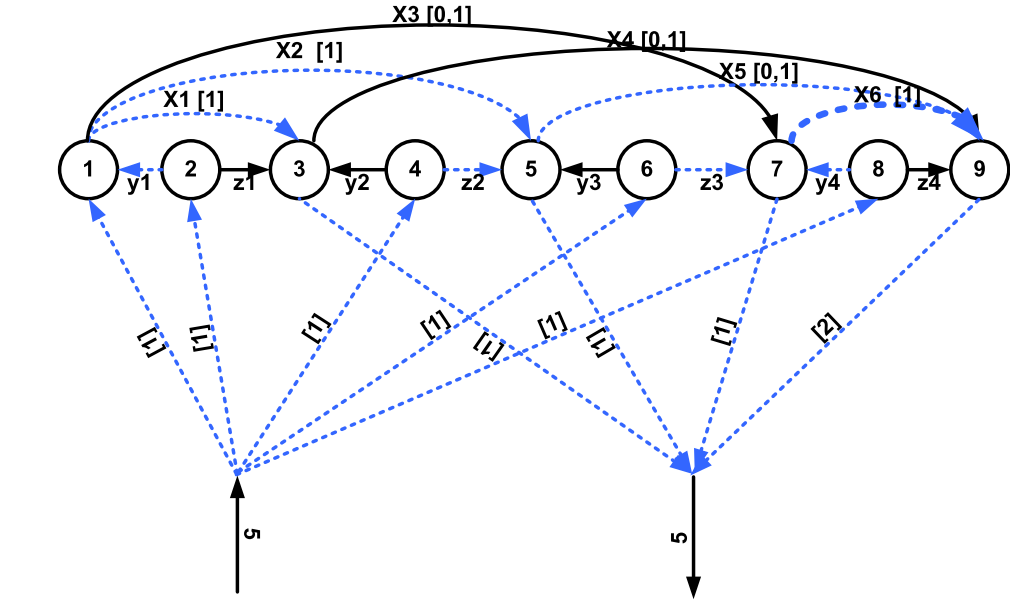
Example 4.5 *We continue our running example. Suppose that* X_6 *is assigned the value* 1*. Note that the feasible flow supports the value* 0 *from* $D(X_6)$*. Hence, we find a path in the residual graph from vertex* 9 *to vertex* 7*, which forms a cycle with* e_{X_6} *(Figure 4.5, dashed edges) and redirect the flow to construct a new feasible flow (Figure 4.6) that supports* 1*.* $\diamond$

Figure 4.5 Flow redirection in residual graph $G_f = (V, E_f)$.



Following Regin [Reg94, Reg96], we can make the algorithm incremental. Suppose that during search X_i is fixed to value v. If the last computed flow was a support for $X_i = v$, then there is no need to recompute the flow. We simply need to recompute the strongly connected components in the new residual graph and enforce *DC* in $O(n)$ time. If the last computed flow is not a support for $X_i = v$, we can find a cycle in the residual graph containing the edge associated with X_i in $O(n)$ time. By pushing a unit of flow over this

Figure 4.6 An updated network flow that supports $X_6 = 1$.



cycle, we obtain a flow that is a support for $X_i = v$. Enforcing domain consistency can be done in $O(n)$ after computing the strongly connected components. Consequently, there is an incremental cost of $O(n)$ when a variable is fixed, and the cost of enforcing *DC* down a branch of the search tree is $O(n^2)$. Algorithm 4.2 shows the pseudocode of the incremental algorithm.

4.2.4 Network flow-based problem (dual model) (DFB)

In the previous section we showed that the SEQUENCE constraint can be reformulated as an integer linear program (4.5), whose matrix A has the consecutive ones property in each column. The important point to observe here is that the consecutive ones property comes from the way we post the AMONG constraints: AMONG is shifted along the sequence of variables and the size of the sliding window is fixed. However, if we consider the GEN-SEQUENCE constraint , where the AMONG constraints can be posted over any sequence of variables, the corresponding *ILP* formulation of the problem might not have the consecutive ones property any more. Moreover, we can show that there is no equivalent transformation in general of the *ILP* model that can be reduced to a network flow problem as the following example shows.

Example 4.6 (Running example (GEN-SEQUENCE)) *Consider the* GEN-SEQUENCE $(l, u, k, [X_1, \ldots, X_5])$ *constraint which is formed by the conjunction of four* AMONG *constraints with windows: [1,5], [2,4], [3,5] and [1,3]. This constraint can be reformulated as the following ILP*

$$l \leq \quad X_1 + X_2 + X_3 \qquad\qquad \leq u \tag{4.6}$$

$$l \leq \qquad X_2 + X_3 + X_4 \qquad\quad \leq u, \tag{4.7}$$

$$l \leq \qquad\qquad X_3 + X_4 + X_5 \quad \leq u, \tag{4.8}$$

$$l \leq \quad X_1 + X_2 + X_3 + X_4 + X_5 \quad \leq u, \tag{4.9}$$

where $X_i \in \{0, 1\}$.

By introducing surplus/slack variables, Y_i *and* Z_i*, we convert this to a set of equalities in the standard from:*

$$\begin{aligned} X_1 + X_2 + X_3 - Y_1 &= l, & X_1 + X_2 + X_3 + Z_1 &= u, \\ X_2 + X_3 + X_4 - Y_2 &= l, & X_2 + X_3 + X_4 + Z_2 &= u, \\ X_3 + X_4 + X_5 - Y_3 &= l, & X_3 + X_4 + X_5 + Z_3 &= u, \\ X_1 + X_2 + X_3 + X_4 + X_5 - Y_4 &= l, & X_1 + X_2 + X_3 + X_4 + X_5 + Z_4 &= u \end{aligned}$$

where $Y_i, Z_i \geq 0$*. We can express it as an integer linear program in matrix form:*

$$\begin{pmatrix} 1&1&1&0&0&-1&0&0&0&0&0&0&0&0 \\ 1&1&1&0&0&0&1&0&0&0&0&0&0&0 \\ 0&1&1&1&0&0&0&-1&0&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&1&0&0&0&0&0 \\ 0&0&1&1&1&0&0&0&0&-1&0&0&0&0 \\ 0&0&1&1&1&0&0&0&0&0&1&0&0&0 \\ 1&1&1&1&1&0&0&0&0&0&0&-1&0&0 \\ 1&1&1&1&1&0&0&0&0&0&0&0&1&0 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ Y_1 \\ Z_1 \\ \vdots \\ Y_4 \\ Z_4 \end{pmatrix} = \begin{pmatrix} l \\ -u \\ l \\ -u \\ l \\ -u \\ l \\ -u \end{pmatrix} \tag{4.10}$$

We call the ILP matrix as A*. Clearly, the matrix* A *does not have consecutive ones in every column.* $\diamond$

In general, if a matrix does not have the consecutive ones property, it may be possible to re-order the windows to achieve the consecutive ones property. If such a re-ordering exists, it can be found and performed in $O(m + n + r)$ time, where r is the number of non-zero entries in the matrix [BL76]. Even when re-ordering cannot achieve the consecutive ones property there may, nevertheless, be an equivalent network matrix. Bixby and Cunningham [BC80] give a procedure to find an equivalent network matrix, when it exists, in $O(mr)$ time. Alternative procedure is given in Section 20.1 of [Sch86].

Example 4.7 *We show that the matrix of the problem described by Equations* (4.6)–(4.9) *is not equivalent to any network matrix. We denote by* A *the ILP matrix of the system:*

$$A = \begin{pmatrix} 1&1&1&0&0 \\ 1&1&1&0&0 \\ 0&1&1&1&0 \\ 0&1&1&1&0 \\ 0&0&1&1&1 \\ 0&0&1&1&1 \\ 1&1&1&1&1 \\ 1&1&1&1&1 \end{pmatrix}$$

At the first step of the test we need to construct a graph G_i for each ith row, $i = 1, \ldots, 8$ of the matrix A. Each G_i contains seven vertices $\{1, \ldots, 8\} \setminus \{i\}$. Two vertices k and p are adjacent in G_i if there exists a column j in A such that $A[k, j] \neq 0$, $A[p, j] \neq 0$ and $A[i, j] = 0$. Figure 4.7(a) shows G_1 and Figure 4.7(b) shows G_8, respectively.

We use the following statement from [Sch86].

If A is a network flow matrix, there exists an i for which G_i is disconnected.

Note that, for example, the graph G_8 is disconnected so A could be a network matrix. We continue the test. Following the procedure we consider the connected components of G_8. The graph G_8 contains 7 connected components, $C_1, \ldots, C_7$, one vertex each $C_i = \{i\}$, $i = 1, \ldots, 7$. Next we introduce a notion of support for a row, W. A support for a row is the set of coordinates in which the vector is nonzero. A support for the 8th row which is the set of non zero coordinates in A, $W_8 = \{1, 2, 3, 4, 5\}$ and compute values of W_i, $i = 1, \ldots, 7$ and U_k, $k = 1, \ldots, 7$ as follows:

$$W_i = W_8 \cap support\ of\ the\ ith\ row$$

$$U_k = \bigcup \{W_i | i \in C_k\}$$

In our case $W_i = U_i$, , $i = 1, \ldots, 7$ and $W_1 = W_2 = \{1, 2, 3\}$, $W_3 = W_4 = \{2, 3, 4\}$, $W_5 = W_6 = \{3, 4, 5\}$ and $W_7 = 1, 2, 3, 4, 5$.

Finally, we construct the undirected graph H with vertices $C_1, \ldots, C_7$ where two distinct vertices C_k and C_l are adjacent iff

$$W_l \cap W_k \neq \{\}$$

$$W_l \not\subseteq W_k$$

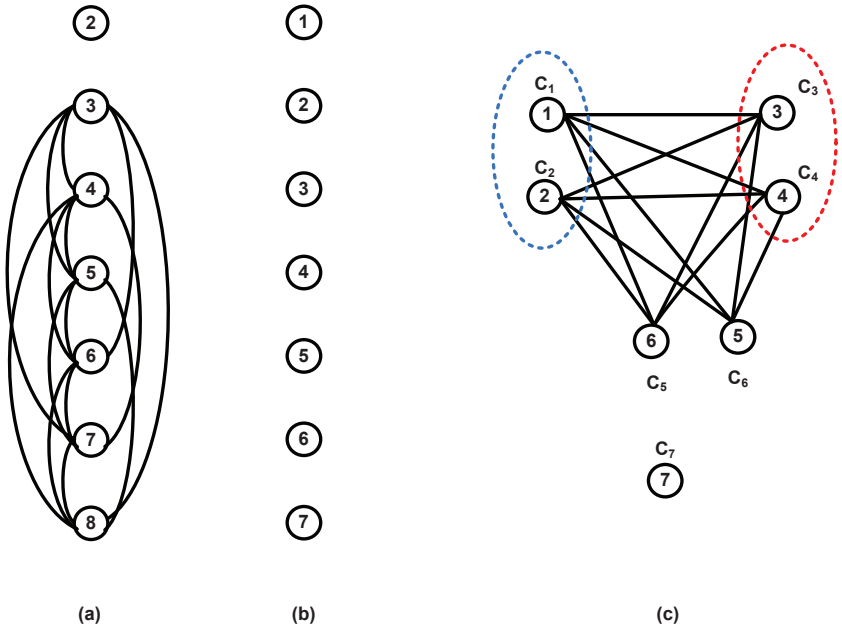
$$W_k \not\subseteq W_l$$

Note these conditions are a simplified version of the condition 4 from [Sch86], page 284, as in our case C_i are of size 1. Figure 4.7(c) shows the graph H. It is easy to see that graph is not bipartite. The set of vertices $\{1, 2\}$ and $\{3, 4\}$ have to belong to different partitions. At the same time vertices $\{5, 6\}$ are connected to vertices from both sets $\{1, 2\}$ and $\{3, 4\}$. We use the following simplified statement from [Sch86]:

A is a network matrix if and only if H is bipartite.

Therefore, the matrix A is not a network matrix. $\diamond$

Figure 4.7 (a) Graph G_1 (b)Graph G_8, (c) the undirected graph H



However, we observe that all rows of the matrix in Example 4.6 do have consecutive ones property if we ignore the slack variables. This means all columns have consecutive ones in the dual formulation of the problem. We exploit this property to reformulate the GEN-SEQUENCE constraint as a network flow problem.

Reformulation as an integer linear program. We explain how the reformulation works on the running example 4.6. We recall that GEN-SEQUENCE can be encoded as *ILP* with a constant cost function:

$$Maximise\ 0 \tag{4.11}$$

$$l \leq X_1 + X_2 + X_3, \quad -u \leq -X_1 - X_2 - X_3, \tag{4.12}$$

$$l \leq X_2 + X_3 + X_4, \quad -u \leq -X_2 - X_3 - X_4, \tag{4.13}$$

$$l \leq X_3 + X_4 + X_5, \quad -u \leq -X_3 - X_4 - X_5, \tag{4.14}$$

$$l \leq X_1 + X_2 + X_3 + X_4 + X_5, \quad -u \leq -X_1 - X_2 - X_3 - X_4 - X_5, \tag{4.15}$$

$$0 \leq X_i, \quad -1 \leq -X_i, i = 1, \ldots, 5 \tag{4.16}$$

Note that we include unary constraints on the variables X_i in our formulation. In the matrix form:

$$Maximise\ 0 \tag{4.17}$$

$$A\vec{X} \geq \vec{b}, \tag{4.18}$$

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & -1 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix},$$

$$\vec{X} = \Big(X_1, \ldots, X_5\Big)^T,$$

$$\vec{b} = \Big(l, -u, l, -u, l, -u, l, -u, l, -u, l, -u, 0, 0, 0, 0, 0, -1, -1, -1, -1, -1\Big)^T$$

The dual formulation of the problem is :

$$\begin{aligned} Minimise \quad & -b^T y \\ & A^T y = 0 \\ & y \geq 0 \end{aligned}$$

where

$$A^T = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \end{pmatrix},$$

As was mention above, since the matrix A has the consecutive ones property on rows, the matrix A^T has the consecutive ones property on columns. The dual problem can thus be converted to a network flow using the same transformation as with the SEQUENCE constraint.

From integer linear program to a network flow problem. We apply the Veinott and Wagner [WM62] procedure to A^T (4.19) and obtain a matrix with a single 1 and a single -1 in each column:

$$A^T = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

Theorem 4.3 *The* GEN-SEQUENCE *constraint is satisfiable if and only if there are no negative cycles in the flow graph associated with the dual linear program. This graph consists of* $2n + 2m$ *edges,* $n + 1$ *vertices, and an amount of flow to send equal to* 0.

Proof: Von Neumann's Strong Duality Theorem [AMO93] states that if the primal and the dual problems are feasible, then they have the same objective value. Moreover, if the primal is unsatisfiable, the dual is unbounded. The SEQUENCE constraint is thus satisfiable if the objective function of the dual problem is zero. It is unsatisfiable if it tends to negative infinity.

Negative cost cycle. If there is a negative cycle in the graph, then we can push an infinite amount of flow resulting in a infinitely small cost . Hence the dual problem is unbounded, and the primal is unsatisfiable.

Positive cost cycle. Suppose that there are no negative cycles in the graph. Pushing any amount of flow over a cycle of positive cost results in a flow of cost greater than zero. Such a flow is not optimal since the null flow has a smaller objective value.

Zero cost cycle. Pushing any amount of flow over a null cycle does not change the objective value. Therefore the null cost flow is an optimal solution and since this solution is bounded, the primal is satisfiable. Note that the objective value of the dual (zero) is in this case equal to the objective value of the primal.

The number of vertices in the graph is equal to the number variables in the constraint plus one as we add a zero row during the transformation, which gives $n + 1$ vertices. The number of edges equals to the number of constraints in the primal model, including $2n$ unary constraints on variables X, is $2n + 2m$. The total amount of flow is

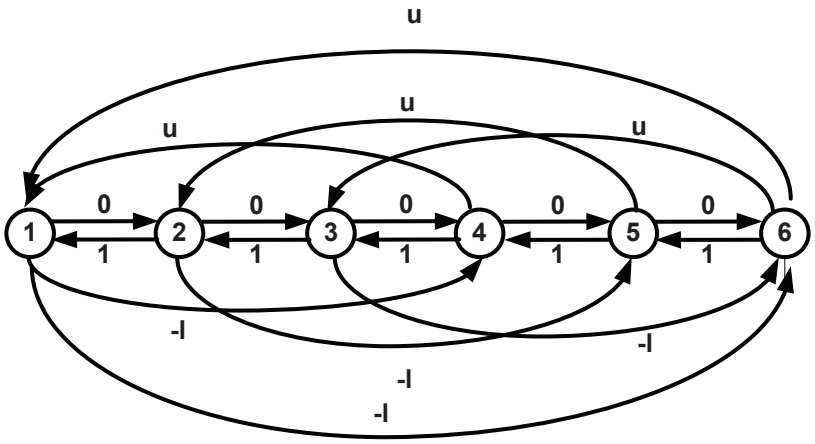
$$\sum_{i=1,b_i>0}^{2n+2m} b_i = 0.$$

$\diamond$

The flow associated with our running example is given in Figure 4.8. There are $5 + 1$ nodes labelled from 1 to 6 where node i is connected to node $i + 1$ with an edge of cost 0 and node $i + 1$ is connected to node i with an edge of cost 1. Note that these two edges correspond to the domain of the variable X_i. The lower bound of X_i is the weight of the edge from node i to node $i + 1$ and the upper bound of X_i is the edge from $i + 1$ to i. For each window we have an edge from ith node to $i + k$th node with cost $-l$ and an edge from $i + k$th to ith node with cost u. All nodes have a null supply and a null demand.

Enforcing domain consistency Based on Theorem 4.3 we can construct a network flow-based *DC* algorithm for the GEN-SEQUENCE constraint that outperforms the previous known algorithm by $O(n)$ factor [HPRS]. Algorithm 4.3 shows the pseudocode for this algorithm.

Figure 4.8 Network flow associated with the dual *ILP* model of GEN-SEQUENCE.



Algorithm 4.3 *DC* propagator for the GEN-SEQUENCE

1: **procedure** PROPAGATORGENSEQUENCE($\vec{p_1}, \ldots, \vec{p_m}, [X_1, \ldots, X_n]$)
2: Construct $G(V, E)$.
3: **if** $\exists$ a negative cycle in $G(V, E)$ **then**
4: return Failure.
5: Find all pairs shortest paths in G.
6: **for** $i \leftarrow 1$ **to** n **do**
7: Let (V_i, V_{i+1}) and (V_{i+1}, V_i) be the edges that corresponds to X_i in G.
8: Let $w_{i,j}$ be the weight of the path from V_i to V_j in G.
9: $D(X_i) = D(X_i) \setminus [-\inf, -w_{i,i+1})$
10: $D(X_i) = D(X_i) \setminus (w_{i+1,i}, \inf]$
11: return True.

Theorem 4.4 *Algorithm 4.3 enforces domain consistency of the* GEN-SEQUENCE *constraint in* $O(n^2 \log n + nm)$ *time.*

Proof: By Theorem 4.3 the absence of negative cycles in G guaranties existence of a solution of the constraints (line 3).

We find for each variable X_i the smallest (largest) value in its domain such that assigning this value to X_i does not create a negative cycle.

We recall that the lower bound corresponds to the weight of the edge $e_{lb(X_i)}$ from V_i to V_{i+1} and the upper bound corresponds to the weight of the edge $e_{ub(X_i)}$ from V_{i+1} to V_i. We find the weights, $w_{i,i+1}$ and $w_{i+1,i}$, of shortest paths from V_i to V_{i+1} and V_{i+1} to V_i, respectively, and ensure that the weights of $e_{lb(X_i)}$ and $e_{ub(X_i)}$ do not create a negative cycle. So we have conditions that $-w_{e_{lb(X_i)}} + w_{i+1,i} \geq 0$ or $X_i \leq w_{i+1,i}$. Clearly, any

large value of X_i creates a negative cycle in the graph. Similarly, $w_{e_{ub(X_i)}} + w_{i,i+1} \geq 0$ and $X_i \geq -w_{i,i+1}$.

Complexity argument: The flow graph has $O(n)$ nodes and $O(m)$ edges. Testing whether there is a negative cycle takes $O(nm)$ time using the Bellman-Ford algorithm. Johnson's algorithm solves the all-pair shortest path problem in $O(|V|^2 \log |V| + |V||E|)$ time which in our case gives $O(n^2 \log n + nm)$ time. $\diamond$

We can make the propagator incremental using the algorithm by Cotton and Maler [CM06] to maintain the shortest path between $|p|$ pairs of vertices in $O(|E| + |V| \log |V| + |P|)$ time upon edge reduction. Each time a variable domain is changed, the shortest paths can be recomputed in $O(m + n \log n)$ time. This gives $O(nm + n^2 \log n)$ time complexity down a branch of the search tree.

Algorithm 4.4 Incremental *DC* propagator for the GEN-SEQUENCE

1: **procedure** PROPAGATORGENSEQUENCEINC($X_k, \vec{p_1}, \ldots, \vec{p_m}, [X_1, \ldots, X_n]$)
2: Recompute all pairs shortest paths in G.
3: **for** $i \leftarrow 1$ **to** n **do**
4: Let (V_i, V_{i+1}) and (V_{i+1}, V_i) be the edges that corresponds to X_i in G.
5: Let $w_{i,j}$ be the weight of the path from V_i to V_j in G.
6: $D(X_i) = D(X_i) \setminus [-\inf, -w_{i,i+1})$
7: $D(X_i) = D(X_i) \setminus (w_{i+1,i}, \inf]$
8: return True.

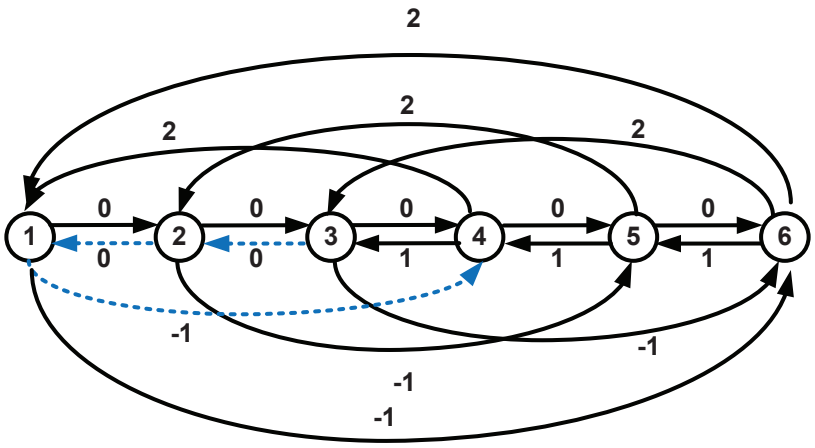
The pseudocode of the incremental algorithm is shown as Algorithm 4.4.

Example 4.8 *We continue our running example 4.6. We assume that $l = 1$ and $u = 2$. Suppose that X_1 and X_2 are assigned to the value 0. Consider the variable X_3. We show that X_3 has to take value 1. We find the shortest path in G from vertex 3 to vertex 4. The weight of this path $w_{3,4}$ is -1 (Figure 4.9, dashed edges). Hence, $X_3 \geq 1$.* $\diamond$

4.3 Decompositions of the SEQUENCE constraint

In this section we present several decompositions of the SEQUENCE constraint that achieve domain consistency for the SEQUENCE constraint and two that do not achieve domain consistency but are efficient in practice. The first algorithm that we present is based on encoding of the SEQUENCE constraint using the REGULAR constraint based on the last occurrence of a value from S (we will refer to this algorithm as LO). It exploits special features

Figure 4.9 GEN-SEQUENCE from Example 4.6 with $l = 1$, $u = 2$ and $X_1 = X_2 = 0$. The shortest path in G between vertices 1 and 4.



of practical applications of SEQUENCE, namely short windows and small bounds. The time complexity of this algorithm is linear in the number of variables, but it polynomially depends on k and l, u bounds. The second algorithm is based on the decomposition of SEQUENCE into simple linear constraints on cumulative sums encoding (CS). It does not achieve *DC*, but it works in $O(n)$ time and is very efficient for problems where $u - l$ is small. The third algorithm establishes *DC* on SEQUENCE by enforcing failed literal test on the second decomposition in $O(n^3)$ time (CS_{DC}). The fourth algorithm uses a dynamic programming technique to achieve *DC* in $O(nk^2u)$. It is based on the partial sum encoding of the SEQUENCE constraints (PS) and performs well for problems with small length of the sliding window. The fifth algorithm represents the SEQUENCE constraint as a conjunction of linear inequalities and enforces *DC* on SEQUENCE in $O(nlog(n))$ amortised time (DFB). The sixth algorithm also uses dynamic programming and takes only $O(nlog(k))$ time (LG). It does not achieve domain consistency, but it is stronger than the decomposition of SEQUENCE into AMONG constraints and is asymptotically faster. We also consider an improvement on the last decomposition that increases its propagation strength. Table 4.1 gives a summary of the decompositions and filtering algorithms for the SEQUENCE constraint that we consider in this chapter.

4.3.1 Decomposition into AMONG constraints (AD)

The first decomposition that we consider is the existing decomposition into AMONG constraints. This decomposition naturally follows from the definition of the SEQUENCE constraint. We call this the AD encoding. It was shown that the AD decomposition hinders propagation [HPRS]. We give another example that shows the weakness of the decomposi-

Table 4.1: A summary of the decompositions and filtering algorithms for the SEQUENCE constraint. The first column shows the name of the algorithm and the second column describes what this algorithm stands for.

PS	the *DC* decomposition based on partial sums
$HPRS$	the *DC* filtering algorithm based on cumulative sums
AD	the decomposition into AMONG constraints
LG	the decomposition based on a log based encoding
CS	the decomposition based on cumulative sums
LO	the *DC* decomposition based on the REGULAR constraint
FB	the *DC* filtering algorithm that on network flow
MR	the *DC* decomposition of multiple SEQUENCEs based on REGULAR

tion.

Example 4.9 *Consider the* SEQUENCE$(1, 1, 3, [X_1, X_2, X_3, X_4])$ *with domains* $D(X_1) = D(X_3) = D(X_4) = \{0, 1\}$ *and* $D(X_1) = 0$. *The decomposition contains* AMONG$(1, 1, [X_1, X_2, X_3], \{1\})$ *and* AMONG$(1, 1, [X_2, X_3, X_4], \{1\})$ *constraints. Both of these constraints are domain consistent, while enforcing domain consistency on the* SEQUENCE *constraint sets* X_4 *to* 0. $\diamond$

4.3.2 Decomposition based on cumulative sums (CS)

Our first reformulation is based on computing cumulative sums similar to the $HPRS$ algorithm described in Section 4.2.1. The idea of this encoding is to mimic the behaviour of the $HPRS$ algorithm using a set of constraints over *integer variables* that encode cumulative sums values. Therefore, we encode a sequence of cumulative sums *values* y_i, $i = 0, \ldots, n$ by introducing a sequence of cumulative sum *integer variables*, Y_j, $i = 0, \ldots, n$, where $Y_j = \sum_{i=1}^{j} X_i$ each with domain $[0, j]$. We encode this linearly

$$Y_0 = 0, Y_i = X_i + Y_{i-1}, \qquad i = 1, \ldots, n. \tag{4.19}$$

$$l \leq Y_{i+k} - Y_i, \quad i = 1, \ldots, n - k + 1 \tag{4.20}$$

$$Y_{i+k} - Y_i \leq u, \quad i = 1, \ldots, n - k + 1 \tag{4.21}$$

We call this the CS encoding. Not surprisingly, this encoding hinders propagation.

Example 4.10 *Consider, for example,* SEQUENCE $(1, 2, 2, [X_1, X_2, X_3, X_4], \{1\})$, $D(X_3) = \{0\}$ *and* $D(X_i) = \{0, 1\}$, $i \in \{1, 2, 4\}$. *The corresponding cumulative sum*

variables Y have the following domains: $Y_0 \in \{0\}$, $Y_1 \in \{0,1\}$, $Y_2, Y_3 \in \{1,2\}$, $Y_4 \in \{2,3\}$. All constraints in CS are bounds consistent. However, domain consistency on the SEQUENCE *prunes the value 0 from the domains of X_2 and X_4.* $\diamond$

However, the CS decomposition detects domain disentailment (see Section 2.4.3 for the domain disentailment definition) which is exactly what the repair procedure from the $HPRS$ algorithm achieves.

Theorem 4.5 *Enforcing bounds consistency on CS detects domain disentailment for the* SEQUENCE *constraint and takes $O(n^2)$.*

Proof: Suppose that all constraints (4.19)–(4.21) are bounds consistency. We show that the lower bounds of variables Y can be transformed to a solutions of the SEQUENCE constraint. The same result holds for the upper bounds of Y.

Consider the assignment $x_i = lb(Y_i) - lb(Y_{i-1})$, $i = 1, \ldots, n$. The value of X_i is either 0 or 1. By constraint (4.20) we have: $l \leq lb(Y_{i+k}) - lb(Y_i)$, $i = 1, \ldots, n-k+1$. Otherwise the lower bound of the variable Y_{i+k} does not have a bound support. By (4.21) we have: the $lb(Y_{i+k}) - lb(Y_i) \leq u$, $i = 1, \ldots, n-k+1$. Otherwise the lower bound of the variable Y_i does not have a bound support.

We rewrite this as $l \leq lb(Y_{i+k}) - lb(Y_{i+k-1}) + lb(Y_{i+k-1}) - \ldots - lb(Y_{i+1}) + lb(Y_{i+1}) - lb(Y_i) \leq u$. This means that $l \leq x_{i+k} + x_{i+k-1} \ldots + x_{i+1} \leq u$, $i = 1, \ldots, n-k+1$. Hence the assignment x satisfies the SEQUENCE constraint.

Complexity argument: There are $O(n)$ constraints in the system and each constraint can be invoked at most $O(n)$ times. All constraints are of bounded size. Therefore the total time complexity is $O(n^2)$. $\diamond$

Theorem 4.5 suggests that we can enforce domain consistency on the SEQUENCE constraint by enforcing singleton bounds consistency on the set of constraints (4.19) - (4.21).

Theorem 4.6 *Singleton bounds consistency on the variables $X_1, \ldots, X_n$ on CS enforces domain consistency on* SEQUENCE$[X_1, \ldots, X_n]$ *and takes $O(n^3)$ time.*

Proof: Proof of the theorem follows from Equations 2.22 and Theorem 4.5. $\diamond$

Example 4.11 *Consider the* SEQUENCE *constraint from Example 4.10. Enforcing singleton bounds consistency on CS prunes the value 0 from the domains of X_2 and X_4.* $\diamond$

The CS decomposition can be also applied to propagate the GEN-SEQUENCE constraint. However, the number of constraints in the decomposition can increased by factor of

$O(n)$, because there are $O(n^2)$ windows of variables in the worst case. Theorems 4.5–4.6 holds for the GEN-SEQUENCE constraint except of complexity argument. The complexity of detecting disentailment is $O(n^3)$. Therefore, the time complexity of enforcing domain consistency is $O(n^4)$.

4.3.3 Decomposition based on REGULAR constraints (LO)

As mentioned is Section 4.2.2, van Hoeve *et al.* give an encoding of SEQUENCE using the REGULAR constraint [HPRS]. This permits domain consistency to be achieved in $O(n2^k)$ time. The states of the automata used in this encoding record which of the last k values encountered are from the set S. We can slightly improve upon this encoding by having states record just the last $k-1$ values encountered. A transition is then permitted if the last $k-1$ values encountered plus the current variable have the correct frequency of values from the given set.

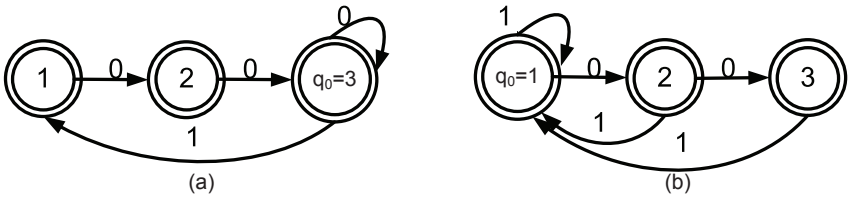
We now give an alternative encoding using the REGULAR constraint. The encoding exploits two features of many car sequencing and staff rostering problems. First, such problems typically only place upper bounds on occurrences (e.g. at most 1 in 3 cars can have the sun-roof). Second, in many problems the lower and upper bounds are typically small (e.g. in all data files in Prob001 in CSPLib, $u \leq 2$ and $k \leq 5$).

Suppose we wish to ensure that at most 1 in k Boolean variables X_i take the value 1. Consider an automaton whose states record the minimum of k and the distance back to the last occurrence of 1. If 1 has not yet occurred, the distance is taken to be k. The transition function from the state q on seeing X_i is $t(q, X_i) = \min(k, q+1)$ if $X_i = 0$ and $t(q, X_i) = 1$ if $q = k$ and $X_i = 1$. The initial state of the automaton is k and any state is accepting (Figure 4.10(a)). A similar automaton can be constructed for $u > 1$, but we need states to record the distances back to the last u occurrences of value 1.

Now, suppose we wish to ensure at least 1 in k variables take the value 1. The states of the automaton record the distance back to the last occurrence of 1. If 1 has not yet occurred, the distance is taken to be the number of variables seen so far. The transition function from the state q on seeing X_i is $t(q, X_i) = q+1$ if $X_i = 0$ and $q < k$, and $t(q, X_i) = 1$ if $q \leq k$ and $X_i = 1$. The initial state of the automaton is 1 and any state is accepting (Figure 4.10(b)).

Thus, to encode a SEQUENCE constraint, we convert it into a sequence of ATLEAST and ATMOST constraints. We can convert the sequence of ATLEAST constraints into a sequence of ATMOST constraints (or vice verse depending on which representation gives smaller complexity) by inverting the value being counted. For example, the constraint that

Figure 4.10 (a) Automation for the ATMOSTSEQ constraint with $u = 1$ and $k = 3$ (b) Automation for the ATLEASTSEQ constraint with $l = 1$ and $k = 3$



at least 3 in any 5 days must be work days is equivalent to at most 2 in 5 days are rest days. Finally, we construct the product of the automata for the two sequences of ATMOST or ATLEAST constraints. The complexity of enforcing domain consistency on SEQUENCE using this encoding is $O(nk^{\min(l,k-l)} \times k^{\min(u,k-u)})$. Note that if $u > k - u$ for the ATMOSTSEQ constraint then we convert ATMOSTSEQ to ATLEASTSEQ first and, then, build an automaton to represent the constraint. This transformation is the reason for the min operator in the time complexity. We will refer to this encoding as LO as the automaton records the last occurrence(s).

4.3.4 Decomposition based on partial sums (PS)

The next encoding is arguably the simplest encoding which enforces domain consistency. The PS encoding simply decomposes the constraint into a set of equations based on partial sums: $P_{i,j} = \sum_{l=i}^{j} X_l$ each with domain $[0, \min(u, j - i + 1)]$. The PS encoding of the SEQUENCE constraint is $P_{i,i+k-1} \leq u$ and $P_{i,i+k-1} \geq l$ for $1 \leq i \leq n - k + 1$ as well as $P_{i,i} = X_i$ for $1 \leq i \leq n$ and most importantly, all possible ways of adding two of these variables to create another: $P_{i,j} = P_{i,m} + P_{m+1,j}$ for $1 \leq i \leq m < j \leq n, j \leq i + k - 1$. Note there are $O(nk^2)$ constraints of the last form.

Theorem 4.7 *Bounds consistency on the PS encoding enforces domain consistency of the* SEQUENCE *constraint in $O(nk^2u)$ down a branch of the search tree.*

Proof: We say that domain D bounds captures C if for each $X_i + \cdots + X_j \leq c \in C$, $\max D(P_{i,j}) \leq c$ and for each $X_i + \cdots + X_j \geq c \in C$, $\min D(P_{i,j}) \geq c$. Clearly the domain resulting from bounds consistency applied to PS bounds captures the AD encoding.

We show that if D is bounds consistent with PS and bounds captures C then it also bounds captures C' which results from eliminating the least (or greatest) indexed variable X_i.

We consider the least variable X_i, the greatest is similar. Consider Fourier-Motzkin elimination of X_i. For each pair of constraints in C of the form $X_i + \cdots + X_{j_1} \leq c_1$ and

$X_i + \cdots + X_{j_2} \geq c_2$, Fourier-Motzkin elimination creates the constraint

1. (a) if $j_1 > j_2$ then $X_{j_2+1} + \cdots + X_{j_1} \leq c_1 - c_2$,
2. (b) if $j_1 < j_2$ then $X_{j_1+1} + \cdots + X_{j_2} \geq c_2 - c_1$.
3. (c) if $j_1 = j_2$ then $0 \leq c_1 - c_2$.

Now since D bounds captures C we have $ub(P_{i,j_1}) \leq c_1$ and $lb(P_{i,j_2}) \geq c_2$. For case (a) by bounds consistency on the constraint $P_{i,j_1} = P_{i,j_2} + P_{j_2+1,j_1}$ we have $ub(P_{j_2+1,j_1}) \leq c_1 - c_2$, for (b) bounds consistency on $P_{i,j_2} = P_{i,j_1} + P_{j_1+1,j_2}$ gives $lb(P_{j_1+1,j_2}) \geq c_2 - c_1$, and for (c) the new constraint is $true$ since otherwise $D(P_{i,j_1}) = \emptyset$. Hence the new constraint is bounds captured by D.

To prove domain consistency of SEQUENCE, let C be the AD encoding plus inequalities fixing X variables in the current domain D (which we assume is bounds consistency with PS). Clearly D bounds captures C. Consider any variable X_i, and eliminate from C other variables in order $X_1, \ldots, X_{i-1}, X_n, X_{n-1}, \ldots, X_{i+1}$ to obtain C'. Note that if we eliminate variables in this order we preserve the consecutively property of each inequality. Now C' only involves the variable X_i. By the correctness of Fourier-Motzkin elimination (Section 2.2.3) there are solutions of C extending any solution of C'. Since D bounds captures C' by repeated use of the above argument we have that there are solutions to C for each $d \in D(X_i)$.

Complexity argument: We note that the domains of the variables in each constraint $P_{i,j} = P_{i,m} + P_{m+1,j}$ can change at most $3u$ times in a forward computation. Each propagation is $O(1)$ hence the overall complexity down a branch is $O(nk^2u)$. $\diamond$

As in the case of the CS decomposition, the PS decomposition can be also used to propagate the GEN-SEQUENCE$(\vec{p_1}, \ldots, \vec{p_t}, [X_1, X_2, \ldots, X_n], S)$ constraint(Definition 4.2). We introduce variables $P_{i,j}$, $1 \leq i \leq j \leq O(n)$ and the following constraints: $P_{s_h,s_h+k_h-1} \leq u_h$ and $P_{s_h,s_h+k_h-1} \geq l_h$ for $1 \leq h \leq t$, $P_{i,i} = X_i$ for $1 \leq i \leq n$ and $P_{i,j} = P_{i,m} + P_{m+1,j}$ for $1 \leq i \leq m < j \leq n, j \leq O(n)$. In the case of the GEN-SEQUENCE constraint there are $O(n^3)$ constraints of the last form. Theorem 4.7 holds for the GEN-SEQUENCE constraint as the proof does not depend on that number of the variables $P_{i,j}$.

4.3.5 Decomposition based on a log based encoding of SEQUENCE (LG)

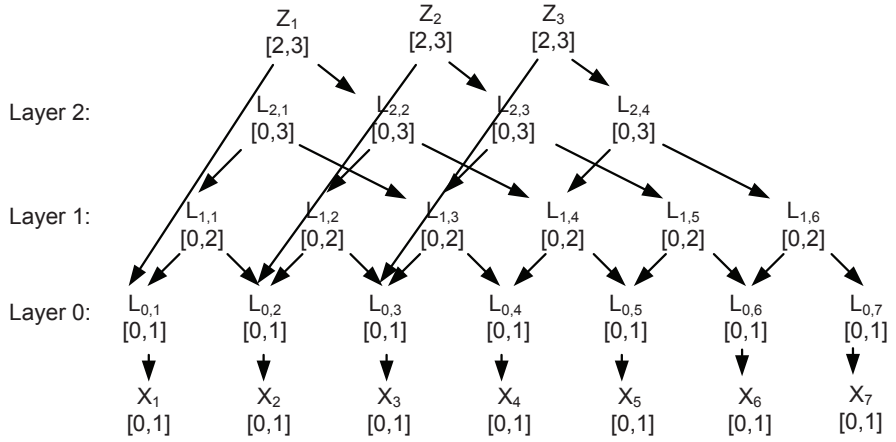
Our final encoding (called LG) is based on a simple dynamic program that builds up partial sums on counts. We introduce $L[i,j]$ with domain $[0, \min(u, 2^{i-1})]$ for the partial sums $\sum_{h=j}^{j+2^i-1} X_h$ where $0 \leq i \leq \lfloor \log h \rfloor$ and $1 \leq j \leq n - 2^i + 1$. Note that $L[i,j] = P_{j,j+2^i-1}$.

This requires the following constraints: $L[0,j] = X_j, 1 \leq j \leq n$ and $L[i,j] = L[i-1,j] + L[i-1, j+2^{i-1}], 1 \leq j \leq n, i > 0$. Suppose $k = \sum_{i=1}^{m} 2^{a_i}$ where $a_1 < \ldots < a_m$ (in other words, a_i is the ith bit set in the binary representation of h). We also need the vector, Z_1 to Z_{n-h+1} each with domain $[l,u]$ and constraint:

$$Z_j \quad = \quad \sum_{i=1}^{m} L[a_i, j + \sum_{h=1}^{i-1} 2^{a_h}]$$

Figure 4.11 shows initial domains and intravariable dependencies for the SEQUENCE $(2,3,5,[X_1,\ldots,X_7])$ constraint.

Figure 4.11 Dependencies between partial sums variables L and X and their initial domains for the SEQUENCE$(2,3,5,[X_1,\ldots,X_7])$ constraint.



We have $O(n \log h)$ variables L that are subject to $O(n \log h)$ ternary constraints and $O(n)$ variables Z that are subject to $O(n)$ linear constraints of length $O(\log h)$ The constraint propagation cost equals to the number of invocations of the filtering algorithm for this constraint times the cost of one invocation. The number of invocations is proportional to the number of values in its variable domains. Hence, the propagation cost of a ternary constraint is $O(u)$. For all ternary constraints we have $PC_1 = O(u)O(n \log h) = O(nu \log h)$. Also, we have $O(n)$ variables Z that are subject to $O(n)$ linear constraints. Note that this constraint is a constraint of unbounded arity. However, as we pointed out in Section 2.4.3, we can further decompose this constraint into constraints of fixed arity constraints and this decomposition does not hinder propagation. Hence, instead of having $O(n)$ linear constraints of arity $O(\log h)$, we have $O(n \log h)$ ternary constraints.

The cardinality of each variable in these constraints is bounded by $O(u)$. Consequently, the propagation cost for linear constraints is $PC_2 = O(u)O(n \log h) = O(nu \log h)$. Therefore we can enforce bounds consistency on this encoding in $O(nu \log h) + O(nu \log h) = O(nu \log h)$ time down the whole branch of a search tree. However,

this may not achieve domain consistency on the SEQUENCE constraint.

Example 4.12 *Consider X_1 to X_4 with domains $\{0,1\}$, $X_5 = X_6 = 1$, and the global constraint* SEQUENCE$(4, 4, 5, [X_1, \ldots, X_6])$. *Then, we have:*

$$\begin{aligned} L[0,j] &= X_j \\ L[1,j] &= L[0,j] + L[0,j+1] \\ L[2,j] &= L[1,j] + L[1,j+2] \\ Z_1 &= L[0,1] + L[2,2] \\ Z_2 &= L[0,2] + L[2,3] \end{aligned}$$

Enforcing BC on the decomposition gives $L[0,1], L[0,2], L[0,3], L[0,4] \in \{0,1\}$, $L[0,5] = L[0,6] = 1$, $L[1,1], L[1,2], L[1,3], L[1,4] \in \{1,2\}$, $L[1,5] = 2$, $L[2,2], L[2,3] \in \{3,4\}$, and $Z_1 = Z_2 = 4$. However, enforcing domain consistency of the SEQUENCE *constraint prunes value zero from $D(X_1)$.* $\diamond$

There are a number of redundant constraints which we can add to improve propagation. For example, we can post:

$$Z_j = \sum_{i=1}^{m} L[a_i, j + \sum_{h=i+1}^{m} 2^{a_h}]$$

It is not hard to show that such additional redundant constraints can help propagation.

Example 4.13 *Suppose we add the redundant constraints to the constraint in Example 4.12:*

$$\begin{aligned} Z_1 &= L[2,1] + L[0,5] \\ Z_2 &= L[2,2] + L[0,6] \end{aligned}$$

Enforcing bounds consistency now sets $L[2,1] = L[2,2] = 3$, $L[0,1] = 1$ and $X_1 = 1$. $\diamond$

4.3.6 Theoretical comparison

We compare theoretically those encodings on which we may not achieve domain consistency on the SEQUENCE constraint. We will show that we get more propagation with LG than AD, but that AD, CS and LG are otherwise incomparable. It should be noted that during propagation all auxiliary variables in CS, LG, AD and PS encodings will always have ranges as their domains, consequently, bounds consistency is equivalent to domain consistency for them.

Proposition 4.1 *Bounds consistency on LG is strictly stronger than bounds consistency on AD.*

Proof: Suppose LG is bounds consistent. Consider any AMONG constraint in AD. It is not hard to see how, based on the partial sums in LG, we can construct support for any value assigned to any variable in this AMONG constraint. To show strictness, consider SEQUENCE$(l, u, k, [X_1, \ldots, X_6])$, $l = u = 3$, $k = 4$, with $X_1, X_2 \in \{1\}$ and $X_3, \ldots, X_6 \in \{0, 1\}$. Enforcing bounds consistency on LG fixes $X_5 = X_6 = 1$. On the other hand, AD is bounds consistent. $\diamond$

Proposition 4.2 *Bounds consistency on CS is incomparable to bounds consistency on AD.*

Proof: Consider SEQUENCE $(1, 1, 3, [X_1, X_2, X_3, X_4])$ with $X_1 \in \{0\}$ and $X_2, X_3, X_4 \in \{0, 1\}$. Now AD is bounds consistent. In CS, we have $Y_0, Y_1 \in \{0\}$, $Y_2 \in \{0, 1\}$, $Y_3, Y_4 \in \{1\}$. As Y_3 and Y_4 are equal, enforcing bounds consistency on CS prunes 1 from the domain of X_4.

Consider SEQUENCE $(1, 2, 2, [X_1, X_2, X_3, X_4])$ with $X_3 \in \{0\}$ and $X_1, X_2, X_4 \in \{0, 1\}$. In CS, we have $Y_0 \in \{0\}$, $Y_1 \in \{0, 1\}$, $Y_2, Y_3 \in \{1, 2\}$, $Y_4 \in \{2, 3\}$. All constraints in CS are bounds consistent. Enforcing bounds consistency on AD prunes 0 from the domains of X_2 and X_4. $\diamond$

From the proof of Theorem 4.2 it follows that bounds consistency on CS does not enforce domain consistency on SEQUENCE when SEQUENCE is monotone.

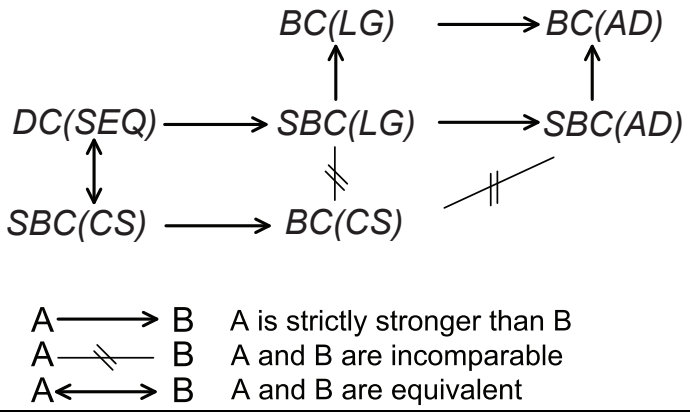
Proposition 4.3 *Bounds consistency on CS is incomparable with bounds consistency on LG.*

Proof: Consider SEQUENCE $(2, 2, 4, [X_1, X_2, X_3, X_4, X_5])$ with $X_1 \in \{1\}$ and $X_2, X_3, X_4, X_5 \in \{0, 1\}$. All constraints in the LG decomposition are bounds consistent. In CS, we have $Y_0 \in \{0\}$, $Y_1 \in \{1\}$, $y_2, Y_3 \in \{1, 2\}$, $Y_4 \in \{2\}$, $Y_5 \in \{3\}$. As Y_4 and S_5 are ground and $Y_5 = Y_4 + 1$, enforcing bounds consistency on CS fixes $X_5 = 1$.

Consider SEQUENCE $(2, 3, 3, [X_1, X_2, X_3, X_4])$ with $X_1 = 1$ and $X_2, X_3, X_4 \in \{0, 1\}$. Now CS is bounds consistent. However, enforcing bounds consistency on LG prunes 0 from X_4. $\diamond$

Recall that singleton bounds consistency on CS is equivalent to domain consistency on SEQUENCE. We therefore also consider the effect of singleton consistency on the other encodings where propagation is hindered. Unlike CS, singleton bounds consistency on AD or LG may not prune all possible values.

Figure 4.12 Relations among decompositions of the SEQUENCE constraint.



Proposition 4.4 *Domain consistency on* SEQUENCE *is strictly stronger than singleton bounds consistency on* LG.

Proof: Consider SEQUENCE $(2, 2, 4, [X_1, X_2, X_3, X_4, X_5], \{1\})$ with $X_1 \in \{1\}$ and X_2, X_3, X_4, $X_5 \in \{0, 1\}$. Consider $X_5 = 0$ and the LG decomposition. We have $L[0, 1] \in \{1\}$, $L[0, 2], L[0, 3], L[0, 4] \in \{0, 1\}$, $L[0, 5] \in \{0\}$, $L[1, 1], L[1, 2] \in \{1, 2\}$, $L[1, 3], L[1, 4] \in \{0, 1\}$, $L[2, 1], L[2, 2] \in \{2\}$. All constraints in LG are bounds consistent. Consequently, we do not detect that $X_5 = 0$ does not have support. $\diamond$

Proposition 4.5 *Domain consistency on* SEQUENCE *is strictly stronger than singleton bounds consistency on* AD.

Proof: By transitivity from Theorems 4.4 and 4.1. $\diamond$

We summarise relations among decompositions in Figure 4.12. As before, we denote *SBC(X)* enforcing singleton bounds consistency on a decomposition X. We collapse all decomposition that enforce *DC* in a single node *DC*(SEC) to simplify the diagram.

An interesting research question, that was arose by an anonymous reviewer, is that to consider theoretical properties of combinations of these encodings. For example, the AD and CS encoding are incomparable, so we might expect that their compliment each other.

4.4 Generalisations of the SEQUENCE constraint

In this section we consider generalisations of the SEQUENCE constraint and soft forms of this constraint, like the SOFTSEQUENCE constraint.

4.4.1 The generalised SEQUENCE constraint

The GEN-SEQUENCE constraint is a natural generalisation of the SEQUENCE constraint, where AMONG constraints with different parameters are enforced on consecutive variables. We recall Definition 4.3 of the constraint. GEN-SEQUENCE$(\vec{p_1}, \ldots, \vec{p_m}, [X_1, X_2, \ldots, X_n], S)$ holds if and only if AMONG$(l_i, u_i, k_i, [X_{s_i}, \ldots, X_{s_i+k_i-1}], S)$ for $1 \leq i \leq m$ where $\vec{p_i} = \langle l_i, u_i, k_i, s_i \rangle$.

Section 4.2.4 gives a reformulation technique to propagate the GEN-SEQUENCE constraint that exploit the consecutive ones in each row property in *ILP* formulation of GEN-SEQUENCE. The algorithm runs in $O(nm + n^2 \log n)$ time down a branch of the search tree.

4.4.2 The Soft SEQUENCE constraint

A soft form of the SEQUENCE constraint is a relaxation of the SEQUENCE constraint that often occurs in practice. The soft SEQUENCE constraint takes into account by how much each AMONG constraint is violated. We recall the definition of the SOFTSEQUENCE constraint 4.4. SOFTSEQUENCE$(l, u, k, T, [X_1, \ldots, X_n])$ holds iff:

$$T \quad \geq \quad \sum_{i=1}^{n-k+1} \max(l - \sum_{j=0}^{k-1} X_{i+j}, \sum_{j=0}^{k-1} X_{i+j} - u, 0) \tag{4.22}$$

As before to simplify notations, we consider SOFTSEQUENCE on Boolean variables as this does not hinder propagation .

Example 4.14 (Running example (SOFTSEQUENCE)) *Consider a relaxation of Example 4.1. Suppose we allow a schedule to violate individual* AMONG *constraints but we restrict the total number of violations to be* 1. *This restriction can be encoded as the soft* SOFTSEQUENCE$(l, u, k, T, [X_1, \ldots, X_6])$, $l = 1$, $u = 2$, $k = 3$ *and* $T \leq 1$ *constraint. Again, we vary the values of parameters* l, u, k *and* T *to highlight properties of our algorithms.* $\diamond$

We again convert to a flow problem by means of an integer linear program, but this time with a varying objective function. Consider the reformulation using our running example 4.14. We introduce variables, Q_i and P_i to represent the penalties that may arise from violating lower and upper bounds respectively. We can then express this SOFTSEQUENCE

constraint as follows. The objective function gives a lower bound on T.

$$\begin{aligned}
&\text{Minimise} \sum_{i=1}^{4}(P_i + Q_i) && \text{subject to :}\\
&X_1 + X_2 + X_3 - Y_1 + Q_1 = l, && X_1 + X_2 + X_3 + Z_1 - P_1 = u,\\
&X_2 + X_3 + X_4 - Y_2 + Q_2 = l, && X_2 + X_3 + X_4 + Z_2 - P_2 = u,\\
&X_3 + X_4 + X_5 - Y_3 + Q_3 = l, && X_3 + X_4 + X_5 + Z_3 - P_3 = u,\\
&X_4 + X_5 + X_6 - Y_4 + Q_4 = l, && X_4 + X_5 + X_6 + Z_3 - P_4 = u
\end{aligned}$$

where Y_i, Z_i, P_i and Q_i are non-negative. The penalty variables used for SOFTSEQUENCE arise directly out of the problem description and occur naturally in the LP formulation. We could also view them as arising through the methodology of [HPR06], where edges with costs are added to the network graph for the hard constraint to represent the softened constraint.

In matrix form, this is:

$$Minimise \sum_{i=1}^{4}(P_i + Q_i) \quad \text{subject to:}$$

$$\begin{pmatrix}
1&1&1&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&1&1&1&0&0&0&0&0&1&0&0&0&0&0&0&0&-1&0&0&0&0\\
0&0&1&1&1&0&0&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0\\
0&0&1&1&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&-1&0&0\\
0&0&0&1&1&1&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&1&0\\
0&0&0&1&1&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&-1
\end{pmatrix}
\begin{pmatrix}
X_1\\ \vdots\\ X_6\\ Y_1\\ Z_1\\ \vdots\\ Y_4\\ Z_4\\ Q_1\\ P_1\\ \vdots\\ Q_4\\ P_4
\end{pmatrix}
=
\begin{pmatrix}
l\\u\\l\\u\\l\\u\\l\\u
\end{pmatrix},$$

If we transform the matrix as before, we get a minimum cost network flow problem:

$$Minimise \sum_{i=1}^{4}(P_i + Q_i) \quad \text{subject to:}$$

$$\begin{pmatrix}
1&1&1&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&0&0&0&0&0&0&-1&-1&0&0&0&0&0&0\\
-1&0&0&1&0&0&0&-1&-1&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&-1&-1&0&0&0&0\\
0&-1&0&0&1&0&0&0&0&-1&-1&0&0&0&0&0&0&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&-1&-1&0&0\\
0&0&-1&0&0&1&0&0&0&0&0&-1&-1&0&0&0&0&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&-1&-1\\
0&0&0&-1&-1&-1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&1
\end{pmatrix}
\quad \vec{X} = \vec{b},$$

where

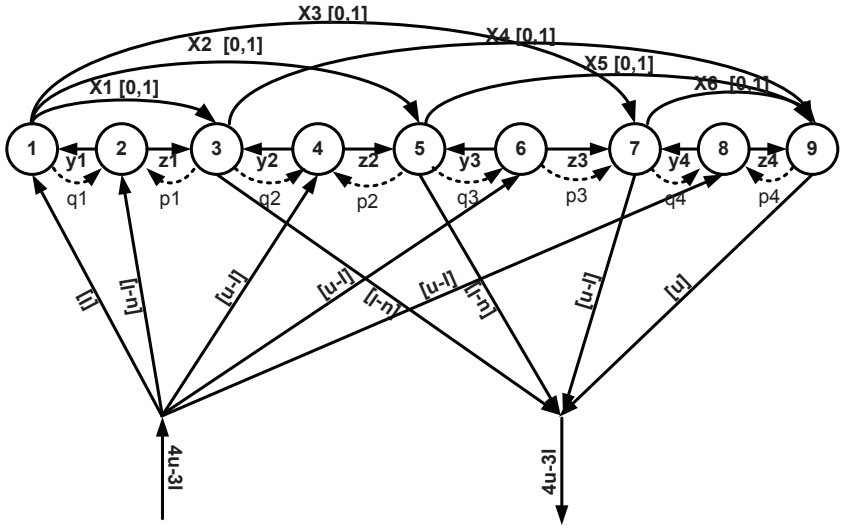
$$\vec{X}^T = \Big(X_1 \ldots X_6\ Y_1\ Z_1 \ldots Y_4\ Z_4\ Q_1\ P_1 \ldots Q_4\ P_4\Big)$$

and

$$\vec{b}^T = \Big(l, u-l, l-u, u-l, l-u, u-l, l-u, u-l, \ -u\Big)$$

The flow graph $G = (V, E)$ for this system is presented in Figure 4.13. Dashed edges have cost 1, while other edges have cost 0. The minimal cost flow in the graph corresponds to a minimal cost solution to the system of equations

Figure 4.13 A flow graph for SOFTSEQUENCE$(l, u, 3, T, [X_1, \ldots, X_6])$



Theorem 4.8 *For any constraint* SOFTSEQUENCE$(l, u, k, T, [X_1, \ldots, X_n], v)$*, there is an equivalent network flow graph. There is a one-to-one correspondence between solutions of the constraint and feasible flows of cost less than or equal to* $\max(D(T))$.

Proof: The existence of a network flow graph follows from the equivalence between solutions of *ILP* and solutions of the SEQUENCE constraint, correctness of the Veinott-Wagner procedure and Theorem 2.5. $\diamond$

Using Theorem 4.8, we construct a *DC* filtering algorithm for the SOFTSEQUENCE constraint. A pseudocode for a network flow-based *DC* algorithm for the SOFTSEQUENCE constraint is shown as Algorithm 4.5.

Theorem 4.9 *Algorithm 4.5 enforces domain consistency on the variables* X *of the* SOFTSEQUENCE *constraint and bounds consistency on the variable* T *in* $O(n^2 \log n \log\log u)$ *time.*

Proof: By Theorem 4.8 a support for value 1 (0) in $D(X_i)$, $i = 1, \ldots, n$ exists if and only if there exists a min cost flow f that sends (does not send) a unit flow through the edge labelled with X_i with $w(f) \leq ub(T)$ by Theorem 4.8. The lower bound of T is supported if and only if there exists a feasible flow of cost at most $lb(T)$ (line 7).

Algorithm 4.5 *DC* propagator for the SOFTSEQUENCE

1: **procedure** PROPAGATORSOFTSEQUENCE($l, u, k, T, [X_1, \ldots, X_n]$)
2: Construct $G(V, E)$.
3: Construct min cost flow f.
4: **if** $w(f) > ub(T)$ **then**
5: return Failure.
6: **else**
7: $D(T_i) = D(T_i) \setminus [-\inf, w(f) - 1]$
8: Find all pairs shortest paths in $G_f = (V, E_f)$.
9: **for** $i \leftarrow 1$ **to** n **do**
10: Let $e_{X_i} = (V_i, V_j)$ be the edge that corresponds to the variable X_i in G.
11: Let $w_{i,j}$ be the weight of the shortest path from V_i to V_j in G_f.
12: **if** $f(e_{X_i}) = 0$ and $w(f) - w_{j,i} + w(e_{X_i}) > ub(T)$ **then**
13: $D(X_i) = D(X_i) \setminus \{0\}$
14: **if** $f(e_{X_i}) = 1$ and $w(f) + w_{i,j} - w(e_{X_i}) > ub(T)$ **then**
15: $D(X_i) = D(X_i) \setminus \{1\}$
16: return True.

A min cost flow f in the graph $G = (V, E)$ gives a support for the lower bound of the variable T. Moreover, it gives a support for one of values of each variable (line 4).

If $f(e_{X_i})$ pushes a unit of flow through the edge e_{X_i} then the value 1 is supported otherwise the value 0 is supported. Suppose that $f(e_{X_i}) = 0$. (The case $f(e_{X_i}) = 1$ is similar). To find support for $X_i = 0$ we have to find a min cost flow f' such that $f'(e_{X_i}) = 0$ and make sure that $w(f') \leq ub(T)$ (line 12). Hence, by Theorem 2.1, we need to find the weight $w_{j,i}$ of a shortest path from V_j to V_i in the residual graph G_f. We also have to make sure that if we redirect the flow through the edge e_{X_i} the weight of the min cost flow is less than $lb(T)$, so that $w(f) - w(e_{X_i}) + w_{j,i} \leq ub(T)$,

Complexity argument: The minimal cost flow in a graph can be found in $O(|V||E| \log\log U \log |V|C) = O(n^2 \log n \log\log u)$ time [AMO93] (line 4). All pairs shortest path algorithm on the residual graph takes $O(n^2 \log n)$ time using Johnson's algorithm [CLRS01](line 8). The loop 9–15 takes $O(n)$ time. This gives an $O(n^2 \log n \log\log u)$ time complexity to enforce *DC* on SOFTSEQUENCE. $\diamond$.

Example 4.15 *Consider how Algorithm 4.5 works on the running example 4.14. We suppose that the first three variables are fixed to the value one,* $X_1 = X_2 = X_3 = 1$. *A min*

Figure 4.14 A flow graph for SEQUENCE$(1, 2, 3, T, [1, 1, 1, X_4, X_5, X_6])$, $T = [0, 1]$.

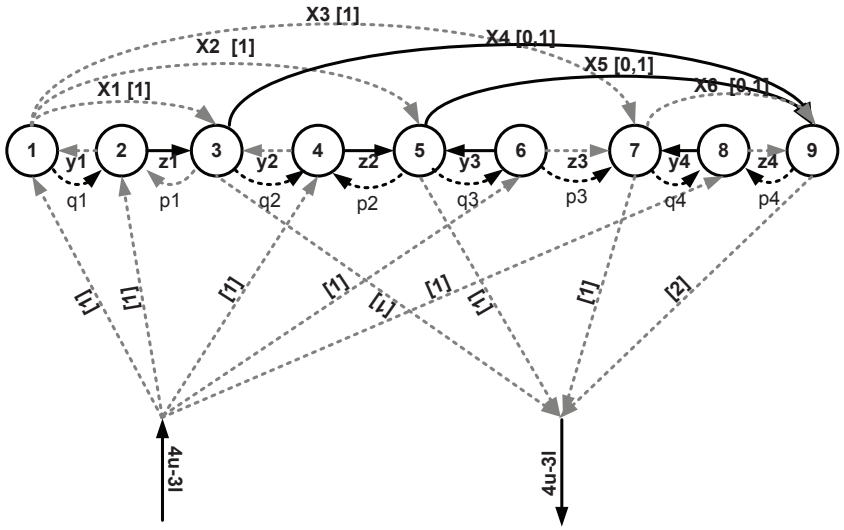
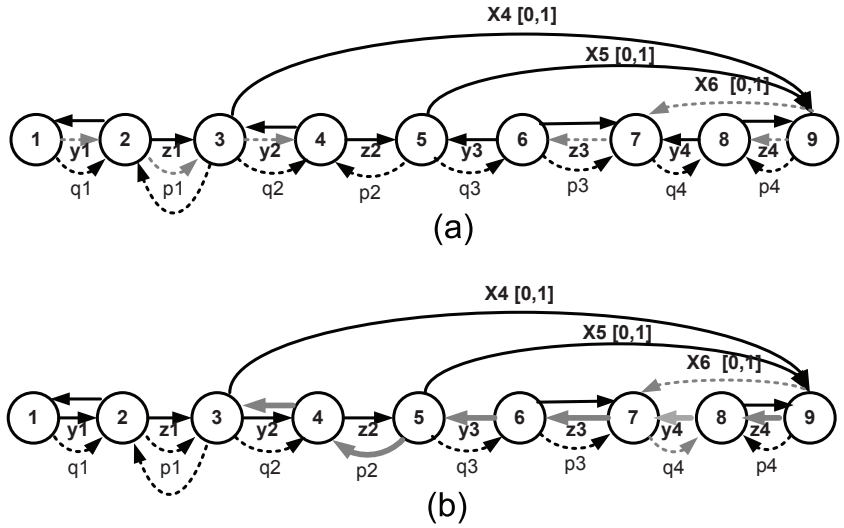


Figure 4.15 The residual graph for the flow at Figure 4.14



cost flow in the corresponding graph is shown in Figure 4.3 (gray dashed edges). Note that the cost of the flow is 1 *as the first* AMONG *constraint is violated by one unit. Hence, the lower bound of the variable* T *has to be changed to* 1.

The residual graph that corresponds to the flow in Figure 4.14 is shown in Figure 4.15(a). The reversed edges are highlighted in gray dashed.

Consider the variable X_4. *The min cost flow* f *does not send a unit of flow through the corresponding edge* e_{X_4}. *Therefore, the value* 0 *is supported. Consider a support for the value* 1. *We find the shortest path from the vertex* 9 *to the vertex* 3. *The weight of this path is* 1 *(Figure 4.15(b), thick gray edges). If we redirect the flow through the edge* e_{X_4} *then the min cost flow is* 2 *which violates constraint. Therefore, the value* 1 *can be removed form* $D(X_4)$. $\diamond$

4.4.3 The Soft GEN-SEQUENCE constraint

A soft form of the GEN-SEQUENCE constraints is a relaxation of the GEN-SEQUENCE constraint. The soft GEN-SEQUENCE constraint takes into account by how much each AMONG constraint is violated. The soft GEN-SEQUENCE $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}], T)$ introduces a violation variable T and is defined as follow.

$$T \geq \sum_{i=1}^{m} \max(l_i - \sum_{j=s_i}^{s_i+k_i-1} X_j, \sum_{j=s_i}^{s_i+k_i-1} X_j - u_i, 0) \tag{4.23}$$

As before, we consider soft GEN-SEQUENCE on Boolean variables as this does not hinder propagation to simplify notations.

Example 4.16 (Running example (Soft GEN-SEQUENCE)) *Consider a relaxation of Example 4.6. Suppose we allow to violate individual* AMONG *constraints but we restrict the total number of violations to be* 1. *Hence, the domain of the cost variable* T *is* $\{0, 1\}$. $\diamond$

To express the soft GEN-SEQUENCE constraint as a linear program, we introduce penalty variables for each inequality associated to the hard GEN-SEQUENCE, namely, Q_i and P_i, $i = 1, \ldots, m$ and minimise the sum of penalty variables. We show the reformulation using the running example 4.16.

$$\begin{aligned}
Minimise \sum_{i=1}^{4}(P_i + Q_i) \qquad & \text{subject to :} \\
l \leq X_1 + X_2 + X_3 + Q_1, \qquad & -u \leq -X_1 - X_2 - X_3 + P_1, \\
l \leq X_2 + X_3 + X_4 + Q_2, \qquad & -u \leq -X_2 - X_3 - X_4 + P_2, \\
l \leq X_3 + X_4 + X_5 + Q_3, \qquad & -u \leq -X_3 - X_4 - X_5 + P_3, \\
l \leq X_1 + X_2 + X_3 + X_4 + X_5 + Q_4, \qquad & -u \leq -X_1 - X_2 - X_3 - X_4 - X_5 + P_4 \\
0 \leq X_i,\ -1 \leq -X_i, \qquad & 0 \leq Q_i,\ 0 \leq P_i
\end{aligned}$$

In matrix form, this is:

$$Minimise \sum_{i=1}^{4}(P_i + Q_i) \tag{4.24}$$

$$A\vec{X} \geq \vec{b}, \tag{4.25}$$

$$A = \begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & -1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
-1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

where

$$\vec{X} = \Big(X_1, X_2, X_3, X_4, X_5, Q_1, P_1, \ldots, Q_4, P_4\Big)$$

$$\vec{b} = \Big(l, -u, l, -u, l, -u, l, -u, 0, \ldots, 0, -1, \ldots, -1, 0, \ldots, 0, 0, \ldots, 0\Big)$$

The dual formulation of the problem is :

$$\begin{aligned} Maximise \quad & -b^T y \\ & A^T y = \vec{c} \\ & y \geq 0. \end{aligned}$$

where

$$A^T = \begin{pmatrix}
1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

$$c^T = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The dual problem can be transformed using row operations to obtain the consecutive ones property on the columns of the matrix. Note that the first five rows of the matrix already have the consecutive ones property. For each of the first 2×5 columns, one needs to obtain ones between the last entry set to 1 (-1) in the first 5 columns and the single 1 (-1 that can be obtained by multiplying the corresponding row by -1) under these rows. This is done by selecting the row with a single $1(-1)$ whose corresponding column is set to $1(-1)$ and adding this row to every row above until the consecutive ones property is reached on this column. So we obtain the following matrix:

$$A^T = \begin{pmatrix}
1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\
0 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & -1 & 1 \\
0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 \\
0 & 0 & 0 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{pmatrix}$$

Using the same technique for the SEQUENCE constraint, we obtain a system that can be solved using a network flow algorithm. Figure 4.16 shows the flow graph for the running example 4.16. Note that the flow graphs for the hard and soft GEN-SEQUENCE constraints have a very similar structure. Consider the flow graph for the GEN-SEQUENCE constraint in the running example (Figure 4.8). It includes arcs that correspond to original variables X and are labelled 0 or 1, $i = 1, \ldots, n$ and arcs that correspond to linear inequalities and are labelled $-l$ or u. The flow graph for the soft GEN-SEQUENCE constraint contains the same arcs for variables, however, each of the inequality arcs is split into two arcs by introducing a node with unit demand or supply[1].

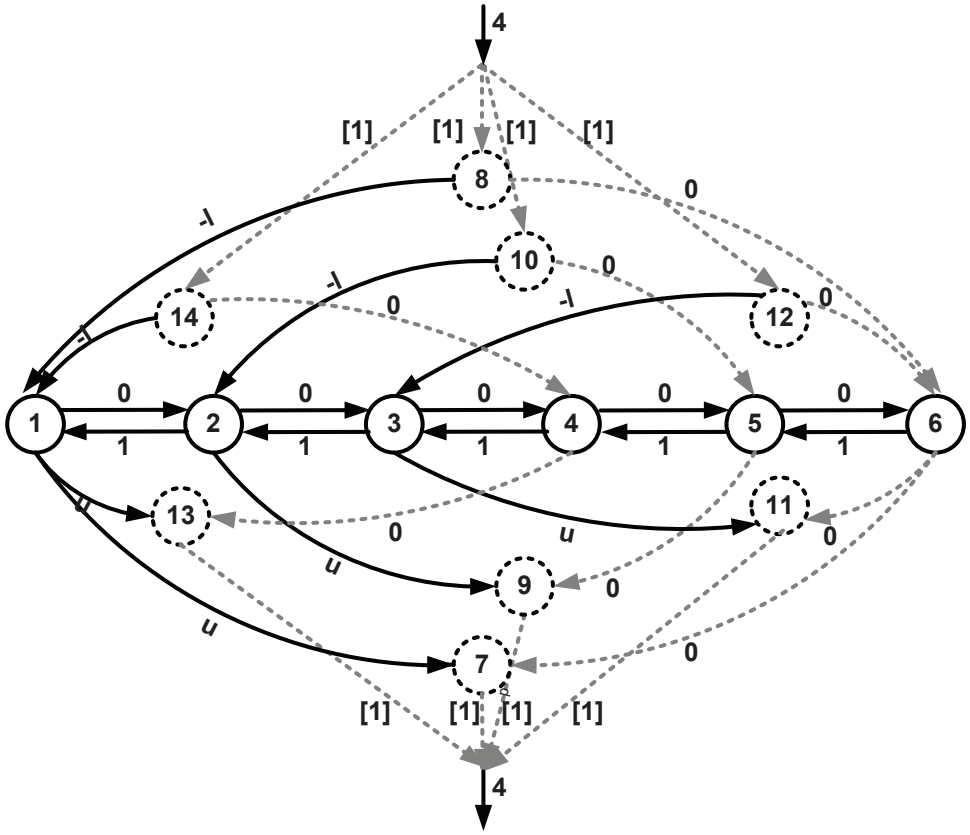
This leads to a difference between the two flow graphs: the flow graph for the GEN-SEQUENCE constraint has zero flow circulation, while the flow graph for the soft GEN-SEQUENCE constraint contains a flow of value $m = 4$. Note that the capacity of each edge connecting either the source or the sink to other nodes is exactly 1. However, the capacities of the other edges in the flow network are not bounded. Therefore, these edges can carry several units of flow in a feasible minimum cost flow, which makes a flow-based *DC* propagator for the soft GEN-SEQUENCE constraint more computationally expensive compared to the hard case. The flow graph for the soft GEN-SEQUENCE constraint has $O(n + m)$ nodes and $O(n + m)$ edges. The minimal cost flow can be found in $O(|V| \log |E|(|E| + |V| \log |V|) = O((n + m)^2 \log^2(n + m))$ time [AMO93]. The *DC* filtering algorithm for the soft GEN-SEQUENCE constraint works exactly the same as for the soft SEQUENCE constraint (Section 4.4.2), except that finding all pairs of shortest paths is replaced with finding all pairs of minimal cost flows. Hence, the total time complexity of the flow-based *DC* filtering algorithm is $O(n(n + m)^2 \log^2(n + m))$.

4.4.4 The SLIDINGSUM constraint

The SLIDINGSUM constraint [BCR05] is a further generalisation of the GEN-SEQUENCE constraint from Boolean to integer variables, which we extend to allow arbitrary windows.

[1] Note that, in contrast to the soft SEQUENCE constraint, this flow graph is not obtained by the methodology of [HPR06].

Figure 4.16 Network flow associated with the soft GEN-SEQUENCE constraint posted on the running example. The edge capacities are written in square brackets [] to differentiate them from the edge costs. Gray edges show a possible flow in the network.



We recall the definition of the constraint. SLIDINGSUM $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}])$ holds if $l_i \leq \sum_{j=s_i}^{s_i+k_i-1} X_i \leq u_i$ holds for $1 \leq i \leq n-k+1$ where $\vec{p_i} = \langle l_i, u_i, k_i, s_i \rangle$ is, as with the generalised SEQUENCE, a window.

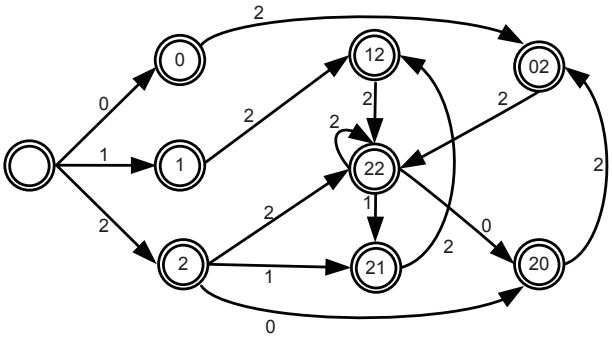
A bounds consistency propagator for this constraint can be constructed the same way as the propagator for the GEN-SEQUENCE constraint. The only difference is that instead of constraint (4.16) with right hand sides 0 or 1 we have a set of constraints:

$$a_i \leq X_i, \;\; -b_i \leq -X_i, i = 1, \ldots, n$$

The proof of the reformulation is identical to the proofs for the GEN-SEQUENCE constraint as well as complexity guaranties.

Similarly, a bounds consistency propagator for the soft GEN-SEQUENCE constraint can be constructed in the same way as propagator for the GEN-SEQUENCE constraint.

Figure 4.17 The automaton for the Multiple SEQUENCE constraint with 2 SEQUENCEs: SEQUENCE$(0, 1, 2, [X_1, \ldots, X_n], \{1\})$ and SEQUENCE$(2, 3, 3, [X_1, \ldots, X_n], \{2\})$, $D(X_i) = \{0, 1, 2\}$, $i = 1, \ldots, n$



4.4.5 The Multiple SEQUENCE constraint

We often have multiple SEQUENCE constraints applied to the same sequence of variables. For instance, we might insist that at most 1 in 3 cars have the sun roof option and simultaneously that at most 2 in 5 of those cars have electric windows. We propose an encoding for enforcing domain consistency on the conjunction of m such SEQUENCE constraints (we shall refer to this as MR). Suppose that the jth such constraint is SEQUENCE$(l_j, u_j, k_j, [X_1, \ldots, X_n], v_j)$. For simplicity, we suppose that the values being counted are disjoint. The extension to the non-disjoint case is straightforward but notationally messy. We channel into a new sequence of variables Y_i where $Y_i = j$ if $X_i \in v_j$ else $Y_j = 0$. We now construct an automaton whose states record the last $k' - 1$ values used where k' is the largest k_j. Transitions of the automaton ensure that all SEQUENCE constraints are satisfied. Therefore, domain consistency can be enforced using the REGULAR constraint in $O(nm^{k'-1})$ time. The automaton for the Multiple SEQUENCE constraint with 2 SEQUENCEs is presented in Figure 4.17.

Next we show that enforcing even bounds consistency of the Multiple SEQUENCE constraint is NP-hard.

Theorem 4.10 *Enforcing BC on the Multiple* SEQUENCE *constraint is NP-hard.*

Proof: We do a reduction from a 1-in-3 *SAT* problem on positive clauses. The decision 1-in-3 *SAT* problem is to decide whether there exists a satisfying assignment for a *SAT* formula over positive literals such that there is exactly one true literal in each clause.

Consider a problem $\phi(X)$ with N variables and M clauses. The Multiple SEQUENCE constraint has $2NM$ integer variables with domains $\{0, 1, 2, 3\}$. The Multiple Sequence constraint consists of three

SEQUENCE constraints: SEQUENCE$(N, N, 2N, [X_1, \ldots, X_{2NM}], \{1, 2\})$, SEQUENCE$(0, 1, N, [X_1, \ldots, X_{2NM}], \{2\})$ and SEQUENCE$(M, M, 2NM, [X_1, \ldots, X_{2NM}], \{2\})$. The sequence of variables includes M blocks of length $2N$. The first N variables of a block of $2N$ variables represent an assignment and the second N variables are dummy variables. If X_i, $i = 1, \ldots, N$ equals 1 or 2 then the ith Boolean variable is true, otherwise the Boolean variable is false.

Each block of $2N$ variables corresponds to a clause. Let the jth clause be $x_1 \vee x_4 \vee x_5$. Then domains of variables $X_{2N(j-1)+i} = \{0, 1\}$, $i = (1, \ldots, 2N) \wedge (i \notin \{1, 4, 5\})$. $X_{2N(j-1)+i} = \{2, 3\}$, $i \in \{1, 4, 5\}$.

The first SEQUENCE constraint ensures that there is a sliding assignment along the sequence of variables. If a variable X_i takes value 1 or 2 in the first block then X_{i+2Nj} takes value 1 or 2 in the jth block, $j = 1, \ldots, M$. If a variable X_i takes value 0 or 3 in the first block then X_{i+2Nj} takes value 0 or 3 in the jth block, $j = 1, \ldots, M$. The second SEQUENCE constraint ensures that at most one variable in each block takes value 2 and the third SEQUENCE constraint forces at least one variable in each block to take value 2. Therefore there is exactly one variable in each block that takes value 2. Consecutively, the only Boolean variable in each clause takes the value true. The Multiple SEQUENCE constraint is $bounds\ consistent$ if $\phi(X)$ has a satisfying assignment. $\diamond$

4.5 Other related work

The SEQUENCE constraint was introduced by Beldiceanu and Contejean [BC94]. In addition to the domain consistency algorithm for the SEQUENCE constraint that we described in Sections 4.2.1– 4.2.2, several incomplete algorithms have been proposed. Beldiceanu and Carlsson suggested a greedy filtering algorithm for the CARDPATH constraint that can be used to propagate the SEQUENCE constraint [BC01]. Regin and Puget proposed a filtering algorithm for the Global Sequencing constraint (GSC) that combines a SEQUENCE and a global cardinality constraint (GCC) [RP97]. Regin decomposed GSC into a set of variable disjoint AMONG and GCC constraints [Reg05]. This encoding is equivalent to the AD decomposition if we use it to propagate the SEQUENCE constraint.

4.6 Experimental results

To compare the performance of the different algorithms and encodings, we carried out a number of experiments. Our setup is similar to the set up in [HPRS] and consists of two

sets of experiments. The first series of experiments is on a single SEQUENCE constraint. The aim of these experiments to test 'pure' speed of the filtering algorithms and decompositions. Hence, we randomly generated instances so we could control the parameters precisely. The second series of experiments uses nurse rostering benchmarks to test more realistic situations. The aim of these experiments is to test an interaction of the SEQUENCE constraint with other constraints.

Experiments were run with ILOG Solver 6.1 on an Intel Pentium 4 CPU 3.20Ghz, 1G RAM. Boost graph library version $1.34.1$ was used to implement incremental flow-based algorithms for the SEQUENCE and SOFTSEQUENCE constraints. The current flow was stored in a set of backtrackable integers provided by ILOG solver. On each backtrack we have to update the current network flow in the Boost data structure that represents the flow graph. This introduces a linear overhead on backtracking. Note that this overhead can be avoided if we implement backtrackable data-structures for graphs and implement network flow algorithms using these date-structures..

We recall that Table 4.1 gives a summary of the decompositions and filtering algorithms for the SEQUENCE constraint that we consider in these experiments.

4.6.1 Random instances

For each possible combination of $n \in \{100, 250, 500, 1000, 3000, 5000, 6000\}$, $k \in \{7, 15, 50\}$, $\Delta = u - l \in \{1, 3, 5\}$, we generated twenty instances with random lower bounds in the interval $[0, k - \Delta)$. We used random value and variable orderings and a time-out of 300 sec. Results for different values of Δ are presented in Tables 4.2- 4.3 for $n \in \{100, 250, 500, 1000, 3000, 5000, 6000\}$. For all experimental results we show the number of solved instances and cpu time averaged by the number of solved instances in seconds. Note that in the summary part of all tables we show the average time and the average number of backtracks over solved instances only.

Results for different values of Δ are presented in Figures 4.18–4.19 for all values of n. We compare 6 algorithms and decompositions for the SEQUENCE constraint.

Instances can be partitioned into 2 groups based on the hardness of the instance. In the first group $\Delta = 1$. On these instances, assignment of one variable has a strong impact on other variables. In the extreme case when $\Delta = 0$ instantiation of one variable assigns on average another n/k variables. So, from a theoretical point of view, we expect DC propagators to significantly shrink variable domains and reduce the search tree. As can be seen from Table 4.2, *DC* propagators outperform non-*DC* propagators. Surprisingly, CS

Table 4.2: Randomly generated instances with a single SEQUENCE constraint and $\Delta = 1$. Number of instances solved in 300 sec / average time to solve.

n	k	PS		$HPRS$		AD		LG		CS		FB	
100	7	**20**	/ 0.01	**20**	/ 0.01	**20**	/ 14.05	**20**	/ **0.00**	**20**	/ 0.00	**20**	/ 0.01
	15	**20**	/ 0.04	**20**	/ 0.01	16	/ 9.21	17	/ 4.47	**20**	/ **0.00**	**20**	/ 0.01
	50	**20**	/ 1.47	**20**	/ 0.01	18	/ 17.01	18	/ 4.71	**20**	/ **0.00**	**20**	/ 0.01
250	7	**20**	/ **0.02**	**20**	/ 0.05	14	/ 0.66	18	/ 1.68	**20**	/ 0.03	**20**	/ 0.08
	15	**20**	/ 0.12	**20**	/ 0.05	3	/ 0.04	4	/0	**20**	/ **0.02**	**20**	/ 0.08
	50	**20**	/ 5.06	**20**	/ 0.05	4	/ 11.85	4	/ 0.33	**20**	/ **0.02**	**20**	/ 0.07
500	7	**20**	/ **0.04**	**20**	/ 0.35	8	/ 2.20	12	/ 4.14	**20**	/ 0.13	**20**	/ 0.30
	15	**20**	/ 0.30	**20**	/ 0.31	6	/ 0.01	6	/ 0.01	**20**	/ **0.09**	**20**	/ 0.30
	50	**20**	/ 15.86	**20**	/ 0.26	2	/ 0.03	2	/ 0.01	**20**	/ **0.07**	**20**	/ 0.28
1000	7	**20**	/ **0.09**	**20**	/ 2.36	4	/ 0.01	4	/ 0.02	**20**	/ 0.71	**20**	/ 1.18
	15	**20**	/ 0.64	**20**	/ 2.06	2	/ 0.59	3	/ 0.02	**20**	/ **0.38**	**20**	/ 1.17
	50	**20**	/ 28.47	**20**	/ 1.48	1	/0	1	/0	**20**	/ **0.28**	**20**	/ 1.14
3000	7	**20**	/ **0.35**	**20**	/ 64.04	3	/ 0.07	8	/ 0.08	**20**	/ 15.14	**20**	/ 10.44
	15	**20**	/ **2.24**	**20**	/ 51.04	1	/0	1	/0	**20**	/ 5.49	**20**	/ 11.90
	50	0	/0	**20**	/ 35.48	0	/0	0	/0	**20**	/ **2.61**	**20**	/ 10.12
5000	7	**20**	/ **0.66**	15	/ 262.17	1	/0	8	/ 0.20	**20**	/ 64.05	**20**	/ 36.09
	15	**20**	/ **3.76**	17	/ 211.17	0	/0	1	/0	**20**	/ 24.46	**20**	/ 34.59
	50	0	/0	19	/ 146.63	0	/0	0	/0	**20**	/ **8.24**	**20**	/ 31.66
6000	7	**20**	/ **0.87**	0	/0	0	/0	2	/ 0.25	**20**	/ 95.64	**20**	/ 55.56
	15	**20**	/ **4.35**	5	/ 280.15	0	/0	0	/0	**20**	/ 36.83	**20**	/ 50.09
	50	0	/0	12	/ 193.41	0	/0	0	/0	**20**	/ **12.54**	**20**	/ 47.73
TOTALS													
solved/total		360	/420	368	/420	103	/420	129	/420	**420**	/420	**420**	/420
avg time for sol		3.576		46.688		7.867		1.900		12.700		13.943	
avg bt for sol		0		0		208033		38993		378		0	

Table 4.3: Randomly generated instances with a single SEQUENCE constraint and $\Delta = 3$. Number of instances solved in 300 sec / average time to solve.

n	k	PS		$HPRS$		AD		LG		CS		FB	
100	7	**20**	/ 0.01	**20**	/ 0.00	**20**	/ **0.00**	**20**	/ **0.00**	**20**	/ 0.01	**20**	/ 0.02
	15	**20**	/ 0.04	**20**	/ **0.00**	17	/ 0.00	19	/ 0.07	**20**	/ 0.01	**20**	/ 0.02
	50	**20**	/ 1.49	**20**	/ **0.00**	19	/ 1.37	19	/ 0.00	**20**	/ 0.01	**20**	/ 0.01
250	7	**20**	/ 0.02	**20**	/ 0.04	**20**	/ 0.00	**20**	/ **0.00**	**20**	/ 0.08	**20**	/ 0.10
	15	**20**	/ 0.13	**20**	/ **0.05**	18	/ 0.01	19	/ 0.01	**20**	/ 0.05	**20**	/ 0.10
	50	**20**	/ 5.18	**20**	/ 0.04	1	/0	3	/ 0.01	**20**	/ **0.03**	**20**	/ 0.09
500	7	**20**	/ 0.04	**20**	/ 0.27	**20**	/ **0.01**	**20**	/ 0.01	**20**	/ 0.55	**20**	/ 0.37
	15	**20**	/ 0.32	**20**	/ 0.45	17	/ 0.01	**20**	/ **0.01**	**20**	/ 0.26	**20**	/ 0.37
	50	**20**	/ 13.04	**20**	/ 0.29	2	/ 0.02	2	/ 0.02	**20**	/ **0.13**	**20**	/ 0.36
1000	7	**20**	/ 0.10	**20**	/ 2.19	**20**	/ **0.02**	**20**	/ 0.03	**20**	/ 3.83	**20**	/ 1.48
	15	**20**	/ **0.73**	**20**	/ 2.69	15	/ 0.02	16	/ 0.03	**20**	/ 1.98	**20**	/ 1.45
	50	**20**	/ 24.37	**20**	/ 2.57	5	/ 0.05	6	/ 0.04	**20**	/ **0.60**	**20**	/ 1.42
3000	7	**20**	/ 0.41	**20**	/ 51.08	**20**	/ **0.12**	**20**	/ 0.15	**20**	/ 105.78	**20**	/ 13.43
	15	**20**	/ **2.42**	**20**	/ 73.55	9	/ 0.16	14	/ 0.18	**20**	/ 41.74	**20**	/ 13.00
	50	0	/0	**20**	/ 59.02	3	/ 0.19	3	/ 0.17	**20**	/ **6.93**	**20**	/ 12.91
5000	7	**20**	/ 0.81	14	/ 201.29	**20**	/ **0.31**	**20**	/ 0.34	6	/ 144.07	**20**	/ 38.47
	15	**20**	/ **3.42**	13	/ 235.41	11	/ 0.31	14	/ 0.39	12	/ 139.79	**20**	/ 39.02
	50	0	/0	13	/ 213.87	0	/0	1	/0	**20**	/ **32.16**	**20**	/ 37.79
6000	7	**20**	/ 1.06	0	/0	**20**	/ **0.46**	**20**	/ 0.54	5	/ 283.36	**20**	/ 59.77
	15	**20**	/ **4.02**	0	/0	6	/ 0.47	7	/ 0.50	7	/ 150.46	**20**	/ 57.56
	50	0	/0	5	/ 232.23	5	/ 0.69	5	/ 0.61	**20**	/ **46.43**	**20**	/ 55.84
TOTALS													
solved/total		360	/420	345	/420	268	/420	288	/420	370	/420	**420**	/420
avg time for sol		3.200		39.608		0.214		0.137		26.550		15.884	
avg bt for sol		0		0		1133		117		493		**0**	

Table 4.4: Randomly generated instances with a single SEQUENCE constraint and $\Delta = 5$. Number of instances solved in 300 sec / average time to solve.

n	k	PS		$HPRS$		AD		LG		CS		FB	
100	7	**20**	/ 0.00	**20**	/ 0.00	**20**	/ **0.00**	**20**	/ **0.00**	**20**	/ 0.01	**20**	/ 0.02
	15	**20**	/ 0.05	**20**	/ 0.00	**20**	/ **0.00**	**20**	/ **0.00**	**20**	/ 0.01	**20**	/ 0.02
	50	**20**	/ 1.44	**20**	/ 0.01	**20**	/ 0.00	**20**	/ **0.00**	**20**	/ 0.01	**20**	/ 0.01
250	7	**20**	/ 0.02	**20**	/ 0.02	**20**	/ **0.00**	**20**	/ 0.00	**20**	/ 0.09	**20**	/ 0.12
	15	**20**	/ 0.13	**20**	/ 0.04	**20**	/ **0.01**	**20**	/ 0.01	**20**	/ 0.10	**20**	/ 0.11
	50	**20**	/ 4.92	**20**	/ 0.06	17	/ 0.01	17	/ 0.01	**20**	/ **0.03**	**20**	/ 0.11
500	7	**20**	/ 0.04	**20**	/ 0.15	**20**	/ **0.01**	**20**	/ 0.01	**20**	/ 0.58	**20**	/ 0.44
	15	**20**	/ 0.35	**20**	/ 0.25	**20**	/ **0.01**	**20**	/ **0.01**	**20**	/ 0.69	**20**	/ 0.45
	50	**20**	/ 12.06	**20**	/ 0.37	18	/ 0.02	19	/ 0.02	**20**	/ **0.20**	**20**	/ 0.42
1000	7	**20**	/ 0.09	**20**	/ 0.99	**20**	/ **0.03**	**20**	/ 0.03	**20**	/ 4.33	**20**	/ 1.70
	15	**20**	/ 0.64	**20**	/ 1.83	**20**	/ **0.03**	**20**	/ 0.04	**20**	/ 4.68	**20**	/ 1.70
	50	**20**	/ 30.14	**20**	/ 2.73	10	/ 0.05	9	/ 0.05	**20**	/ **1.24**	**20**	/ 1.69
3000	7	**20**	/ 0.41	**20**	/ 23.85	**20**	/ **0.14**	**20**	/ 0.16	**20**	/ 104.68	**20**	/ 14.96
	15	**20**	/ 2.53	**20**	/ 44.67	**20**	/ **0.16**	**20**	/ 0.20	**20**	/ 125.11	**20**	/ 15.21
	50	0	/0	**20**	/ 66.61	5	/ 0.29	7	/ 0.31	**20**	/ 22.73	**20**	/ **14.61**
5000	7	**20**	/ 0.85	**20**	/ 109.18	**20**	/ **0.36**	**20**	/ 0.42	0	/0	**20**	/ 46.42
	15	**20**	/ 4.15	17	/ 215.97	**20**	/ **0.36**	**20**	/ 0.47	6	/ 160.99	**20**	/ 45.97
	50	0	/0	11	/ 210.53	9	/ 0.48	10	/ 0.46	**20**	/ 108.34	**20**	/ **44.88**
6000	7	**20**	/ 1.08	**20**	/ 185.73	**20**	/ **0.50**	**20**	/ 0.55	0	/0	**20**	/ 74.01
	15	**20**	/ 4.90	6	/ 281.65	**20**	/ **0.51**	**20**	/ 0.63	3	/ 152.83	**20**	/ 70.70
	50	0	/0	10	/ 254.33	5	/ 0.57	6	/ 0.65	18	/ 181.74	**20**	/ **69.84**
TOTALS													
solved/total		360	/420	384	/420	364	/420	368	/420	347	/420	**420**	/420
avg time for sol		3.544		49.350		0.144		0.169		35.020		19.209	
avg bt for sol		0		0		1		1		769		**0**	

has the best time of all combinations and solved all instances up to the size of the sequence 3000. After that CS loses to PS or FB algorithms. Whilst it takes more backtracks compared to the DC propagators which solve problems without search, it is much faster. The PS algorithm is faster compared to other *DC* algorithms on instances with small k and much slower on instances with larger $k \geq 50$. Moreover, its relative performance decays for larger k's. The FB algorithm performs better compared to the $HPRS$ algorithm and it scales better. It is also the only propagator that solves all instances for $\delta \in \{1, 3, 5\}$. The AD and LG decompositions do not perform well on these instances. They cannot solve most of the benchmarks especially for large values of n. Note that the LG decomposition performs slightly better compared to the AD decomposition.

In the second group $\Delta \geq 3$. On these instances, assignment of one variable does not have a big influence on other variables. The overhead of using DC propagators to achieve better pruning outweighs the reduction in the search space. For the instance $\Delta = 3$, the *DC* propagators still outperform the non-*DC* propagators on large values of n or k. The clear winner in the $\Delta = 5$ case are those propagators which do not achieve *DC*. When $k \leq 15$ AD is best. When k gets larger, LG solves more instances and works faster due to its better propagation.

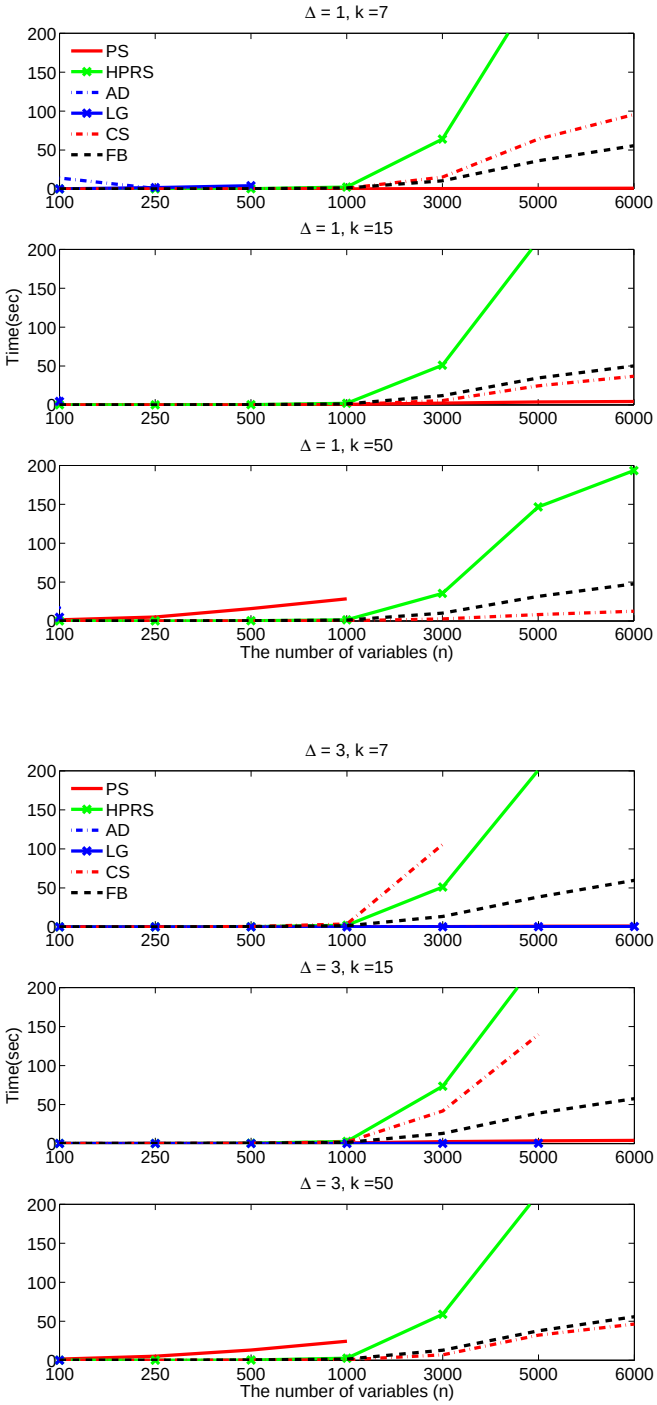
It should be noted that the FB propagator is not the fastest one but has the most robust performance. It is sensitive only to the value of n and not to other parameters, like the length of the window(k) or hardness of the problem(Δ). As can be seen from Figure 4.18- 4.19, the FB propagator scales better than the other propagators with the size of the problem. It appears to grow quadratically with the number of variables, while the $HPRS$ propagator displays cubic growth.

We do not present the results for the LO decomposition in this section because the size of the automaton for $k \in \{15, 50\}$ is too large.

4.6.2 Nurse Rostering Problems

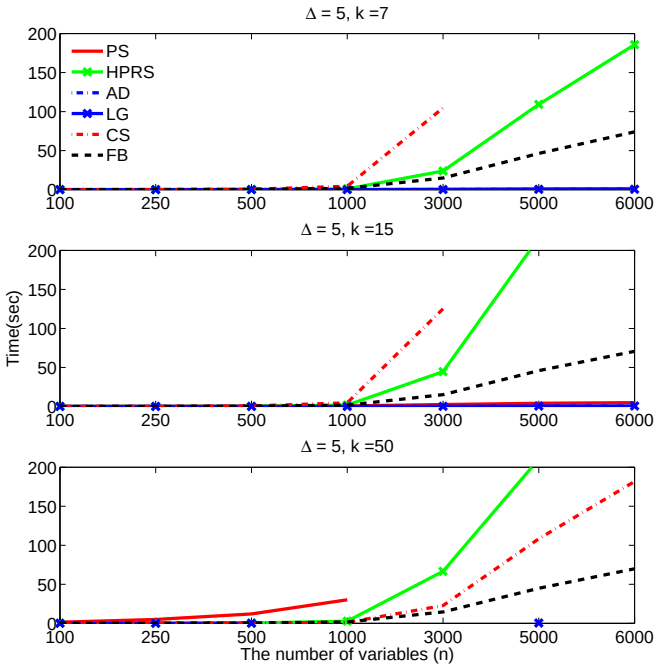
All instances are taken from `http://www.projectmanagement.ugent.be/nsp.php`. The basic model includes the following three constraints: each shift has a minimum required number of nurses, each nurse should have at least 12 hours of break between 2 shifts, each nurse should have at least two consecutive days on any shift. In addition, each model has an additional SEQUENCE constraint that is specified in the first column in the table of results. For each day in the scheduling period, a nurse is assigned to a day (D), evening (E), or night (N) shift or takes a day-off (O). We introduce one variable $X[i, j]$ for

Figure 4.18 Randomly generated instances with a single SEQUENCE constraints for different combinations of $\Delta = \{1, 3\}$ and k.



each nurse, $i = 1 \ldots p$ and each day, $j = 1 \ldots n$, where p is the number of nurses, n is the number of days in the scheduling period. Each model was run on 50 instances. The scheduling period is 28 days. The number of nurses in each instance was set to the maximal number of nurses required for any day over the period of 28 days. The time limit for all

Figure 4.19 Randomly generated instances with a single SEQUENCE constraint for different combinations of $\Delta = \{5\}$ and k.



instances was 900 sec. For variable ordering, we branched on the smallest domain.

Table 4.5 gives results for those instances that were solved by each propagator. In these experiments $n < 50$. As expected from the random experiments, the AD and LG decompositions outperform all other decompositions. The only exception are instances with $\Delta = 0$ where non-DC propagators lose to DC algorithms and the CS decomposition (third and forth rows).

4.6.3 Multiple SEQUENCE instances

We also evaluated the performance of the different propagators on problems with multiple SEQUENCE constraints. We again used randomly generated instances and nurse rostering problems. For each possible combination of $n \in \{50, 100\}$, $k \in \{5, 7\}$, $\Delta = 1$, we generated twenty random instances of Multiple SEQUENCE with 4 SEQUENCEs. All variables had domains of size 5. An instance was obtained by selecting random lower bounds in the interval $[0, k - \Delta]$. We excluded instances where $\sum_{i=1}^{m} l_i \geq k$ to avoid unsatisfiable instances. We used a random variable and value ordering, and a time-out of 300 sec. All SEQUENCE constraints were enforced on disjoint sets of cardinality one.

Experimental results are presented in Table 4.6. They show that the multiple SEQUENCE propagator significantly outperforms other propagators on random benchmarks

Table 4.5: Models of the nurse rostering problem using the SEQUENCE constraint. Number of instances solved in 900 sec / average time to solve.

SEQUENCE	PS	LO	$HPRS$	AD	LG	CS	FB
$(1,3,3,\{O\})$	**46** / 7.78	**46** / 9.06	**46** / 9.74	**46** / **7.59**	**46** / 7.66	39 / 50.32	**46** / 22.47
$(3,5,5,\{O\})$	**48** / 10.02	**48** / 8.97	**48** / 13.58	**48** / 8.00	**48** / **7.61**	39 / 46.49	**48** / 32.91
$(2,2,5,\{O\})$	**46** / 8.41	**46** / **7.78**	**46** / 17.33	41 / 13.87	41 / 14.07	44 / 32.75	44 / 36.47
$(2,2,7,\{O\})$	23 / 132.47	**24** / 100.27	21 / 56.71	18 / 18.46	18 / **18.27**	19 / 25.98	21 / 176.50
$(2,3,5,\{O\})$	26 / **38.98**	26 / 44.83	27 / 67.13	**28** / 74.02	**28** / 75.30	22 / 84.40	25 / 49.23
$(2,5,7,\{O\})$	12 / 18.07	12 / 17.64	**13** / 78.03	**13** / 70.28	**13** / 72.20	12 / **13.70**	12 / 22.25
$(1,3,4,\{O\})$	**26** / 53.93	25 / 30.73	**26** / 53.50	**26** / 44.53	**26** / 46.12	20 / 38.36	24 / **24.49**
TOTALS							
solved/total	**227** /350	**227** /350	**227** /350	220 /350	220 /350	195 /350	220 /350
avg time for solved	30.418	**25.362**	32.186	26.263	26.680	43.582	45.501
avg bt for solved	104949	92401	149198	175855	175855	192618	**42562**

Table 4.6: Randomly generated instances with 4 SEQUENCE constraints and $\Delta = 1$. Number of instances solved in 300 sec / average time to solve.

n	k	MR	PS	$HPRS$	AD	LG	CS	FB	LO
50	5	**20** / **0.04**	13 / 15.68	13 / 22.40	13 / 15.09	13 / 9.78	1 /0	11 / 40.14	13 / 18.93
	7	**20** / **0.56**	6 / 19.50	6 / 14.64	7 / 14.69	7 / 5.98	1 /0	5 / 6.39	6 / 15.37
100	5	**20** / **0.39**	0 /0	0 /0	0 /0	0 /0	0 /0	0 /0	0 /0
	7	**20** / **1.36**	2 / 0.03	2 / 0.03	2 / 29.50	2 / 1.56	1 /0	2 / 0.05	2 / 0.02
TOTALS									
solved/total		**80** /80	21 /80	21 /80	22 /80	22 /80	3 /80	18 /80	21 /80
avg time for sol		0.589	15.280	18.051	16.270	7.824	0	26.307	16.110
avg bt for sol		**0**	115758	115758	305610	152058	0	26923	115758

in both metrics, namely time to find a valid sequence and the number of solved instances. For bigger values of n, the multiple SEQUENCE is the only filtering algorithm that successfully solved all instances. However, it should be noted that, due to its space complexity, to use this propagator successfully, k and m should be relatively small and n should be less than 100.

In the second series of experiments we used nurse scheduling problems benchmarks. We removed the last constraint from the basic model described in the previous section and added two sets of SEQUENCE constraints to give two different models. In the first model, each nurse has to work one or two night shifts in 7 consecutive days, one or two evening shifts, one to five day shifts and have two to five days-off. In the second model, each nurse has to work one or two night shifts in 7 consecutive days, and have one or two days-off in 5 days. In order to measure the performance of the Multiple SEQUENCE constraint on large problems, we built a schedule over a 28 day period. The number of nurses was equal to maximal number of nurses required for any day over the period multiplied by 1.5. The total number of variables in an instance is about 500. Table 4.7 shows the number of instances solved by each propagator. The multiple SEQUENCE propagator solved more instances compared to the other propagators.

4.6.4 The Soft SEQUENCE constraint

We evaluated performance of the soft SEQUENCE constraint on random problems. For each possible combination of $n \in \{50, 100\}$, $k \in \{5, 15, 25\}$, $\Delta = \{1, 5\}$ and $m \in \{4\}$ (where m is the number of SEQUENCE constraints), we generated twenty random instances the same way as in Section 4.6.3. Instances with this set of parameters are hard instances for SEQUENCE propagators as less than 22% of instances were solved by any of domain consistency propagators. To relax these instances, we allow violating the SEQUENCE constraint with a cost that has to be less than or equal to 15% of the length of the sequence. Experimental results are presented in Table 4.8. As can be seen from the table, the FB_S algorithms is competitive with the decomposition into soft AMONG constraints on easy problems and outperforms the decomposition on hard problems.

We observed that the FB_S propagator is very slow for the soft SEQUENCE constraint. Note that the number of backtracks of FB_S is three orders of magnitude smaller compared to AD_S. We profiled the algorithm and found that it spends most of the time performing the all pairs shortest path algorithm. Unfortunately, this is difficult to compute incrementally because the residual graph can be different on every invocation of the propagator.

Table 4.7: Models of the nurse rostering problem using the SEQUENCE constraint. Number of instances solved in 300 sec / average time to solve.

	MR	PS	$HPRS$	AD	LG	CS	FB	LO
Model 1	**10** / 12.91	4 / 10.00	4 / 6.26	4 / **5.20**	5 / 60.48	5 / 52.20	4 / 27.09	4 / 11.22
Model 2	**8** / 5.66	4 / 0.07	4 / **0.04**	4 / **0.03**	4 / **0.03**	4 / **0.03**	4 / 0.08	4 / 0.08
TOTALS								
solved/total	**18** /100	8 /100	8 /100	8 /100	9 /100	9 /100	8 /100	8 /100
avg time for sol	9.69	5.03	3.15	**2.61**	33.61	29.02	13.59	5.65
avg bt for sol	**16628**	30696	30696	30232	413227	413510	30696	30689

Table 4.8: Randomly generated instances with 4 soft SEQUENCEs. Number of instances solved in 300 sec / average time to solve.

		$\Delta = 1$				$\Delta = 5$			
n	k	AD_S		FB_S		AD_S		FB_S	
50	7	6	/ 19.30	**7**	/ **27.91**	**20**	/ **0.01**	**20**	/ 2.17
	15	8	/ 36.07	**13**	/ **20.41**	**11**	/ **49.49**	10	/ 30.51
	25	6	/ 0.73	**10**	/ **23.27**	**10**	/ **6.40**	**10**	/ 7.41
100	7	1	/0	**3**	/ **7.56**	**19**	/ **10.50**	18	/ 16.51
	15	0	/0	**5**	/ **6.90**	**3**	/ **0.01**	**3**	/ 7.20
	25	0	/0	**5**	/ **4.96**	**5**	/ **19.07**	**5**	/ 23.99
	TOTALS								
	solved/total	21	/120	**43**	/120	**68**	/120	66	/120
	avg time for solved	19.463		**18.034**		**13.286**		13.051	
	avg bt for solved	245245		**343**		**147434**		128	

4.7 Conclusions

The SEQUENCE constraint is useful in modelling a range of rostering, scheduling and car sequencing problems. We proved that down the whole branch of a search tree domain consistency can be enforced on the SEQUENCE constraint in just $O(n^2)$ time using a reformulation into a network flow problem. This improves upon the previous bound of $O(n^3)$ for each call down the branch [HPRS]. We also introduced a soft version of the SEQUENCE constraint and proposed an $O(n^2 \log n \log \log u)$ time domain consistency algorithm based on minimum cost network flows. We proposed domain consistency algorithms for several generalisations of the SEQUENCE constraint, including GEN-SEQUENCE, SLIDINGSUM and soft versions of these constraints. To propagate the SEQUENCE constraint, we also introduced six new encodings, some of which do not hinder propagation. Our experiments on random and nurse scheduling benchmarks suggest that, on very large and tight problems, the domain consistency algorithms and decompositions mostly outperform incomplete decompositions. However, on smaller or looser problems much simpler encodings are better, even though these encodings hinder propagation. Experimental results also demonstrate that the FB filtering algorithm is more robust than existing propagators. This also suggests that we can construct a portfolio-based algorithm for the SEQUENCE constraint that, depending on the looseness of the constraint, can invoke the best encoding. When there are multiple SEQUENCE constraints, especially when we are forcing values to occur, a more expensive propagator shows promise. We also showed that our flow-based domain consistency propagator for the soft SEQUENCE constraint outperforms a decomposition into AMONG constraints.

Chapter 5

The GRAMMAR constraint

5.1 Introduction

An attractive mechanism to specify global constraints in rostering and other domains is using formal languages. For instance, we might need to produce an optimum schedule for a company that is subject to various regulation rules, e.g. an employee can work at most two consecutive night shifts or must have two days-off per week. One way to encode rules of this type is to represent them with a finite automaton. Pesant [Pes04] introduced the REGULAR constraint that specifies that an assignment of a sequence of variables forms a string from a regular language .

Definition 5.1 *The constraint* REGULAR$(\mathcal{A}, [X_1, \ldots, X_n])$ *is satisfied if and only if* X_1 *to* X_n *is a string accepted by the finite automaton* $\mathcal{A}$*.*

The REGULAR constraint is widely used and is implemented in many modern constraint solvers, as a lot of constraints can be encoded as the REGULAR constraint. However, in modelling shift scheduling problems, a regular language might not be expressive enough to succinctly encode regulation rules. A naive REGULAR encoding of regulation rules can be very expensive, as the resulting automaton is very large in some cases, which leads to large memory consumption and slows down search [QW07]. Alternatively, regulation rules can be represented in a succinct way using the GRAMMAR constraint [Sel06, QW06], an approach that was shown to be effective by Quimper and Walsh [QW07] and Kadioglu and Sellmann [KS08] .

Definition 5.2 *The constraint* GRAMMAR$([X_1, \ldots, X_n], G)$ *is satisfied if and only if* X_1 *to* X_n *is a string accepted by the context free grammar* G *[Sel06, QW06].*

A GRAMMAR constraint can be exponentially more succinct than the corresponding REGULAR constraint [KNW09], but reasoning with it is significantly more expensive. Therefore, an interesting problem is whether we can propagate the GRAMMAR constraint more efficiently than the worst-case complexity which is $O(n^3|G|)$. In this work we investigate two different approaches to this problem.

The first approach is to reformulate a GRAMMAR constraint as a REGULAR constraint and use existing automata minimisation algorithms to reduce the size of REGULAR. An advantage of this approach is that if the reformulated REGULAR constraint has a reasonable size and the minimisation algorithm is effective we obtain a fast filtering algorithm for the GRAMMAR constraint. In particular, this approach should work well for grammars that are close to a regular grammar but this is not obvious from the representation of the grammar in Chomsky normal form, and grammars that have external constraints, like a restriction on the length of a derivation from some non-terminals. A disadvantage of this approach is that the reformulation might increase exponentially the size of problem specification so that we will not be able to complete our reformulation procedure in the worst case. Therefore, another challenge here is to predict the size of the resulting REGULAR constraint to avoid running into the worst case scenario. We investigate this direction in Section 5.3.

The second approach is to impose some restrictions on the context-free grammar G in the specification of the GRAMMAR constraint so that G is strictly between regular and context-free grammars in the Chomsky hierarchy. A large body of work in formal language theory considers these subclasses of grammars. Several restricted forms of context-free grammars have been proposed that permit linear parsing algorithms whilst being more expressive than regular grammars. Examples of such grammars are LL(k), LR(1), and LALR. Such grammars play an important role in compiler theory and pattern recognition. For instance, yacc generates parsers that accept LALR languages. Therefore, we might expect that using these restricted classes leads to a more efficient filtering algorithm. This idea is investigated in Section 5.4.

Another direction that we investigate in the chapter is whether we can extend the GRAMMAR constraint to model over-constrained problems and problems with preferences. For instance, in some problems we might need to express a relaxed version of regulation rules, like an employee works without a break during a day or works extra hours. In this case an employee is paid at a higher rate. The goal in this type of problems is to find a schedule that minimises the overall additional expenses. To express these restrictions we introduce a generalisation of the GRAMMAR constraint — the weighted GRAMMAR constraint.

Definition 5.3 *The* WEIGHTEDGRAMMAR$(G, W, z, [X_1, \ldots, X_n])$ *constraint, where* G *is a context-free grammar,* W *is a function that maps productions of* G *to weights,* z *is a cost variable and* $X_1, \ldots, X_n$ *are decision variables, holds if and only if an assignment* X *forms a string belonging to the grammar* G *and the minimal weight of a derivation of* X *is less than or equal to* z*. The weight of a string derivation is the sum of the production weights used in the derivation.*

We propose a *DC* filtering algorithm and a decomposition of the weighted GRAMMAR constraint that does not hinder propagation in Section 5.5. We also show that the weighted GRAMMAR constraint can be used to model the soft GRAMMAR constraint with Hamming and edit distances violation measures (Section 2.3.1)

Definition 5.4 *The soft* GRAMMAR$(G, z, [X_1, \ldots, X_n])$ *constraint holds if and only if the string* $[X_1, \ldots, X_n]$ *is at most Hamming (edit) distance* z *from a string in the context free grammar* G*.*

Finally, we show that the weighted GRAMMAR constraint can be used to encode the EDITDISTANCE constraint.

Definition 5.5 EDITDISTANCE$([X_1, \ldots, X_n], [Y_1, \ldots, Y_m], N)$ *holds if and only if the edit distance between assignments of two sequences of variables* **X** *and* **Y** *is less than or equal to* N*.*

In this chapter we make the following contributions:

- investigate the relationship between REGULAR and GRAMMAR and propose an algorithm to convert the GRAMMAR constraint into the REGULAR constraint (Sections 5.3.1–5.3.3).
- propose an efficient algorithm to compute the number of states of a non-deterministic finite automaton that is equivalent to the original GRAMMAR constraint (Section 5.3.4).
- propose minimisation techniques for the REGULAR constraint (Sections 5.3.5–5.3.6).
- experimentally evaluate the proposed reformulations on some shift scheduling problems and show that it is often beneficial to reformulate a GRAMMAR constraint as a REGULAR constraint on these problems (Section 5.3.7).

- explore the gap in the Chomsky hierarchy between regular and context-free grammars to identify grammar classes which can or cannot be propagated more efficiently than context-free grammars (Section 5.4).
- introduce weighted and soft GRAMMAR constraints and propose domain consistency propagators for these constraints (Section 5.5.1 and Section 5.5.3).
- propose a decomposition of the weighted GRAMMAR constraints into set of bounded arity constraints (Section 5.5.2).
- propose an encoding of the EDITDISTANCE constraint into the linear weighted GRAMMAR constraints (Section 5.5.4).
- experimentally evaluate the proposed propagators and decompositions on some shift scheduling problems (Section 5.5.6).

5.2 Dynamic programming based algorithms

In this section we present an overview of existing algorithms for the GRAMMAR constraint. We start by reviewing a domain consistency algorithm for the GRAMMAR constraint [Sel06, QW06]. We use the following example to illustrate the algorithm.

Example 5.1 (Running example (GRAMMAR)) *As the running example we use the* GRAMMAR$([X_1, X_2, X_3], G)$ *constraint with domains* $D(X_1) = \{a\}$, $D(X_2) = \{a, b\}$, $D(X_3) = \{b\}$ *and the grammar* G *in Chomsky normal form* $\{S \rightarrow AB, A \rightarrow AA \mid a, B \rightarrow BB \mid b\}$ *[QW06]. The language generated by* G, $L(G)$, *is all strings of the form* a^+b^+. $\diamond$

A domain consistency propagator for the GRAMMAR constraint that we consider is based on the *CYK* parser for context-free grammars (Section 2.3.4). We recall that the idea of the *CYK* parser is to construct, in a dynamic programming manner, a table that represents all possible derivations of a string. The $[i, j]$th cell of the table contains a set of non-terminals. From each of these non-terminals we can derive a substring of length j that starts at the ith position. If the $[1, n]$th cell of the table contains the starting non-terminal then the string belongs to the context-free language. The *CYK* algorithm requires the input grammar to be in Chomsky normal form.

Similarly to the *CYK* parser, the GRAMMAR propagator constructs all possible derivations of all possible strings that can be formed from variable domain values. The algorithm has two stages. The pseudo-code is presented in Algorithm 5.1. In the first stage,

we construct a dynamic programing table $V[i,j]$, $i,j = 1,\ldots,O(n)$, where an element A of $V[i,j]$ is a non-terminal that generates a substring over a sub-sequence of variables $[X_i,\ldots,X_{i+j-1}]$ (lines 2–8) using values from domains of these variables. For instance, a set of strings over the sequence of variables $[X_1,X_2]$ is $\{aa,ab\}$. Hence, $V[1,2]$ contains all non-terminals that can generate one of these strings. The non-terminal A generates a string aa and the non-terminal S generates a string ab.

In the second stage, we move from $V[1,n]$ to the bottom of table V. For an element A of $V[i,j]$, we determine whether a derivation from A of a substring $[X_i,\ldots,X_{i+j-1}]$ can participate in a derivation of a solution $[X_1,\ldots,X_n]$ from S. If so, we mark A (lines 12–17). All unmarked terminals and non-terminals can be removed from the table. Note that if a terminal a is removed from position i then the value a can be removed from the domain of the variable X_i (lines 18–19). The time complexity of the algorithm is dominated by lines 12 – 17. We consider each non-terminal in the table and try all possible 1-step derivations from this non-terminal in a loop. There are $|N|$ non-terminals in the grammar G and $O(n^2)$ cells in the table. Let $F(A)$ be the number of productions in G with non-terminal A on the left-hand side. Then the number of 1-step derivations from each non-terminal is $O(F(A)n)$. Therefore, the total time complexity on the propagator is $O(n^2\sum_{A\in N} nF(A)) = O(n^3|G|)$. Note that this is the same complexity as the complexity of the *CYK* parsing algorithm (Section 2.3.4).

Example 5.2 *The dynamic programming table V produced by the propagator of the* GRAMMAR *constraint for Example 5.1 is given in Figure 5.1. Figure 5.1 (a) shows the result of the first stage. Figure 5.1 (b) shows the result of the second stage after we removed all unmarked entries from table.*$\diamond$

We call a *trace* in the dynamic programming table V a parsing tree of a solution of the GRAMMAR constraint. Figure 5.1 (b) shows all possible traces using arrows. For example, two possible productions from the starting non-terminal S are shown as two pairs of pointers: $S \rightarrow A_{11}B_{21}$ and $S \rightarrow A_{12}B_{31}$.

An alternative view of the dynamic programming table produced by this propagator is as an AND/OR graph [QW07]. This is a layered directed acyclic graph, with layers alternating between AND-NODES or OR-NODES. Each OR-NODE in the AND/OR graph corresponds to an entry $A \in V[i,j]$. An OR-NODE has a child AND-NODE for each production $A \rightarrow BC$ so that $A \in V[i,j]$, $B \in V[i,k]$ and $C \in V[i+k,j-k]$. The children of this AND-NODE are the OR-NODES that correspond to the entries $B \in V[i,k]$ and $C \in V[i+k,j-k]$. Note that the AND/OR graph constructed in this manner is

Algorithm 5.1 The *CYK*-based propagator for the GRAMMAR constraint

1: **procedure** *CYK*-ALG($G, z, [X_1, \ldots, X_n]$)
2: **for** $i = 1$ **to** n **do**
3: $V[i,1] = \{A | A \to a \in G, a \in D(X_i)\}$
4: **for** $j = 2$ **to** n **do**
5: **for** $i = 1$ **to** $n - j + 1$ **do**
6: $V[i,j] = \emptyset$;
7: **for** $k = 1$ **to** $j - 1$ **do**
8: $V[i,j] = V[i,j] \cup \{A | A \to BC \in G, B \in V[i,k], C \in V[i+k, j-k]\}$
9: **if** $S \notin V[1,n]$ **then**
10: return FALSE;
11: mark $(1, n, S)$;
12: **for** $j = n$ **downto** 2 **do**
13: **for** $i = 1$ **to** $n - j + 1$ **do**
14: **for** A such that (i, j, A) is marked **do**
15: **for** $k = 1$ **to** $j - 1$ **do**
16: **for** each $A \to BC \in G$ s.t. $B \in V[i,k], C \in V[i+k, j-k]$ **do**
17: mark (i, k, B); mark $(i + k, j - k, C)$;
18: **for** $i = 1$ **to** n **do**
19: $D(X_i) = \{a \in D(X_i) | A \to a \in G, (i, 1, A)\ is\ marked\}$;
20: return TRUE;

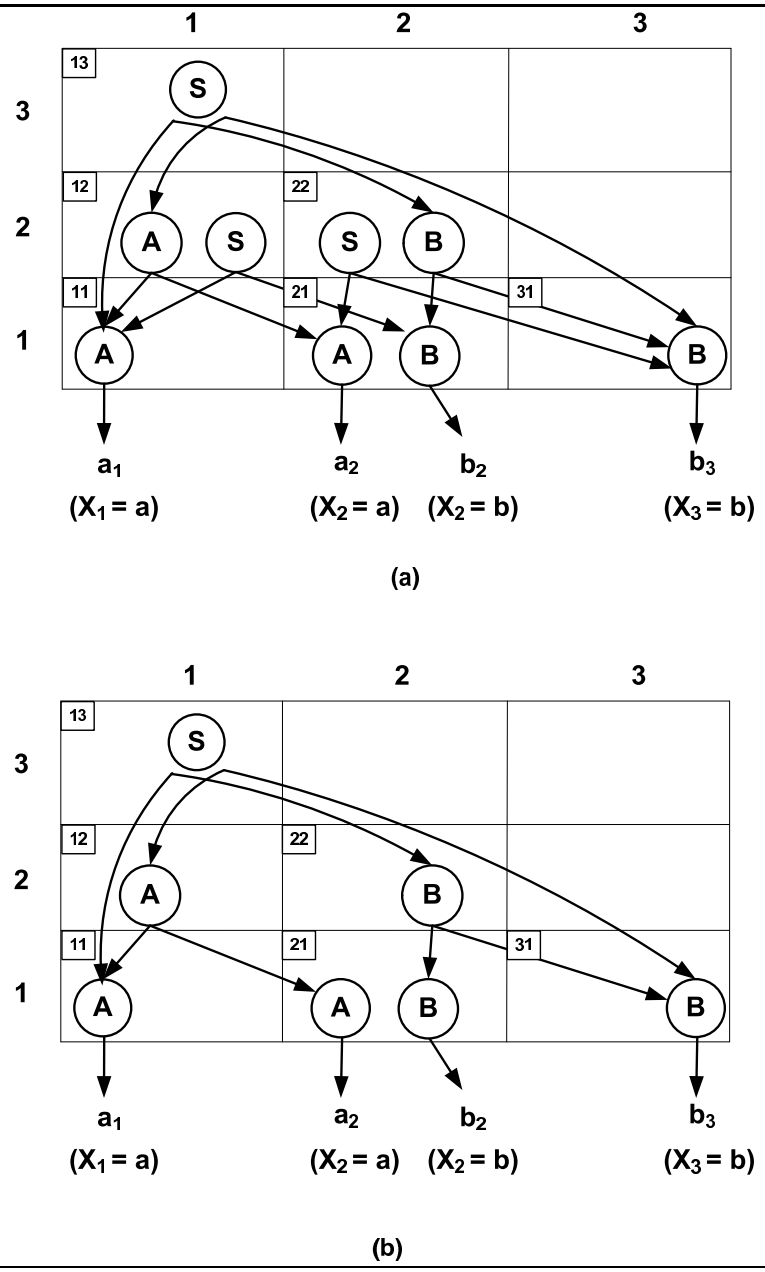
equivalent to the table V, so we use them interchangeably. Figure 5.2 shows the AND/OR graph constructed for the table V in the running example.

Every derivation of a string $s \in \mathcal{L}(G)$ can be represented as a tree that is a subgraph of the AND/OR graph and therefore can be represented as a trace in V. Since every possible derivation can be represented this way, both the table V and the corresponding AND/OR graph are a compilation of all solutions of the GRAMMAR constraint.

5.3 Reformulation of the GRAMMAR constraint

In this section, we investigate whether we can obtain a faster propagation algorithm for the GRAMMAR constraint by reformulation into the REGULAR constraint. We argue that if the reformulation does not produce a large transition relation then we can significantly improve

Figure 5.1 Dynamic programming table produced by the propagator of the GRAMMAR constraint. Pointers correspond to possible derivations.

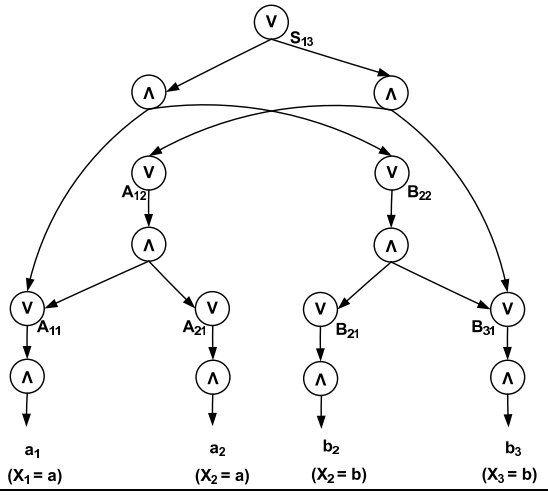


propagation speed of the GRAMMAR constraint. In addition, we can perform optimisations such as minimising the automaton to further compress the constraint representation and, as a result, achieve more performance gain.

In the worst case the reformulation does not give any benefits because the resulting *NFA* is exponentially larger then the original GRAMMAR constraint as the following example shows.

Example 5.3 (Theorem 4 [Sel06]) *Consider* GRAMMAR$([X_1, \ldots, X_n], G)$ *where* G *generates* $L = \{ww^R \# vv^R | w, v \in \{0,1\}^{n/2}\}$*, where* w^R *denote the reverse of* w*. Solutions*

Figure 5.2 AND/OR graph.



of GRAMMAR *can be compiled into the dynamic programming table of size $O(n^3)$, while the smallest equivalent NFA that accepts the same language has exponential size.*

Note that the separation between regular and context free languages concerns expressibility only and does not immediately imply an exponential separation for finite languages.

Therefore, performing the transformation itself is not a suitable test of the feasibility of the approach. In Section 5.3.4 we investigate whether we can compute the size of the resulting automaton without performing the actual transformation. We propose two polynomial algorithms to determine the size of the resulting automaton. The first algorithm finds an upper bound of the size of automaton in linear time, while the second one computes the exact size of *NFA* in $O(n^2)$.

In the rest of this section we describe the reformulation in three steps. First, we convert the GRAMMAR constraint into an acyclic grammar (Section 5.3.1). Then we convert this acyclic grammar into a pushdown automaton (Section 5.3.2), and finally we encode the pushdown automaton as a *NFA* (Section 5.3.3). The first two steps are well known in formal language theory but we describe them for clarity.

5.3.1 Transformation of GRAMMAR into an acyclic grammar

We first construct an acyclic grammar, $G_a(\Sigma, H, P, S)$ such that the language $\mathcal{L}(G_a)$ coincides with solutions of the GRAMMAR constraint. We recall that Σ is the set of terminals, H is the set of non-terminals, P is the set of productions and S is the starting non-terminal (Section 2.3.3). Our algorithm is based on the propagator for the GRAMMAR constraint (Section 5.2). Given the table V produced by the GRAMMAR constraint propagator (Section 5.2), we construct an acyclic grammar G_a in the following way. For each possible

derivation from a non-terminal A, $A \rightarrow BC$, such that $A \in V[i,j]$, $B \in V[i,k]$ and $C \in V[i+k, j-k]$ we introduce a production $A_{i,j} \rightarrow B_{i,k}C_{i+k,j-k}$ in G_a (lines 11–17 of Algorithm 5.2). The start symbol of G_a is $S_{1,n}$. By construction, the obtained grammar G_a is acyclic (Section 2.3.2). Every production in G_a is of the form $A_{i,j} \rightarrow B_{i,k}C_{i+k,j-k}$ and non-terminals $B_{i,k}$, $C_{i+k,j-k}$ occur in rows below jth row in V. Example 5.4 shows the grammar G_a obtained by Algorithm 5.2 on our running example.

Example 5.4 *The acyclic grammar G_a that corresponds to the* GRAMMAR *constraint in constructed from our running example.*

$$S_{1,3} \rightarrow A_{1,2}B_{3,1} \mid A_{1,1}B_{2,2} \qquad A_{1,2} \rightarrow A_{1,1}A_{2,1} \qquad B_{2,2} \rightarrow B_{2,1}B_{3,1}$$
$$A_{1,1} \rightarrow a_1 \;\; A_{2,1} \rightarrow a_2 \qquad B_{2,1} \rightarrow b_2 \qquad B_{3,1} \rightarrow b_3$$

◇

To prove equivalence, we recall that traces of the table V represent all possible derivations of GRAMMAR solutions. Therefore, every derivation of a solution can be simulated by productions from G_A. For instance, consider the solution (a, a, b) of GRAMMAR from Example 5.1. A possible derivation of this string is

$$\frac{S}{S \in V[1,3]} \rightarrow \frac{AB}{A \in V[1,2], B \in V[3,1]} \rightarrow$$
$$\frac{AAB}{A \in V[1,1], A \in V[2,1], B \in V[3,1]} \rightarrow \frac{a_1AB}{A \in V[2,1], B \in V[3,1]} \rightarrow$$
$$\frac{a_1a_2B}{B \in V[3,1]} \rightarrow a_1a_2b_3$$

We can simulate this derivation using productions in G_a: $S_{1,3} \rightarrow A_{1,2}B_{3,1} \rightarrow A_{1,1}A_{2,1}B_{3,1} \rightarrow a_1A_{2,1}B_{3,1} \rightarrow a_1a_2B_{3,1} \rightarrow a_1a_2b_3$.

Observe that the acyclic grammar G_a is essentially a labelling of the AND/OR graph, with non-terminals corresponding to OR-NODES and productions corresponding to AND-NODES. Thus, we use the notation G_a to refer to both the AND/OR graph and the corresponding acyclic grammar.

5.3.2 Transformation into a pushdown automaton

Given an acyclic grammar $G_a = (\Sigma, H, P, S_{1,n})$ from the previous section, we now construct a pushdown automaton $P_a(\langle S_{1,n} \rangle, Q, \Sigma, \Sigma \cup H, \delta, Q_P, F_P)$, where $\langle S_{1,n} \rangle$ is the initial

Algorithm 5.2 Transformation to an Acyclic Grammar

1: **procedure** CONSTRUCTACYCLICGRAMMAR($in : X, G, V; out : G_a$)
2: $\Sigma = \emptyset$ ▷ Σ is the set of terminals in G_a
3: $H = \emptyset$ ▷ H is the set of non-terminals in G_a
4: $P = \emptyset$ ▷ P is the set of productions in G_a
5: **for** $i = 1$ **to** n **do**
6: $V[i, 1] = \{A | A \rightarrow a \in G, a \in D(X_i)\}$
7: **for** $A \in V[i, 1]$ s.t $A \rightarrow a \in G, a \in D(X_i)$ **do**
8: $\Sigma = \Sigma \cup \{a_i\}$
9: $H = H \cup \{A_{i,1}\}$
10: $P = P \cup \{A_{i,1} \rightarrow a_i\}$
11: **for** $j = 2$ **to** n **do**
12: **for** $i = 1$ **to** $n - j + 1$ **do**
13: **for** each $A \in V[i, j]$ **do**
14: **for** $k = 1$ **to** $j - 1$ **do**
15: **for** each $A \rightarrow BC \in G$ s.t. $B \in V[i, k], C \in V[i + k, j - k]$ **do**
16: $H = H \cup \{A_{i,j}, B_{i,k}, C_{i+k,j-k}\}$
17: $P = P \cup \{A_{i,j} \rightarrow B_{i,k} C_{i+k,j-k}\}$

stack of P_a, Σ is the alphabet, $\Sigma \cup H$ is the set of stack symbols, δ is the transition function, $Q = Q_P = F_P = \{q_P\}$ is the single initial and accepting state (Section 2.3.3). We use an algorithm that encodes a context free grammar into a pushdown automaton (*PDA*) that computes the leftmost derivation of a string [HU90]. The stack maintains the sequence of symbols that are expanded in this derivation. At every step, the *PDA* non-deterministically uses a production of the grammar to expand the top symbol of the stack if it is a non-terminal, or consumes a symbol of the input string if it matches the terminal at the top of the stack.

We now describe this reformulation in detail. There exists a single state q_P which is both the starting and an accepting state. For each non-terminal $A_{i,j}$ in G_a we introduce the set of transitions $\delta(q_P, \varepsilon, A_{i,j}) = \{(q_P, \beta) | A_{i,j} \rightarrow \beta \in G_a\}$. For each terminal $a_i \in G_a$, we introduce a transition $\delta(q_P, a_i, a_i) = \{(q_P, \varepsilon)\}$. The automaton P_a accepts on the empty stack. This constructs a pushdown automaton accepting $\mathcal{L}(G_a)$.

Example 5.5 *The pushdown automaton P_a constructed for the running example.*

$$\delta(q_P, \varepsilon, S_{1,3}) = (q_P, A_{1,2}B_{3,1}) \qquad \delta(q_P, \varepsilon, S_{1,3}) = (q_P, A_{1,1}B_{2,2})$$
$$\delta(q_P, \varepsilon, A_{1,2}) = (q_P, A_{1,1}A_{2,1}) \qquad \delta(q_P, \varepsilon, B_{2,2}) = (q_P, B_{2,1}B_{3,1})$$
$$\delta(q_P, \varepsilon, A_{1,1}) = (q_P, a_1) \qquad \delta(q_P, \varepsilon, A_{2,1}) = (q_P, a_2)$$
$$\delta(q_P, \varepsilon, B_{2,1}) = (q_P, b_2) \qquad \delta(q_P, \varepsilon, B_{3,1}) = (q_P, b_3)$$
$$\delta(q_P, a_i, a_i) = (q_P, \varepsilon) \forall i \in \{1, 2\} \qquad \delta(q_P, b_i, b_i) = (q_P, \varepsilon) \forall i \in \{2, 3\}$$

◇

5.3.3 Transformation into a *NFA*

Finally, we construct an $NFA(\Sigma, Q, Q_0, F_0, \sigma)$, denoted N_a, using the *PDA* from the last section. States of this *NFA* encode all possible configurations of the stack of the *PDA* that can appear in parsing a string from G_a. To reflect that a state of the *NFA* represents a stack, we write states as sequences of symbols $\langle\alpha\rangle$, where α is a possibly empty sequence of symbols and $\alpha[0]$ is the top of the stack. For example, the initial state is $\langle S_{1,n}\rangle$ corresponding to the initial stack $\langle S_{1,n}\rangle$ of P_a. Algorithm 5.3 unfolds the *PDA* in a similar way to unfolding the *DFA*. Note that the *NFA* accepts only strings of length n and has the initial state $Q_0 = \langle S_{1,n}\rangle$ and the single final state $F_0 = \langle\rangle$.

We start from the initial stack $\langle S_{1,n}\rangle$ and find all distinct stack configurations that are reachable from this stack using transitions from P_a. For each reachable stack configuration we create a state in the *NFA* and add the corresponding transitions. If the new stack configurations are the result of expansion of a production in the original grammar, these transitions are $\varepsilon-$transitions, otherwise they consume a symbol from the input string. Note that if a non-terminal appears on top of the stack and gets replaced, then it cannot appear in any future stack configuration due to the acyclicity of G_a. Therefore $|\alpha|$ is bounded by $O(\max(|G_a|, n))$ and Algorithm 5.3 terminates. The size of N_a is $O(|G_a|^n)$ in the worst case. The automaton N_a that we obtain before line 21 is an acyclic *NFA* with ε transitions. It accepts the same language as the *PDA* P_a since every path between the starting and the final state of N_A is a trace of the stack configurations of P_a. Figure 5.3(a) shows the automaton N_a with ε-transitions constructed from the running example. After applying the ε-closure operation, we obtain a layered *NFA* that does not have ε transitions (line 21) (Figure 5.3(b)).

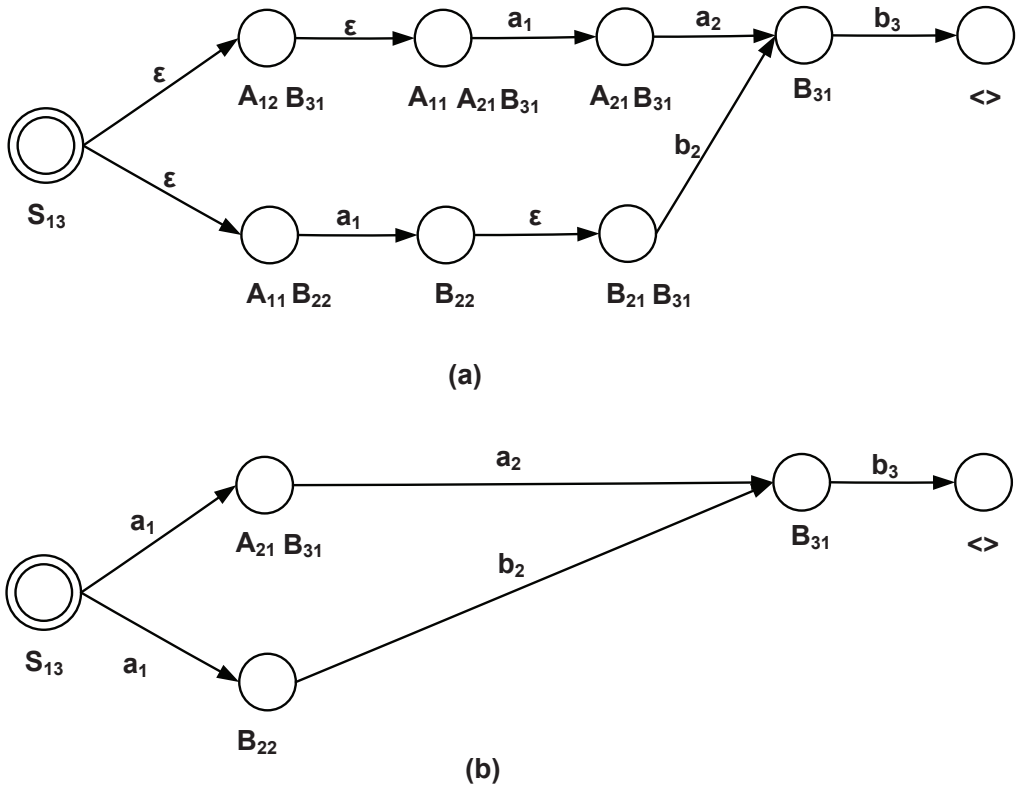
Algorithm 5.3 Transformation to *NFA*

1: **procedure** *PDA* TO *NFA*($in : P_a, out : N_a$)
2: $Q_u = \{\langle S_{1,n}\rangle\}$ ▷ Q_u is the set of unprocessed states
3: $Q = \emptyset$ ▷ Q is the set of states in N_a
4: $\sigma = \emptyset$ ▷ σ is the set of transitions in N_a
5: $Q_0 = \{\langle S_{1,n}\rangle\}$ ▷ Q_0 is the initial state in N_a
6: $F_0 = \{\langle\rangle\}$ ▷ F_0 is the set of final states in N_a
7: **while** $q \in Q_u$ is not empty **do**
8: **if** $q \equiv \langle A_{i,j}, \alpha\rangle$ **then**
9: **for** each transition $\delta(q_P, \varepsilon, A_{i,j}) = (q_P, \beta) \in \delta$ **do**
10: $\sigma = \sigma \cup \{\sigma(\langle A_{i,j}, \alpha\rangle, \varepsilon) = \langle\beta, \alpha\rangle\}$
11: **if** $\langle\beta, \alpha\rangle \notin Q$ **then**
12: $Q_u = Q_u \cup \{\langle\beta, \alpha\rangle\}$
13: $Q = Q \cup \{\langle A_{i,j}, \alpha\rangle\}$
14: **else if** $q \equiv \langle a_i, \alpha\rangle$ **then**
15: **for** each transition $\delta(q_P, a_i, a_i) = (q_P, \varepsilon) \in \delta$ **do**
16: $\sigma = \sigma \cup \{\sigma(\langle a_i, \alpha\rangle, a_i) = \langle\alpha\rangle\}$
17: **if** $\langle\alpha\rangle \notin Q$ **then**
18: $Q_u = Q_u \cup \{\langle\alpha\rangle\}$
19: $Q = Q \cup \{\langle a_i, \alpha\rangle\}$
20: $Q_u = Q_u \setminus \{q\}$
21: $N_a(\Sigma, Q, Q_0, F_0, \sigma) = \varepsilon - Closure(N_a(\Sigma, Q, Q_0, F_0\sigma))$.

5.3.4 Computing the size of the *NFA*

As the *NFA* may be exponential in size, we provide a polynomial method of computing its size in advance. We can use this to decide if it is practical to transform it in this way. Observe first that the transformation of a *PDA* to an *NFA* maintains a queue of states that correspond to stack configurations. Each state encodes a stack configurations and corresponds to several OR-NODES in the AND/OR graph because there are a number of derivation that lead to the same stack. Each state of an OR-NODE v is generated from the states of the parent OR-NODES of v. This suggests a relationship between paths in the AND/OR graph of the *CYK* algorithm and states in N_a. We use this relationship to compute a loose upper bound for the number of states in N_a in time linear in the size of the AND/OR graph by counting the number of paths in that graph. Alternatively, we compute the exact number of

Figure 5.3 N_a produced by Algorithm 5.3



states in N_a in time quadratic in the size of the AND/OR graph.

Theorem 5.1 *There exists a surjection from paths in G_a from the root to* OR-NODES *onto stack configurations in the PDA P_a.*

Proof: Consider a path p from the root of the AND/OR graph to an OR-NODE labelled with $A_{i,j}$. We construct a stack configuration $\Gamma(p)$ that corresponds to p. We start with the empty stack $\Gamma = \langle\rangle$. We traverse the path from the root to $A_{i,j}$. For every AND-NODE $v_1 \in p$, with left child v_l and right child v_r, if the successor of v_1 in p is v_l, then we push v_r on Γ, otherwise do nothing. When we reach $A_{i,j}$, we push it on Γ. The final configuration Γ is unique for p and corresponds to the stack of the *PDA* after having parsed the substring $1 \ldots i-1$ and having non-deterministically chosen to parse the substring $i \ldots i+j-1$ using a production with $A_{i,j}$ on the LHS.

We now show that all stack configurations can be generated by the procedure above. Every stack configuration corresponds to at least one partial leftmost derivation of a string. The leftmost derivation of a string is a derivation that always applies a production rule to the leftmost non-terminal. We say a stack configuration $\langle\alpha\rangle$ corresponds to a partial derivation $dv = \langle a_1, \ldots, a_{k-1}, A_{k,j}, \alpha\rangle$ where $a_1, \ldots, a_{k-1}$ is the parsed substring of length $k-1$ starting at position 1, $A_{k,j}$ is the leftmost non-terminal that is used to parse a substring of

length j starting at position k and the α is the context of the stack after parsing the whole prefix of the string of length $k + j$. Therefore, it is enough to show that all partial leftmost derivations (we omit the prefix of terminals) can be generated by the procedure above.

We prove this by induction. The base case is trivial. Suppose that $\langle a_1, \ldots, a_{i-1}, B_{i,j}, \beta\rangle$ is the partial leftmost derivation such that $\Gamma(p(root, B_{i,j})) \neq \beta$, where $p(root, B_{i,j})$ is a path from the root to the OR-NODE $B_{i,j}$ and for any partial derivation $\langle a_1, \ldots, a_{k-1}, A_{k,j}, \alpha\rangle$, such that $k < i$ $A_{k,j} \in G_a$ $\Gamma(p(root, A_{k,j})) = \alpha$. Consider the production rule that introduces the non-terminal $B_{i,j}$ to the partial derivation. There are two possible cases depending where $B_{i,j}$ is in this production:

1. the production rule is $D \rightarrow C, B_{i,j}$ or
2. the production rule is $D \rightarrow B_{i,j}, C$.

Case 1. Suppose the production rule is $D \rightarrow C, B_{i,j}$ used. Then the partial derivation is $\langle a_1, \ldots, a_f, D, \beta\rangle \Rightarrow|_{D\rightarrow C,B_{i,j}} \langle a_1, \ldots, a_f, C, B_{i,j}, \beta\rangle$. The path from the root to the node $B_{i,j}$ is a concatenation of some paths from D to $B_{i,j}$ and from the root to D. Therefore, $\Gamma(p(root, B_{i,j}))$ is constructed as a concatenation of $\Gamma(p(D, B_{i,j}))$ and $\Gamma(p(root, D))$. $\Gamma(p(D, B_{i,j}))$ is empty because the node $B_{i,j}$ is the right child of the AND-NODE that corresponds to the production $D \rightarrow C, B_{i,j}$ and $\Gamma(p(root, D)) = \beta$ because $f < i$. Therefore, $\Gamma(p(root, B_{i,j})) = \beta$.

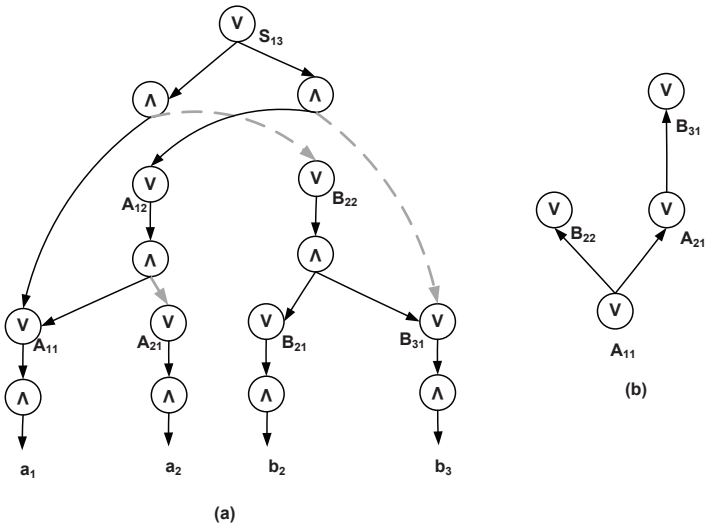
Case 2. Suppose the production rule is $D \rightarrow B_{i,j}, C$ used. Then the partial derivation is $\langle a_1, \ldots, a_{i-1}, D, \gamma\rangle \Rightarrow|_{D\rightarrow B_{i,j},C} \langle a_1, \ldots, a_{i-1}, B_{i,j}, C, \gamma\rangle = \langle a_1, \ldots, a_{i-1}, B_{i,j}, \beta\rangle$. Then, $\Gamma(p(root, D)) = \gamma$, because $i - 1 < i$ and $\Gamma(p(D, B_{i,j})) = \langle C\rangle$, because the node $B_{i,j}$ is the left child of AND-NODE that corresponds to the production $D \rightarrow B_{i,j}, C$. Therefore, $\Gamma(p(root, B_{i,j})) = \langle C, \gamma\rangle = \beta$. This leads to a contradiction.

$\diamond$

Example 5.6 *An example of the mapping described in the last proof is in Figure 5.4(a) for the grammar of our running example. Consider the* OR-NODE $A_{1,1}$. *There are 2 paths from* $S_{1,3}$ *to* $A_{1,1}$. *One is direct and uses only* OR-NODES $\langle S_{1,3}, A_{1,1}\rangle$ *and the other uses* OR-NODES $\langle S_{1,3}, A_{1,2}, A_{1,1}\rangle$. *The 2 paths are mapped to 2 different stack configurations* $\langle A_{1,1}, B_{2,2}\rangle$ *and* $\langle A_{1,1}, A_{2,1}, B_{3,1}\rangle$ *respectively. We highlight edges that are incident to* AND-NODES *on each path and lead to the right children of these* AND-NODES. *There is exactly one such edge for each element of a stack configuration.* $\diamond$

Note that theorem 5.1 only specifies a surjection from paths to stack configurations, not a bijection. Indeed, different paths may produce the same configuration Γ.

Figure 5.4 Computing the size of N_a. (a) AND/OR graph G_a. (b) Stack graph $G_{A_{1,1}}$



Example 5.7 *Consider the grammar* $G = \{S \to AA, A \to a|AA|BC, B \to b|BB, C \to c|CC\}$ *and the* AND/OR *graph of this grammar for a string of length 5. The path* $\langle S_{1,5}, A_{2,4}, B_{2,2}\rangle$ *uses the productions* $S_{1,5} \to A_{1,1}A_{2,4}$ *and* $A_{2,4} \to B_{2,2}C_{4,2}$*, while the path* $\langle S_{1,5}, A_{3,3}, B_{3,1}\rangle$ *uses the productions* $S_{1,5} \to A_{1,2}A_{3,3}$ *and* $A_{3,3} \to B_{3,1}C_{4,2}$*. Both paths map to the same stack configuration* $\langle C_{4,2}\rangle$ *after parsing the first three positions.* $\diamond$

By construction, the resulting *NFA* has one state for each stack configuration of the *PDA* in parsing a string. Since each path corresponds to a stack configuration, the number of states of the *NFA* before applying ε-closure is bounded by the number of paths from the root to any OR-NODE in the AND/OR graph. This is cheap to compute using the following recursive algorithm [Dar01]:

$$PD(v) = \begin{cases} 1 & \text{If } v \text{ has no incoming edges} \\ \sum_p PD(p) & \text{where } p \text{ is a parent of } v \end{cases} \tag{5.1}$$

Therefore, the number of states of the *NFA* N_a is at most $\sum_v PD(v)$, where v is an OR-NODE of G_a (Figure 5.4).

We can compute the exact number of paths in N_a before ε-closure without constructing the *NFA* by counting paths in the *stack graph* G_v for each OR-NODE v. The stack graph captures the observation that each element of a stack configuration generated from a path p is associated with exactly one edge e that is incident on p and leads to the right child of an AND-NODE. G_v contains one path for each sequence of such edges, so that if two paths p and p' in G_a are mapped to the same stack configuration, they are also mapped to the same path in G_v. The stack graph of an OR-NODE $v \in V(G_a)$ is a directed acyclic graph which

is constructed so that G_v, such that for every stack configuration Γ of P_a with k elements, there is exactly one path p in G_v of length k and v' is the i^{th} vertex of p if and only if v' is the i^{th} element from the top of Γ.

Example 5.8 *Consider the grammar of the running example and the* OR-NODE $A_{1,1}$ *in the* AND/OR *graph. The stack graph* $G_{A_{1,1}}$ *for this* OR-NODE *is shown in Figure 5.4(b). Along the path* $\langle S_{1,3}A_{1,1}\rangle$*, only the edge that leads to* $B_{2,2}$ *generates a stack element. This edge is mapped to the edge* $(A_{1,1}, B_{2,2})$ *in* $G_{A_{1,1}}$*. Similarly, the edges that lead to* $A_{2,1}$ *and* $B_{3,1}$ *are mapped to the edges* $(A_{1,1}, A_{2,1})$ *and* $(A_{2,1}, B_{3,1})$ *respectively.* $\diamond$

Since G_v is a DAG, we can efficiently count the number of paths in it. We construct G_v using Algorithm 5.4. The graph G_v computed in Algorithm 5.4 for an OR-NODE v has as many paths as there are unique stack configurations in P_a with v at the top.

Algorithm 5.4 Computing the stack DAG G_v of an OR-NODE v

1: **procedure** STACKGRAPH($(in : G_a, v, out : G_v)$)
2: $V(G_v) = \{v\}$
3: $label(v) = \{v\}$
4: $Q = \{(v, v_p) | v_p \in parents(v)\}$ ▷ queue of edges
5: **while** Q not empty **do**
6: $(v_c, v_p) = pop(Q)$
7: **if** v_p is an AND-NODE and v_c is left child of v_p **then**
8: $v_r = children_r(v_p)$
9: $V(G_v) = V(G_v) \cup \{v_r\}$
10: $E(G_v) = E(G_v) \cup \{(v_l, v_r) | v_l \in label(v_c)\}$
11: $label(v_p) = label(v_p) \cup \{v_r\}$
12: **else**
13: $label(v_p) = label(v_p) \cup label(v_c)$
14: $Q = Q \cup \{(v_p, v'_p) | v'_p \in parents(v_p)\}$

Theorem 5.2 *There exists a bijection between paths in* G_v *and states in the NFA* N_a *which correspond to stacks with* v *at the top.*

Proof: Let p be a path from the root to v in G_a. First, we show that every path p' in G_v corresponds to a stack configuration, by mapping p to p'. Therefore p' corresponds to $\Gamma(p)$. We then show that p' is unique for $\Gamma(p)$. This establishes a bijection between paths in G_v and stack configurations.

We traverse the inverse of p, denoted $inv(p)$ and construct p' incrementally. Note that every vertex in $inv(p)$ is examined by Algorithm 5.4 in the construction of G_v. If $inv(p)$ visits the left child of an AND-NODE, we append the right child of that AND-NODE to p'. This vertex is in G_v by line 7. By the construction of $\Gamma(p)$ in the proof of theorem 5.1, a symbol is placed on the stack if and only if it is the right child of an AND-NODE, hence if and only if it appears in p'. Moreover, if a vertex is the i^{th} vertex in a path, it corresponds to the i^{th} element from the top of $\Gamma(p)$. We now see that p' is unique for $\Gamma(p)$. Two distinct paths of length k cannot map to the same stack configuration, because they must differ in at least one position i, therefore they correspond to stacks with different symbols at position i. Therefore, there exists a bijection between paths in G_v and stack configurations with v at the top. $\diamond$

Hence $|Q(N_a)| = \sum_v \#paths(G_v)$, where v is an OR-NODE of G_a. Computing the stack graph G_v of every OR-NODE v takes $O(|G_a|)$ time, as does counting paths in G_v. Therefore, computing the number of states in N_a takes $O(|G_a|^2)$ time.

We can also compute the number of states in the ε-closure of $N_a = \langle \Sigma, Q, q_0, F, \delta \rangle$. The ε-closure of a state $q \in Q$ is a set of state that are reachable form q without consuming any input symbol: $\varepsilon(q) = \{q' | q \xrightarrow{\varepsilon} q', q' \in Q\}$. For any subset of states, $Q' \subseteq Q$, we define the ε-closure as $\varepsilon(Q') = \cup_{q \in Q'} \varepsilon(q)$. We can eliminate all ε transitions from N_a which is acyclic graph by construction, in the following way. We sort states in topological order. For each state q we find its ε-closure $\varepsilon(q)$. Consider a state q', $q' \in \varepsilon(q)$. For each outgoing transition that consumes a input symbol, $q'' = \delta(q', a)$, we replace this transition with the transition $q'' = \delta(q, a)$. The resulting *NFA* is a non-deterministic *NFA* without ε transitions.

We observe that if none of the OR-NODES that are reachable by paths of length 2 from an OR-NODE v correspond to terminals, then any state that corresponds to a stack configuration with v at the top will only have outgoing $\varepsilon-$transitions and will be removed by the $\varepsilon-$closure. Thus, to compute the number of states in N_a after $\varepsilon-$closure, we sum the number of paths in G_v for all OR-NODES v such that a terminal OR-NODE can be reached from v by a path of length 2.

5.3.5 Transformation into a *DFA*

Finally, we convert the *NFA* into a *DFA* using the standard subset construction [HU90]. Indeed, removing non-determinism may increase the size of the automaton and slow down propagation. However, converting into a *DFA* opens up the possibility of further optimisations. In particular, as we describe in the next section, there are efficient methods to

minimise the size of a *DFA*. By comparison, minimisation of a *NFA* is PSPACE-hard in general [MS72]. Even when we consider just the acyclic *NFA* constructed by unfolding a *NFA*, minimisation remains NP-hard [AJV01].

5.3.6 Automaton minimisation

The *DFA* constructed by this or other methods may contain redundant states and transitions. We can speed up propagation of the REGULAR constraint by minimising the size of this automaton. First, we outline the idea of the REGULAR constraint propagator that was proposed by Pesant [Pes04]. To propagate the REGULAR$([X_1, \ldots, X_n], \mathcal{A})$ constraint, we need to construct an unfolded layered automaton, $unfold_n(\mathcal{A})$, that only accepts words of length n which are accepted by $\mathcal{A}$ (Section 2.3.3 introduces unfolded automata). The first layer contains only the starting state and the last layer contains only the final states. There exists a bijection between solutions of the REGULAR constraint and paths in the unfolded layered automaton [Pes04]. Pesant's propagator was introduced for the REGULAR$([X_1, \ldots, X_n], \mathcal{A})$ constraint where $\mathcal{A}$ is *DFA*. However, it works with *NFAs* without any modification of the algorithm. Hence, we can minimise the original automaton $\mathcal{A}$ or its unfolded version $\mathcal{A}$.

Minimisation can be performed either offline (i.e. before we have the problem data and have unfolded the automaton) or on-line (i.e. once we have the problem data and have unfolded the automaton). There are several reasons why we might prefer an on-line approach where we unfold before minimising. First, although minimising after unfolding may be more expensive than minimising before unfolding, both are cheap to perform. Minimising a *DFA* takes $O(Q \log Q)$ time using Hopcroft's algorithm and $O(nQ)$ time for the unfolded *DFA* where Q is the number of states [Rev92]. Second, thanks to Myhill-Nerode's theorem, minimisation does not change the layered nature of the unfolded *DFA*. Third, and perhaps most importantly, minimising a *DFA* after unfolding can give an exponentially smaller automaton than minimising the *DFA* and then unfolding. To put it another way, unfolding may destroy the minimality of the *DFA*. We recall that we write $f_n(\mathcal{A}) \ll g_n(\mathcal{A})$ if and only if $f_n(\mathcal{A}) \leq g_n(\mathcal{A})$ for all n, and there exists $\mathcal{A}$ such that $\log \frac{g_n(\mathcal{A})}{f_n(\mathcal{A})} = \Omega(n)$.(Section 2.3.3).

Theorem 5.3 *Given any DFA* $\mathcal{A}$, $|\min(unfold_n(\mathcal{A}))| \ll |unfold_n(\min(\mathcal{A}))|$.

Proof: To show $|\min(unfold_n(\mathcal{A}))| \leq |unfold_n(\min(\mathcal{A}))|$, we observe that both $\min(unfold_n(\mathcal{A}))$ and $unfold_n(\min(\mathcal{A}))$ are automata that recognise the same language. By definition, minimisation returns the smallest *DFA* accepting this language. Hence $\min(unfold_n(\mathcal{A}))$ cannot be larger than $unfold_n(\min(\mathcal{A}))$.

To show unfolding then minimising can give an exponentially smaller sized *DFA*, consider the following language L_n. A string of length k belongs to L_n if and only if it contains the symbol j, $j = k \mod n$, where n is a given constant. The alphabet of the language L_n is $\{0, \ldots, n-1\}$. The minimal *DFA* for this language has $\Omega(n2^n)$ states as each state needs to record which symbols from 0 to $n-1$ have been seen so far, as well as the current length of the string mod n. Unfolding this minimal *DFA* and restricting it to strings of length n gives an acyclic *DFA* with $\Omega(n2^n)$ states. Note that all strings are of length n and the equation $j = n \mod n$ has the single solution $j = 0$. Therefore, the language L_n consists of the strings of length n that contain the symbol 0. On the other hand, if we unfold and then minimise, we get an acyclic *DFA* with just $2n$ states. Each layer of the *DFA* has two states which record whether 0 has been seen. $\diamond$

Further, if we make our initial problem domain consistent, domains might be pruned which give rise to possible simplifications of the *DFA*. We use $simplify(\mathcal{A})$ for the simplified form of $\mathcal{A}$ constructed by deleting transitions and states that are no longer reachable after domains have been reduced. We show here that we should also perform such simplification before minimising.

Theorem 5.4 *Given any DFA* $\mathcal{A}$, $|\min(simplify(unfold_n(\mathcal{A})))| \ll |simplify(\min(unfold_n(\mathcal{A})))|$.

Proof: Both $\min(simplify(unfold_n(\mathcal{A})))$ and $simplify(\min(unfold_n(\mathcal{A})))$ are *DFA*s that recognise the same language of strings of length n. By definition, minimisation must return the smallest *DFA* accepting this language. Hence $\min(simplify(unfold_n(\mathcal{A})))$ is no larger than $simplify(\min(unfold_n(\mathcal{A})))$.

To show that minimisation after simplification may give an exponentially smaller sized automaton, consider the language which contains sequences of integers from 1 to n in which at least one integer is repeated and in which the last two integers are different. The alphabet of the language L_n is $\{1, \ldots, n\}$. The minimal unfolded *DFA* for strings of length n from this language has $\Omega(2^n)$ states as each state needs to record which integers have been seen. Suppose the integer n is removed from the domain of each variable. The simplified *DFA* still has $\Omega(2^n)$ states to record which integers 1 to $n-1$ have been seen. On the other hand, suppose we simplify before we minimise. By a pigeonhole argument, we can ignore the constraint that an integer is repeated. Hence we just need to ensure that the string is of length n and that the last two integers are different. The minimal *DFA* accepting this language requires just $O(n)$ states. $\diamond$

5.3.7 Experimental results

We empirically evaluated the results of our method on a set of shift-scheduling benchmarks [DPR05, CBCGLM07] [1]. These shift-scheduling benchmarks were used in a series of papers on GRAMMAR constraints, including [KS08], [QW07] and [QR10]. Experiments were run with the MiniSat+ solver for pseudo-Boolean instances and GeCode 2.2.0 for constraint problems, on an Intel Xeon 4 CPU, 2.0 Ghz, 4G RAM. We use a timeout of 3600 sec in all experiments. The problem is to schedule employees to activities subject to various rules, e.g. a full-time employee has one hour for lunch. These rules are specified by a context-free grammar augmented with the following restrictions on productions [QW07]:

1. a full-time employee has to work between 7.5 and 9.5 hours
2. a part-time employee has to work between 3.25 and 6 hours
3. a full-time employee has to have one hour lunch break
4. a full-time employee has to have two 15 minute short breaks one before and one after lunch
5. a part-time employee has to have one 15 minute short break
6. an employee can change activities after a break or a lunch
7. an employee has to work consecutively on one activity for at least one hour.

A schedule for an employee has $n = 96$ slots of 15 minutes represented by n variables. In each slot, an employee can work on an activity (a_j), take a break (b), lunch (l) or rest (r). These rules are specified by the following grammar:

$$\begin{aligned} S \to RPR, \quad & P \to WbW, \quad L \to lL|l, \\ S \to RFR, \quad & R \to rR|r, \quad W \to A_i, \\ A_i \to a_i A_i|a_i, \quad & F \to PLP \end{aligned}$$

To take into account restrictions on productions we introduce functions $f(i,j)$ are predicates that restrict the start i and length j of any string matched by a specific production and $open(i)$ is a function that returns 1 if the business is open at i^{th} slot and 0 otherwise.

$$\begin{aligned} P : f_P(i,j) \equiv 13 \leq j \leq 24, \quad & L : f_L(i,j) \equiv j = 4 \\ F : f_F(i,j) \equiv 30 \leq j \leq 38, \quad & W : f_W(i,j) \equiv j \geq 4 \\ A_j : f_{A_j}(i,j) \equiv & \; open(i) \end{aligned}$$

[1] I would like to thank Louis-Martin Rousseau and Claude-Guy Quimper for providing me with the benchmark data

In addition, the business requires a certain number of employees working in each activity at given times during the day. We minimise the number of slots in which employees work such that the demand is satisfied.

As shown in [QW07], this problem can be converted into a pseudo-Boolean (PB) model. The GRAMMAR constraint is converted into a *SAT* formula in conjunctive normal form using the AND/OR graph. To model labour demand for a slot we introduce Boolean variables $b(i, j, a_k)$, equal to 1 if j^{th} employee performs activity a_k at i^{th} time slot. For each time slot i and activity a_k we post a pseudo-Boolean constraint $\sum_{j=1}^{m} b(i, j, a_k) > d(i, a_k)$, where m is the number of employees and $d(i, a_k)$ is the labour demand at the ith time point for the activity a_k. The objective is modelled using the function $\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{a} b(i, j, a_k)$. Additionally, the problem can be formulated as an optimisations problem in a constraint solver, using a matrix model with one row for each employee. We post a GRAMMAR constraint on each row, AMONG constraints (see Definition 4.1) on each column for labour demand and LEX constraints between adjacent rows to break symmetry. We use the static variable and value ordering used in [QW07]. We assign variables top down and from left to right. The value ordering is $r, b, l, a1, a2$. The goal is to minimise the number of slots in which employees work.

We compare the PB decomposition and the CP model described above with the reformulation of the GRAMMAR constraint as a REGULAR constraint. Using Algorithm 5.4, we computed the size of an equivalent *NFA*. Surprisingly, this has a reasonable size, so we converted the GRAMMAR constraint to a *DFA* then minimised. In order to reduce the blow-up that may occur during the conversion of the *NFA* to the *DFA*, we heuristically minimised the *NFA* using the following simple observation: two states are equivalent if they have identical outgoing transitions. We traverse the *NFA* from the last to the first layer and merge equivalent states and then apply the same procedure to the reversed *NFA*. We repeat until we cannot find a pair of equivalent states. We also simplified the original *CYK* table, taking into account whether the business is open or closed at each slot (this information is based on the value of $open(i)$ function). Theorem 5.4 suggests such simplification can significantly reduce the size both of the *CYK* table and of the resulting automata. In practice we also observe a significant reduction in size. The resulting minimised automaton obtained before simplification is about ten times larger compared to the minimised *DFA* obtained after simplification. Table 5.1 gives the sizes of representations at each step. We see from this that the minimised *DFA* is in every case smaller than the original *CYK* table. Interestingly, the subset construction generates the minimum *DFA* from the *NFA*, even in the case of two

activities, and heuristic minimisation of the *NFA* achieves a notable reduction.

For each instance, we used the resulting *DFA* in place of the GRAMMAR constraint in both the CP model and the *PB* model using the encoding of the REGULAR constraint (*DFAs* or *NFAs*) into CNF [Bac07]. We compare the model that uses the *PB* encoding of the GRAMMAR constraint (GR_1) with two models that use the *PB* encoding of the REGULAR constraint (REGULAR_1, REGULAR_2), a CP model that uses the GRAMMAR constraint (GR_1^{CP}) and a CP model that uses a REGULAR constraint (REGULAR_1^{CP}). REGULAR_1 and REGULAR_1^{CP} use the DFA, whilst REGULAR_2 uses the *NFA* constructed after simplification by when the business is closed.

The performance of a *SAT* solver can be sensitive to the ordering of the clauses in the formula. To test robustness of the models, we randomly shuffled each of *PB* instances to generate 10 equivalent problems and averaged the results over 11 instances. Also, the GRAMMAR and REGULAR constraints were encoded into a *PB* formula in two different ways. The first encoding ensures that unit propagation enforces domain consistency on the constraint. The second encoding ensures that unit propagation detects disentailment of the constraint, but does not always enforce domain consistency. For the GRAMMAR constraint we omit the same set of clauses as in [QW07] to obtain the weaker *PB* encoding. For the REGULAR constraint we omit the set of clauses that performs backward propagation of the REGULAR constraint. Note that Table 5.2 shows the median time and the number of backtracks to prove optimality over 11 instances. For each model we show the median time and the corresponding number of backtracks for the best *PB* encoding between the one that achieves domain consistency and the weaker one.

Table 5.2 shows the results of our experiments using these 5 models. The model REGULAR_2 outperforms GR_1 in all benchmarks, whilst REGULAR_1 outperforms GR_1 in most of the benchmarks. The model REGULAR_2 also proves optimality in several instances of hard benchmarks. It should be noted that performing simplification before minimisation is essential. It significantly reduces the size of the encoding and speeds up MiniSat+. Finally, we note that the *PB* models consistently outperformed the CP models, in agreement with the observations of [QW07] . Between the two CP models, REGULAR_1^{CP} is significantly better than GR_1^{CP}, finding a better solution in many instances and proving optimality in two instances. In addition, although we do not show it in the table, GeCode is approximately three orders of magnitude faster per branch with the REGULAR_1^{CP} model. For instance, in benchmark number 2 with 1 activity and 4 workers, it explores approximately 80 million branches with the REGULAR_1^{CP} and 24000 branches with the GR_1^{CP} model within

Table 5.1: Shift Scheduling Problems. G_a is the acyclic grammar, NFA_a^ε is *NFA* with ε-transitions, NFA_a is *NFA* without ε-transitions, $\min(NFA_a)$ is minimised *NFA*, DFA is *DFA* obtained from $\min(NFA_a)$, $\min(DFA)$ is minimised DFA, $\#act$ is the number of activities, $\#$ is the benchmark number.

$\#act$	$\#$	G_a		NFA_a^ε		NFA_a		$\min(NFA_a)$		DFA		$\min(DFA)$	
		terms	prods	states	trans	states	trans	states	trans	states	trans	states	trans
1	2/3/8	4678	/ 9302	69050	/ 80975	29003	/ 42274	**3556**	/ **4505**	3683	/ 4617	3681	/ 4615
1	4/7/10	3140	/ 5541	26737	/ 30855	11526	/ 16078	**1773**	/ **2296**	1883	/ 2399	1881	/ 2397
1	5/6	2598	/ 4209	13742	/ 15753	5975	/ 8104	**1129**	/ **1470**	1215	/ 1553	1213	/ 1551
2	1/2/4	3777	/ 6550	42993	/ 52137	19654	/ 29722	**3157**	/ **4532**	3306	/ 4683	3303	/ 4679
2	3/5/6	**5407**	/ 10547	111302	/ 137441	50129	/ 79112	5975	/ **8499**	6321	/ 8846	6318	/ 8842
2	8/10	**6087**	/ 12425	145698	/ 180513	65445	/ 104064	7659	/ **10865**	8127	/ 11334	8124	/ 11330
2	9	4473	/ 8405	76234	/ 93697	34477	/ 53824	**4451**	/ **6373**	4691	/ 6614	4688	/ 6610

Table 5.2: Shift Scheduling Problems. GR_1 is the *PB* model with GRAMMAR, REGULAR_1 is the *PB* model with $\min(simplify(DFA))$, REGULAR_2 is the *PB* model with $\min(simplify(NFA))$, GR_1^{CP} is the CP model with GRAMMAR, REGULAR_1^{CP} is the CP model with $\min(simplify(DFA))$. We show **t**ime and number of **b**acktracks to prove optimality (the median time and the median number of backtracks for the *PB* encoding over **s**olved shuffled instances), number of **a**ctivities, the number of **w**orkers and the benchmark number $\#$.

			PB/MiniSat+										CSP/GeCode					
a	$\#$	w	GR_1				REGULAR_1				REGULAR_2				GR_1^{CP}			REGULAR_1^{CP}		
			cost	s	t	/ b	cost	s	t	/ b	cost	s	t	/ b	cost	t	/ b	cost	t	/ b
1	2	4	**26.00**	11	27	/ 8070	**26.00**	11	9	/ 11053	**26.00**	11	**4**	/ **7433**	26.75	-	/ -	**26.00**	-	/ -
1	3	6	**36.75**	11	530	/ 101560	**36.75**	11	94	/ 71405	**36.75**	11	**39**	/ **58914**	37.00	-	/ -	37.00	-	/ -
1	4	6	**38.00**	11	31	/ 16251	**38.00**	11	12	/ 10265	**38.00**	11	**6**	/ **7842**	**38.00**	-	/ -	**38.00**	-	/ -
1	5	5	**24.00**	11	5	/ 3871	**24.00**	11	2	/ 4052	**24.00**	11	**2**	/ **2598**	**24.00**	-	/ -	**24.00**	-	/ -
1	6	6	**33.00**	11	9	/ 5044	**33.00**	11	4	/ 4817	**33.00**	11	**3**	/ **4045**	-	-	/ -	**33.00**	-	/ -
1	7	8	**49.00**	11	22	/ 7536	**49.00**	11	9	/ 7450	**49.00**	11	**7**	/ 8000	**49.00**	-	/ -	**49.00**	-	/ -
1	8	3	**20.50**	11	13	/ 4075	**20.50**	11	4	/ 5532	**20.50**	11	**2**	/ **1901**	21.00	-	/ -	**20.50**	92	/ 2205751
1	10	9	**54.00**	11	242	/ 106167	**54.00**	11	111	/ **91804**	**54.00**	11	**110**	/ 109123	-	-	/ -	-	-	/ -
2	1	5	**25.00**	11	92	/ 35120	**25.00**	11	96	/ 55354	**25.00**	11	**32**	/ **28520**	**25.00**	-	/ -	**25.00**	90	/ 1289554
2	2	10	**58.00**	1	3161	/ **555249**	**58.00**	0	-	/ -	**58.00**	4	**2249**	/ 701490	-	-	/ -	**58.00**	-	/ -
2	3	6	**37.75**	0	-	/ -	**37.75**	1	3489	/ 590649	**37.75**	9	**2342**	/ **570863**	42.00	-	/ -	40.00	-	/ -
2	4	11	**70.75**	0	-	/ -	71.25	0	-	/ -	71.25	0	-	/ -	-	-	/ -	-	-	/ -
2	5	4	**22.75**	11	739	/ 113159	**22.75**	11	823	/ 146068	**22.75**	11	**308**	/ **69168**	23.00	-	/ -	23.00	-	/ -
2	6	5	**26.75**	11	86	/ 25249	**26.75**	11	153	/ 52952	**26.75**	11	**28**	/ **21463**	**26.75**	-	/ -	**26.75**	-	/ -
2	8	5	**31.25**	11	1167	/ 135983	**31.25**	11	383	/ 123612	**31.25**	11	**74**	/ **47627**	32.00	-	/ -	31.50	-	/ -
2	9	3	**19.00**	11	1873	/ 333299	**19.00**	11	629	/ 166908	**19.00**	11	**160**	/ **131069**	19.25	-	/ -	**19.00**	-	/ -
2	10	8	**55.00**	0	-	/ -	**55.00**	0	-	/ -	**55.00**	0	-	/ -	-	-	/ -	-	-	/ -

the 1 hour timeout.

5.4 Restrictions of the GRAMMAR constraint

In this section we investigate whether we can obtain an efficient propagator for the GRAMMAR$([X_1, \ldots, X_n], G)$ constraint if we restrict the context free grammar G to, for instance, deterministic and unambiguous context-free grammar. We want to point out that these restricted forms of context-free grammars permit quadratic parsing algorithms while still being more expressive than regular grammars. We show that we cannot exploit the properties of these grammars that make fast parsing algorithms possible. Hence, we answer this question negatively by showing that detecting disentailment for the GRAMMAR constraint in these cases is as hard as parsing an unrestricted context free grammar. We also consider the class of linear grammars and give a propagator that runs in quadratic time.

5.4.1 Simple Context-Free Grammars

In this section we show that propagating a *simple* context-free grammar constraint is at least as hard as parsing an (unrestricted) context-free grammar. A grammar G is *simple* if it is in Greibach form (Section 2.3.2), and for every non-terminal A and terminal a there is at most one production of the form $A \to a\alpha$. Hence, restricting ourselves to languages recognised by simple context-free grammars does not improve the complexity of propagating a global grammar constraint. Simple context-free languages are deterministic context-free languages (characterised by deterministic push-down automata), and also $LL(1)$ languages [Roz97], so this result also holds for propagating these more general classes of languages.

First, we encode variable domains as a regular language that contains all strings that can be formed using values in $D(X_1), \ldots, D(X_n)$. Given finite sets $D_1, \ldots, D_n$, their *Cartesian product language* $L(R_{D_1,\ldots,D_n})$ is the cross product of the domains $\{a_1 a_2 \cdots a_n \mid a_1 \in D_1, \ldots, a_n \in D_n\}$. We recall the definition of the GRAMMAR constraint (Definition 5.2). The GRAMMAR$([X_1, \ldots, X_n], G)$ constraint is true for an assignment to variables X if and only if the string formed by this assignment belongs to $L(G)$.

From Definition 5.2, we observe that finding a support for the grammar constraint is related to intersecting the context-free language with the Cartesian product language of the domains.

Proposition 5.1 *Let G be a context-free grammar, $X_1, \ldots, X_n$ be a sequence of variables with domains $D(X_1), \ldots, D(X_n)$. Then $L(G) \cap L(R_{D(X_1),\ldots,D(X_n)}) \neq 0$ if and only if*

GRAMMAR$([X_1, \ldots, X_n], G)$ *is satisfiable.*

Context-free grammars are closed under intersection with regular grammars. To see this, consider a context-free grammar G in Chomsky form and a regular grammar R. Following the "triple construction", the intersection grammar has non-terminals of the form $\langle F, A, F' \rangle$ where F, F' are non-terminals of R and A is a non-terminal of G. Intuitively, $\langle F, A, F' \rangle$ generates strings w that are generated by A and also by F, through a derivation from F to F'. If $A \to BC$ is a production of G, then we add, for all non-terminals F, F', F'' of R, the production $\langle F, A, F'' \rangle \to \langle F, B, F' \rangle \langle F', C, F'' \rangle$. If $A \to a$ is a production of G, then we add, for all non-terminals F, F' of R, the production $\langle F, A, F' \rangle \to \langle F, a, F' \rangle$. If $F \to aF'$ is a production of R, then we add $\langle F, a, F' \rangle \to a$. If $F \to a$ is a production of R, then we add $\langle F, a, F \rangle \to a$. The resulting grammar is $O(|G|n^3)$ in size where n is the number of non-terminals of R. This is similar to the construction of Theorem 6.5 in [HU79] which uses push-down automata instead of grammars. Since emptiness of context-free grammars takes linear time (cf. [HU79]) we obtain through Proposition 5.1 a cubic time algorithm to check whether a global constraint GRAMMAR$([X_1, \ldots, X_n], G)$ has support. In fact, this shows that we can efficiently propagate more complex constraints, such as the conjunction of a context-free with a regular constraint. Note that if R is a Cartesian product language then the triple construction generates the same result as the *CYK* based propagator for the GRAMMAR constraint [Sel06, QW06].

We now show that for *simple* context-free grammars G, detecting disentailment of the constraint GRAMMAR$([X_1, \ldots, X_n], G)$, i.e. testing whether it has a solution, is at least as hard as parsing an arbitrary context-free grammar.

Theorem 5.5 *Let G be a context-free grammar in Greibach form and s a string of length n. One can construct in $O(|G|)$ time a simple context-free grammar G' and in $O(|G|n)$ time a Cartesian product language $L(R_{D(X_1),\ldots,D(X_n)})$ such that $L(G') \cap L(R_{D(X_1),\ldots,D(X_n)}) \neq \emptyset$ if and only if $s \in L(G)$.*

Proof: The idea behind the proof is to determine an unrestricted context free grammar G by mapping each terminal in G to a set of pairs – the terminal and a production that can consume this terminal. This allows us to carry information about the derivation inside a string in G'. Then, we construct a Cartesian product language $L(R_{D(X_1),\ldots,D(X_n)})$ over these pairs so that all strings from this language map only to the string s. Let $G = \langle N, T, P, S \rangle$ and fix an arbitrary order of the productions in P. We now construct the grammar $G' = \langle N, T', P', S \rangle$. For every $1 \leq j \leq |P|$, if the jth production of P is $A \to a\alpha$ then let (a, j) be a new symbol in T' and let the production $A \to (a, j)\alpha$ be in P'. Next, we construct the Cartesian

product language. We define $D(X_i) = \{(a, j)|(s_i = a) \wedge (a, j) \in T'\}$, $i = 1, \ldots, n$ and s_i is the ith letter of s. Clearly, G' is constructed in $O(|G|)$ time and $L(R_{D(X_1),\ldots,D(X_n)})$ in $O(|P|n)$ time.

($\Rightarrow$) Let $L(G') \cap L(R_{D(X_1),\ldots,D(X_n)})$ be non empty. Then there exits a string s' that belongs to the intersection. Let $s' = (a_1, i_1) \cdots (a_n, i_n)$. By the definition of $L(R_{D(X_1),\ldots,D(X_n)})$, the string $a_1 a_2 \cdots a_n$ must equal s. Since $s' \in L(G')$, there must be a derivation by G of the form

$$S \Rightarrow_{G,p_1} a_1\alpha \Rightarrow_{G,p_2} a_1 a_2 \alpha' \cdots \Rightarrow_{G,p_n} a_1 \cdots a_n$$

where p_j is the jth production in P. Hence, $s \in L(G)$.

($\Leftarrow$) Let $s \in L(G)$. Consider a derivation sequence of the string s. We replace every symbol a in s that was derived by the ith production of G by (a, i). By the construction of G', the string s' is in $L(G')$. Moreover, s' is also in $L(R_{D(X_1),\ldots,D(X_n)})$. $\diamond$

Note that context-free parsing has a quadratic time lower bound, due to its connection to matrix multiplication [Lee02]. Given this lower bound and the fact that the construction of Theorem 5.5 requires only linear time, we can deduce the following.

Corollary 5.1 *Let G be a context-free grammar. If G is simple (or deterministic or $LL(1)$) then detecting disentailment of* GRAMMAR$([X_1, \ldots, X_n], G)$ *is at least as hard as context-free parsing of a string of length n.*

We now show the converse to Theorem 5.5 which reduces intersection emptiness of a context-free with a regular grammar, to the membership problem of context-free languages. This shows that the time complexity of detecting disentailment for the GRAMMAR constraint is the same as the time complexity of the best parsing algorithm for an arbitrary context free grammar. Therefore, our result shows that detecting disentailment takes $O(n^{2.4})$ time [CW90], as in the best known algorithm for Boolean matrix multiplication. It does not, however, improve the asymptotic complexity of a domain consistency propagator for the GRAMMAR constraint described in Section 5.2 [Sel06, QW06].

Theorem 5.6 *Let $G = \langle N, T, P, S\rangle$ be a context-free grammar and $L(R_{D(X_1),\ldots,D(X_n)})$ be Cartesian product language. One can construct in time $O(|G| + |T|^2)$ a context-free grammar G' and in time $O(n|T|)$ a string s such that $s \in L(G')$ if and only if $L(G) \cap L(R_{D(X_1),\ldots,D(X_n)}) \neq \emptyset$.*

Proof: *Construction of G'.* We assign an index to each terminal in T. For each position i of the strings of R, we create a bitmap of the alphabet that describes the terminals that may

appear in that position. The jth bit of the bitmap is 1 if and only if the symbol with index j may appear at position i. The string s is the concatenation of the bitmaps for each position and has size $n|T|$. First, we add $B \to 0$ and $B \to 1$ to G'. For each terminal in T with index j, we introduce $T_j \to B^{j-1}1B^{|T|-j}$ into G' to accept any bitmap with 1 at the jth position. Then, for each production in G of the form $A \to a\alpha$ such that the index of a is j, we add $A \to T_j\alpha$ to G'. In this construction, every production in G' except for those with T_i on the left hand side can be uniquely mapped to a production in G.

($\Rightarrow$) Suppose $s' \in L(G')$. Consider a derivation sequence of s'. We construct a string $s \in L(G) \cap L(R)$. Note that every string that belongs to $L(G')$ has length $k|T|$ for some k. Moreover, it is partitioned in blocks of $|T|$ symbols and every block of is generated by one of the non-terminals T_p. We construct s by placing at the jth position the symbol with index i if the jth block of symbols of s' was generated by the non-terminal T_i. Clearly this belongs to R. Then, a derivation of s in G can be created by removing from the derivation of s' productions with T_i on the left hand side and replacing the rest with the corresponding production in G, so $s \in L(G) \cap L(R)$ and $L(G) \cap L(R) \neq \emptyset$.

($\Leftarrow$) Suppose $L(G) \cap L(R) \neq \emptyset$ and let s be a string in the intersection. We construct a string s' that belongs to $L(G')$ by replacing the symbol with index i at position j of s with a block of $|T|$ symbols with 1 at the ith position and 0 elsewhere. We create a derivation of s' in G' by replacing each production $A \to a\alpha$ in it with the pair of productions $A \to T_i\alpha$, $T_i \to (0|1)^{i-1}1(0|1)^{|T|-i}$ where i is the index of a. Thus, $s \in L(G')$. $\diamond$

Theorems 5.5–5.6 show that, on the one hand, many well-studied restricted context-free grammars do not permit constructing a domain consistency propagator for the GRAMMAR constraint that is more efficient than the best parsing algorithm for an arbitrary context-free grammar. On the other hand, the best known propagator that detects disentailment of the GRAMMAR constraint and runs in $O(n^3)$ time can be improved using the best parsing algorithm.

5.4.2 Linear Context-Free Grammars

A context-free grammar is *linear* if every production contains at most one non-terminal in its right-hand side. Linear languages are a proper superset of regular languages and are a strict subset of context-free languages. Linear context-free grammars possess two important properties: (1) membership of a given string of length n can be checked in time $O(n^2)$ (see Theorem 12.3 in [WW86]), and (2) the class is closed under intersection with regular grammars (to see this, apply the "triple construction" as explained after Proposition 5.1).

The first property allows us to show that a *CYK*-based propagator for this type of grammars runs in quadratic time. This is then the third example of a grammar, besides regular and context-free grammars, where the asymptotic time complexity of the parsing algorithm and that of the corresponding propagator are equal. The second property opens the possibility of constructing a polynomial time propagator for a conjunction of the linear GRAMMAR and the REGULAR constraints that runs in $O(n^2)$ time.

Theorem 5.7 *Let G be a linear grammar and* GRAMMAR$([X_1, \ldots, X_n], G)$ *be the corresponding global constraint. There exists a domain consistency propagator for this constraint that runs in $O(n^2|G|)$ time.*

Proof: We convert $G = \langle N, T, P, S \rangle$ into Chomsky normal form (Section 2.3.2). Every linear grammar can be converted into the form $A \rightarrow aB$, $A \rightarrow Ba$ and $A \rightarrow a$, where $a, b \in T$ and $A, B \in N$ (see Theorem 12.3 of [WW86]) in $O(|G|)$ time. To obtain the Chomsky normal form we replace every terminal $a \in T$ that occurs in a production on the right hand side with a new non-terminal Y_a and introduce a production $Y_a \rightarrow a$.

We recall the time complexity analysis for the *CYK*-based domain consistency propagator for an arbitrary context-free grammar constraint (Section 5.2). The time complexity is bounded by all possible 1-step derivations from each non-terminal in the table and is equal to $O(|G|n^3)$. In contrast to unrestricted context-free grammars, the number of possible 1-step derivations from each of these non-terminals is bounded by $O(F(A))$ for a linear grammar as opposed to $O(F(A)n)$ in general case. Therefore, the propagator runs in $O(n^2|G|)$ for a linear grammar G. $\diamond$

It is possible to restrict linear grammars further, so that the resulting global constraint problem is solvable in *linear time*. As an example, consider "fixed-growth" grammars in which there exists l and r with $l + r \geq 1$ such that every production is of the form either $A \rightarrow w \in T^+$ or $A \rightarrow uBw$ where the length of $u \in T^*$ equals l and the length of $w \in T^*$ equals r. In this case, the triple construction (explained after Proposition 5.1) generates $O(|G|n)$ new non-terminals implying linear time propagation (similarly, *CYK* runs in linear time as it only generates non-terminals on the diagonal of the dynamic program). A special case of fixed-growth grammars are regular grammars which have $l = 1$ and $r = 0$ (or vice verse).

5.5 Generalisations of the GRAMMAR constraint

In this section we consider a generalisation to the GRAMMAR constraint — the weighted GRAMMAR constraint that is useful for modelling over-constrained problems and problems with preferences. Consider, for example, a shift scheduling problem where we have two types of shifts: day and night shifts. Suppose there is an employee who prefers day shifts to night shifts. To model these preferences we penalise with the unit weight night shift assignments to this employee. We also put a bound k on the total number of night shift assignments by fixing the upper bound of the cost variable z. In this way we make sure that the employee will work at most k night shifts in a schedule. In a similar way we can model the cost of employee incentives. Suppose the business pays extra salary s for a night shift on public holidays. To model this constraint we increase the weight of the corresponding production of the form $A \rightarrow a$ by s.

The WEIGHTEDGRAMMAR constraint can also be used to model bonus payments for specific activities. For example, the employer wants to provide incentives for employees to work longer hours. Suppose, the grammar that represents regulation rules includes the productions: $Work \rightarrow WorkWork|Work, Work \rightarrow a$ which is the case in our benchmark problems 5.3.7. Hence, an occurrence of a non-terminal $Work$ at the ith level of the dynamic programming table V indicates that an employee performed i consecutive activities. If, for example, the salary of an employee increases by s for each activity performed after the 5th consecutive activity, we can model this by increasing by s the weight of W at the 5th level or higher in the table V as we describe in Section 5.5.1.

For some forms of objective functions, we can use the WEIGHTEDGRAMMAR constraint to construct the conjunction of the Optimisations function and the GRAMMAR constraint without requiring an additional modelling step. This is considered in our empirical evaluation of the WEIGHTEDGRAMMAR constraint.

5.5.1 The weighted GRAMMAR constraint

In this section we propose a *DC* propagator for the WEIGHTEDGRAMMAR constraint. Let us recall that the weighted WEIGHTEDGRAMMAR$(G, W, z, [X_1, \ldots, X_n])$ constraint holds if and only if the assignment X forms a string belonging to the grammar G and the minimal weight of a derivation of X is less than or equal to z, where the matrix W defines weights of productions in the grammar G.

We propose a *DC* propagator for the WEIGHTEDGRAMMAR constraint based on an extension of the *CYK* parser to probabilistic grammars [Ney91] and inspired by the *DC*

propagator for the unweighted GRAMMAR constraint (Section 5.2). We assume that G is in Chomsky normal form and with a single start non-terminal S. The algorithm has two stages similar to Algorithm 5.1. In the first stage, we construct a dynamic programing table $V[i, j]$, $i, j = 1, \ldots, O(n)$, where an element A of $V[i, j]$ is a potential non-terminal that generates a substring $[X_i, \ldots, X_{i+j}]$. We compute a lower bound $l[i, j, A]$ on the minimal weight of a derivation from A. In the second stage, we move from $V[1, n]$ to the bottom of table V. For an element A of $V[i, j]$, we compute an upper bound $u[i, j, A]$ on the maximal weight of a derivation from A of a substring $[X_i, \ldots, X_{i+j}]$. We mark the element A if and only if $l[i, j, A] \leq u[i, j, A]$. The pseudo-code is presented in Algorithm 5.5. Lines 2–5 initialise l and u. Lines 6–16 compute the first stage, whilst lines 21–30 compute the second stage. Finally, we prune inconsistent values in lines 31–32. Algorithm 5.5 enforces domain consistency on variables X and bounds consistency on the cost variable z on the WEIGHTEDGRAMMAR$(G, W, z, [X_1, \ldots, X_n])$ constraint in $O(|G|n^3)$ time as the following Lemmas show.

We can further refine the granularity of the weights by making W a 3D matrix with *three* arguments $W(i, j, P)$, $i, j = 1, \ldots, O(n)$, $P \in G$, so that it gives the weight of using the production P to produce the substring starting at position i with length j. This is similar to the conditional productions used in [QW07] and the cost definition used in the COSTREGULAR constraint [DPR06]. In accordance with these previous uses, we call these *conditional weights*. In fact, we will use conditional weights in our experimental setup. For simplicity, however, we use the simpler definition of unconditional weights in the rest of this section. To incorporate this weight matrix we modify Algorithm 5.5 by replacing $W[A \to BC]$ with $W[A \to BC, i, j]$ in the corresponding lines.

First, we show that Algorithm 5.5 finds correct values of matrices l and u.

Lemma 5.1 *Let $WCYK$-alg compute the value $l[i, j, A]$. Then there is no derivation of a substring $[X_i, \ldots, X_{i+j}]$ from A, $A \in V[i, j]$ of weight less than $l[i, j, A]$.*

Proof: By induction on the length of derivation.

Length 1. Line 8 ensures that the value $l[i, 1, A]$ is the minimal weight of all productions of type $A \to a$ applied to values in the X_i domain.

Length p. Let the lemma's statement hold for all derivations of a substring of length p or less.

Length $p + 1$. Line 16 computes $l[i, p + 1, A]$ using values $l[r, t, B]$, $r \in [i, i + p + 1]$, $t < p + 1$, that are correct by the induction step. Hence, the weight $W[A \to BC] + l[i, k, B] + l[i+k, j-k, C]$ is the minimum possible weight if we apply production $A \to BC$

Algorithm 5.5 The weighted *CYK* propagator

```
1: procedure WCYK-ALG(G, W, z, [X_1, ..., X_n])
2:     for j = 1 to n do
3:         for i = 1 to n − j + 1 do
4:             for each A ∈ G do
5:                 l[i, j, A] = z + 1; u[i, j, A] = −1;
6:     for i = 1 to n do
7:         V[i, 1] = {A | A → a ∈ G, a ∈ D(X_i)}
8:         for A ∈ V[i, 1] s.t A → a ∈ G, a ∈ D(X_i) do
9:             l[i, 1, A] = min{l[i, 1, A], W[A → a]};
10:    for j = 2 to n do
11:        for i = 1 to n − j + 1 do
12:            V[i, j] = ∅;
13:            for k = 1 to j − 1 do
14:                V[i, j] = V[i, j] ∪ {A | A → BC ∈ G, B ∈ V[i, k], C ∈ V[i + k, j − k]}
15:                for each A → BC ∈ G s.t. B ∈ V[i, k], C ∈ V[i + k, j − k] do
16:                    l[i, j, A] = min{l[i, j, A], W[A → BC] + l[i, k, B] + l[i + k, j − k, C]};
17:    if S ∉ V[1, n] then
18:        return 0;
19:    lb(z) = max(lb(z), l[1, n, S]);
20:    mark (1, n, S); u[1, n, S] = ub(z);
21:    for j = n downto 2 do
22:        for i = 1 to n − j + 1 do
23:            for A such that (i, j, A) is marked do
24:                for k = 1 to j − 1 do
25:                    for each A → BC ∈ G s.t. B ∈ V[i, k], C ∈ V[i + k, j − k] do
26:                        if W[A → BC] + l[i, k, B] + l[i + k, j − k, C] > u[i, j, A] then
27:                            continue;
28:                        mark (i, k, B); mark (i + k, j − k, C);
29:                        u[i, k, B] = max{u[i, k, B], u[i, j, A] − l[i + k, j − k, C] − W[A → BC]};
30:                        u[i + k, j − k, C] = max{u[i + k, j − k, C], u[i, j, A] − l[i, k, B] − W[A → BC]};
31:    for i = 1 to n do
32:        D(X_i) = {a ∈ D(X_i) | A → a ∈ G, (i, 1, A) is marked and W[A → a] ≤ u[i, 1, A]};
33:    return 1;
```

such that B derives string of length k starting at position i and C derives the remaining part. Consequently, $l[i, p+1, A]$ computes the minimal weight of all possible derivation from A of length $p+1$ starting at ith position. $\diamond$

Lemma 5.2 *Let $WCYK$-alg compute the value $u[i, j, B]$. Then there is no derivation of $[X_1, \ldots, X_n]$ with weight less than or equal to z such that its substring $[X_i, \ldots, X_{i+j}]$ generated by A has weight greater than $u[i, j, B]$.*

Proof: By induction on the length of derivation.

Length n. The value $u[1, n, S] = z$ is correct.

Length p. Let the lemma's statement hold for all derivations of a substring of length p or greater.

Length $p-1$. Line 29 computes value $u[i, p-1, B]$, using values $l[i+p-1, t-(p-1), C]$, that are correct by Lemma 5.1, and $u[i, t, A]$, $p-1 < t$, that are correct by the induction step. If $u[i, p-1, B] = -1$ then there is no way to extend a derivation from A, $A \in V[i, p-1]$ by the condition in line 26. Consequently, the value $u[i, p-1, B]$ is correct. $\diamond$

Lemma 5.3 *Let $WCYK$-alg compute the value $l[i, j, A]$ such that $l[i, j, A] < z + 1$. Then there exists a substring $[X_i, \ldots, X_{i+j}]$ generated from A, $A \in V[i, j]$ of weight $l[i, j, A]$.*

Proof: By induction on the length of derivation.

Length 1. Line 8 ensures that the value $l[i, 1, A]$ is the minimal weight of all productions of type $A \to a$ applied to values in the X_i domain. Hence, there exist a value a in the domain of X_i and the production $A \to a$ with the weight $l[i, 1, A]$.

Length p. Let the lemma's statement hold for all derivations of a substring of length p or less.

Length $p + 1$. Line 16 ensures that there exist values $l[i, k, B]$, $l[i + k, (p + 1) - k, C]$ and a production $W[A \to BC]$ such that

$$l[i, p + 1, A] = W[A \to BC] + l[i, k, B] + l[i + k, (p + 1) - k, C].$$

By the induction step there exist derivations of the substring $[X_i, \ldots, X_{i+k}]$, $k < j$ from B of weight $l[i, k, B]$ and the substring $[X_{i+k}, \ldots, X_{(p+1)-k}]$, $(p+1)-k < (p+1)$ from C of weight $l[i+k, (p+1)-k, C]$. The substring $[X_i, \ldots, X_{i+k}]$ and $[X_{i+k}, \ldots, X_{(p+1)-k}]$ are disjoint. They can be derived from B and C, respectively, by the induction step. Moreover, there exists a production $A \to BC$. Hence, we can derive a string of the weight $l[i, p+1, A]$. $\diamond$

Lemma 5.4 *Let $WCYK$-alg compute the value $u[i, j, A]$ such that $l[i, j, A] \leq u[i, j, A]$ and $l[1, n, S] \leq u[1, n, S]$. Then there exists a derivation of $[X_1, \ldots, X_n]$ with a weight at most z and its substring $[X_i, \ldots, X_{i+j}]$ generated by A has a weight at most $u[i, j, A]$.*

Proof: As $l[i, j, A] \leq u[i, j, A]$, substring $[X_i, \ldots, X_{i+j}]$ can be generated from A with weight $l[i, j, A]$. We will show that this substring can be extended to a solution of WEIGHTEDGRAMMAR. Line 29 ensures that there exist values $u[i, k, B]$, $l[i + j, k - j, C]$ and a production $W[B \to AC]$ such that [2]

$$u[i, j, A] = u[i, k, B] - l[i + j, k - j, C] - W[B \to AC]$$

[2]The case $B \to CA \in G$(line 30), $B \in V[p, i + j - p], C \in V[p, i - p], A \in V[i, j]$ is similar

We will show that the substring $[X_i, \ldots, X_{i+j}]$ can be extended to a substring $[X_i, \ldots, X_{i+k}]$, $j < k$ and the weight of an extended substring derivation is less than or equal to $u[i, k, B]$.

Line 16 ensures that $l[i, k, B] \leq l[i, j, A] + l[i + j, k - j, C] + W[B \rightarrow AC]$. Also, $l[i, j, A] \leq u[i, j, A]$ by the statement of the lemma. Hence,

$$l[i, k, B] \leq u[i, j, A] + l[i + j, k - j, C] + W[B \rightarrow AC]$$

or

$$l[i, k, B] \leq u[i, k, B]$$

By Lemma 5.3, substring $[X_i, \ldots, X_{i+k}]$ can be derived from $B \in V[i, k]$ with weight $l[i, k, B]$.

Applying the same argument to the value $u[i, k, B]$, $l[i, k, B] \leq u[i, k, B]$, we move up to the start non-terminal S, $S \in V[1, n]$. $\diamond$

Theorem 5.8 *$WCYK$-alg enforces domain consistency on variables X and bounds consistency on the variable z on the* WEIGHTEDGRAMMAR$(G, W, z, [X_1, \ldots, X_n])$ *constraint.*

Proof: Suppose we are looking for a support for $X_i = a$. If the value a was not pruned by the algorithm then there exists a non-terminal A, $A \rightarrow a$, $A \in V[i, 1]$ that was marked. Hence, there exist a non-terminal C and a production $B \rightarrow AC$ such that $l[i, 1, A] + l[i + 1, j, C] + W[B \rightarrow AC] \leq u[i, j, B]$ (line 26). Moreover, $u[i, 1, A] \geq u[i, j, B] - l[i + 1, j - 1, C] - W[B \rightarrow AC]$. Consequently, $l[i, 1, A] \leq u[i, 1, A]$ holds for A. By Lemma 5.4, $X_i = a$ can be extended to a solution of the constraint.

Suppose the value a was pruned from $D(X_i)$. If a was pruned then there is no marked non-terminal A in $V[i, 1, A]$ such that $A \rightarrow a$. Consequently, there is no non-terminal C and production $B \rightarrow AC$ such that $l[i, 1, A] + l[i + 1, j, C] + W[B \rightarrow AC] \leq u[i, j, B]$. Consequently, $u[i, 1, A] = -1$. By Lemma 5.2 $X_i = a$ cannot be extended to a solution.

Lemma 5.3 together with line 19 guarantees bounds consistency on the variable z.

$\diamond$

The time complexity of the $WCYK$-alg algorithm is dominated by lines 21 – 30 and equals $O(|G|n^3)$.

5.5.2 Decomposition of the weighted GRAMMAR constraint

As an alternative to this monolithic propagator, we propose a simple decomposition with which we can also enforce domain consistency. The idea of the decomposition is to in-

troduce arithmetic constraints to compute l and u. Given the table V obtained by Algorithm 5.5, we construct the corresponding AND/OR directed acyclic graph as in [QW07]. We label an OR node by $n(i, j, A)$, and an AND node by $n(i, j, k, A \rightarrow BC)$. We denote the parents of a node nd as $PRT(nd)$ and the children as $CHD(nd)$. For each node two integer variables are introduced to compute l and u. For an OR-node nd, these are $l_O(nd)$ and $u_O(nd)$, whilst for an AND-node nd, these are $l_A(nd)$, $u_A(nd)$.

For each AND node $nd = n(i, j, k, A \rightarrow BC)$ we post a constraint to connect nd to its children $CHD(nd)$:

$$l_A(nd) = \sum_{n_c \in CHD(nd)} l_O(n_c) + W[A \rightarrow BC] \tag{5.2}$$

For each OR node $nd = n(i, j, A)$ we post a set of constraints to connect nd to its parents $PRT(nd)$ and siblings:

$$u_A(nd) = u_O(n_p),\ n_p \in PRT(nd) \tag{5.3}$$

For each OR node $nd = n(i, j, A)$ we post constraints to connect nd to its children $CHD(nd)$:

$$l_O(nd) = \min_{n_c \in CHD(nd)} \{l_A(n_c)\} \tag{5.4}$$

For each OR node $nd = n(i, j, A)$ we post a set of constraints to connect nd to its parents $PRT(nd)$ and siblings:

$$u_O(nd) = max_{n_p \in PRT(nd)} \{u_A(n_p) - l_O(n_{sb}) - W[P]\}, \tag{5.5}$$

where $P = B \rightarrow AC$ or $B \rightarrow CA$, $n_p = n(r, q, t, P)$ is the parent of $nd = n(i, j, A)$ and $n_{sb} = n(i_1, j_1, C)$.

Finally, we introduce constraints to prune X_i. For each leaf of the DAG that is an OR node $nd = n(i, 1, a)$, we introduce:

$$a \in D(X_i) \Rightarrow 0 \leq l_O(nd) \leq z \tag{5.6}$$

$$a \notin D(X_i) \Leftrightarrow l_O(nd) > z \tag{5.7}$$

$$l_O(nd) > u_O(nd) \Rightarrow a \notin D(X_i) \tag{5.8}$$

As the maximal weight of a derivation is less than or equal to z we post:

$$u_O(n(1, n, S)) \leq z \tag{5.9}$$

Bounds propagation will set the lower bound of $l_O(n(i, j, A))$ to the minimal weight of a derivation from A, and the upper bound on $u_O(n(i, j, A))$ to the maximum weight

of a derivation from A. We forbid branching on variables $l_{A|O}$ and $u_{A|O}$ as branching on $l_{A|O}$ would change the weights matrix W and branching on $u_{A|O}$ would add additional restrictions to the weight of a derivation (see Section 3.3). Bounds propagation on this decomposition enforces domain consistency on the WEIGHTEDGRAMMAR constraint because the decomposition constraints mimic the computation of Algorithm 5.5. If we invoke constraints in the decomposition in the same order as we compute the table V, this takes $O(n^3|G|)$ time. For simpler grammars, propagation is faster. For instance, as in the unweighted case, it takes just $O(n|G|)$ time on a regular grammar.

We can speed up propagation by recognising when constraints are entailed. If $l_O(nd) > u_O(nd)$ holds for an OR node nd then constraints (5.5) and (5.4) are entailed. If $l_A(nd) > u_A(nd)$ holds for an AND node nd then constraints (5.2) and (5.3) are entailed. To model entailment we augmented each of these constraints in such a way that if $l_O(nd) > u_O(nd)$ or $l_A(nd) > u_A(nd)$ hold then corresponding constraints are not invoked by the solver.

5.5.3 The Soft GRAMMAR constraint

We can use the WEIGHTEDGRAMMAR constraint to encode a soft version of GRAMMAR constraint which is useful for modelling over-constrained problems. The soft GRAMMAR$(G, z, [X_1, \ldots, X_n])$ constraint holds if and only if the string $[X_1, \ldots, X_n]$ is at most distance z from a string in G. We consider both Hamming and edit distances. We denote GRAMMAR$_h(G, z, [X_1, \ldots, X_n])$ the soft GRAMMAR constraint with the Hamming distance violation measure and GRAMMAR$_e(G, z, [X_1, \ldots, X_n])$ the soft GRAMMAR constraint with the edit distance violation measure.

We encode the soft GRAMMAR$_h(G, z, [X_1, \ldots, X_n])$ constraint as a weighted GRAMMAR$(G', W, z, [X_1, \ldots, X_n])$ constraint. The G' grammar contains G and for each production $A \to a \in G$, we introduce additional unit weight productions to simulate substitution:

$$G' = G \cup \{A \to b, W[A \to b] = 1 | A \to a \in G, A \to b \notin G, b \in \Sigma\}$$

All productions from G have zero weight.

Theorem 5.9 *The* WEIGHTEDGRAMMAR$(G', W, z, [X_1, \ldots, X_n])$ *constraint is equivalent to the* GRAMMAR$_h(G, z, [X_1, \ldots, X_n])$ *constraint.*

Proof: Consider a solution X of GRAMMAR$_h(G, z, [X_1, \ldots, X_n])$ with cost k. X can be transformed into a string S, $S \in G$ with k substitutions. Hence, the string S can be derived

in G' with zero weight. Now we transform the string S back to X using productions of G'. If the symbol $X_i = b$ was substituted by the symbol a in S, we replace $A \to a$ in the parsing tree of S with $A \to b$ increasing the weight of the derivation by one. Note that this replacement affects only a leaf of the parsing tree, hence nothing else has to be changed in the parsing tree after the replacement. The remaining substitution in S can be done in the similar way. Consequently, X can be derived in G' with weight k.

Consider a solution X of WEIGHTEDGRAMMAR$(G', W, z, [X_1, \ldots, X_n])$ with weight k. If we replace productions of the type $A \to b$ with weight one with $A \to a$ with weight zero, then the weight of a derivation is decreased by one. This replacement corresponds to the substitution of the value a in the domain of X_i with b. There are exactly k productions of this type in the parsing tree of X in G', consequently, X is a solution of GRAMMAR$_h(G, z, [X_1, \ldots, X_n])$ with cost k. $\diamond$

For edit distance, we use the same idea of an encoding of the soft GRAMMAR$_e(G, z, [X_1, \ldots, X_n])$ constraint as a weighted GRAMMAR$(G'', W, z, [X_1, \ldots, X_n])$ constraint. The G'' grammar contains G and additional productions to simulate substitution, insertion and deletion:

$$G'' = G \cup \{A \to b, W[A \to b] = 1 | A \to a \in G, A \to b \notin G, b \in \Sigma\} \cup$$
$$\{A \to \varepsilon, W[A \to \varepsilon] = 1 | A \to a \in G, a \in \Sigma\} \cup$$
$$\{A \to Aa, W[A \to Aa] = 1 | a \in \Sigma\} \cup$$
$$\{A \to aA, W[A \to aA] = 1 | a \in \Sigma\}$$

To handle ε productions we modify Algorithm 5.5 so loops in lines (13),(24) run from 0 to j. All productions from G have zero weight.

Theorem 5.10 *The* WEIGHTEDGRAMMAR$(G'', W, z, [X_1, \ldots, X_n])$ *constraint is equivalent to the* GRAMMAR$_e(G, z, [X_1, \ldots, X_n])$ *constraint.*

Proof: Similar to the proof of Theorem 5.9. The only difference is in insertion and deletion operations that are used in the transformation of S back to X.

If the symbol b was deleted before the symbol c in a transformation from X to S then we insert b before c in S. Namely, the production $A \to c$ in the parsing tree of S is replaced with $A \to Ac \to bc$ increasing the weight of the derivation by one.

If the symbol b was inserted in the transformation from X to S then we delete b in S. Namely, the production $A \to b$ in the parsing tree of S is replaced with $A \to \varepsilon$ increasing the weight of the derivation by one.$\diamond$

5.5.4 The EDITDISTANCE constraint

In this section we present another application of the weighted GRAMMAR constraint. We show how to encode an edit distance constraint into such a grammar. In particular, we show that we can use linear weighted GRAMMAR constraint(Section 5.4.2). We recall that the edit distance between two strings is the minimum number of deletion, insertion and substitution operations required to convert one string into another. Each of these operations can change one symbol in a string. W.l.o.g. we assume that $n = m$.

We recall the definition of the EDITDISTANCE constraint: EDITDISTANCE$([X_1, \ldots, X_n], [Y_1, \ldots, Y_m], N)$ holds if and only if the edit distance between assignments of two sequences of variables $\mathbf{X}$ and $\mathbf{Y}$ is less than or equal to N.

We will show that the EDITDISTANCE constraint can be encoded as a weighted form of the linear GRAMMAR constraint. The idea of the encoding is to parse matching substrings using productions of weight 0 and to parse edits using productions of weight 1.

We convert EDITDISTANCE$([\mathbf{X}], [\mathbf{Y}], N)$ into a linear WEIGHTEDGRAMMAR$([\mathbf{Z}_{2n+1}, N, G_{ed})$ constraint. The first n variables in the sequence $\mathbf{Z}$ are equal to the sequence $\mathbf{X}$, the variable Z_{n+1} is ground to the sentinel symbol $\#$ so that the grammar can distinguish the sequences $\mathbf{X}$ and $\mathbf{Y}$, and the last n variables of the sequence $\mathbf{Z}$ are equal to the reverse of the sequence $\mathbf{Y}$. We define the linear weighted grammar G_{ed} as follows.

$$S \rightarrow dSd \qquad w = 0 \qquad \forall d \in \mathbf{D}(\mathbf{X}) \cup \mathbf{D}(\mathbf{Y}) \tag{5.10}$$

$$|d_1 S d_2 \qquad w = 1 \qquad \forall d_1 \in \mathbf{D}(\mathbf{X}), d_2 \in \mathbf{D}(\mathbf{Y}), d_1 \neq d_2 \tag{5.11}$$

$$|dS|Sd \qquad w = 1 \qquad \forall d \in \mathbf{D}(\mathbf{X}) \tag{5.12}$$

$$|\# \qquad w = 0, \tag{5.13}$$

where $\mathbf{D}(X) = \cup_{i=1}^{n} D(X)$. Intuitively, rule (5.10) captures matching terminals, rule (5.11) captures replacement, rules (5.12) capture insertions and deletions. It is easy to show equivalence of these two constraints.

The propagator for the linear WEIGHTEDGRAMMAR constraint takes $O(n^2|G|)$ time since the arguments of Theorem 5.7 also apply to a CYK-based propagator for linear WEIGHTEDGRAMMAR. Down a branch of the search tree, the time complexity is $O(n^2|G|ub(N))$.

We can use this encoding of the EDITDISTANCE constraint into a linear WEIGHTEDGRAMMAR constraint to construct propagators for more complex constraints.

For instance, we can exploit the fact that linear grammars are closed under intersection with regular grammars to propagate efficiently the conjunction of an EDITDISTANCE constraint and REGULAR constraints on each of the sequences **X**,**Y**. More formally, let **X** and **Y** be two sequences of variables of length n subject to the constraints REGULAR$(\mathbf{X}, R_1)$, REGULAR$(\mathbf{Y}, R_2)$ and EDITDISTANCE$([\mathbf{X}], [\mathbf{Y}], N)$. We construct a domain consistency propagator for the conjunction of these three constraints, by computing a grammar that generates strings of length $2n + 1$ which satisfy the conjunction. First, we construct an automaton that accepts $\mathcal{L}(R_1)\#\mathcal{L}(R_2)^R$. Since regular languages are closed under concatenation and reversal, this language is also regular and requires an automaton of size $O(|R_1| + |R_2|)$. Second, we intersect this with the linear weighted grammar that encodes the EDITDISTANCE constraint using the "triple construction". The size of the obtained grammar is $G_\wedge = |G_{ed}|(|R_1| + |R_2|)^2$ and this grammar is a weighted linear grammar. Therefore, we can use the linear WEIGHTEDGRAMMAR$(\mathbf{Z}, N, G_\wedge)$ constraint to encode the conjunction. Note that the size of $G_\wedge$ is only quadratic in $|R_1| + |R_2|$, because G_{ed} is a linear grammar. The time complexity to enforce domain consistency on this conjunction of constraints is $O(n^2|G_\wedge|) = O(n^2d^2(|R_1| + |R_2|)^2)$ for each invocation and $O(n^2d^2(|R_1| + |R_2|)^2 ub(N))$ down a branch of the search tree.

5.5.5 Other related work

Cost based GRAMMAR constraints (CFGC) were independently proposed by Kadioglu and Sellmann in [KS08] The difference between this work and ours is that in [KS08] weights can only be placed on individual terminals. Here, we allow weights to be placed on any production and therefore WEIGHTEDGRAMMAR is more expressive than CFGC. For instance, we can express the Soft GRAMMAR constraint with the edit distance violation measure using the weighted GRAMMAR constraint, but we cannot encode this constraint with the CFGC constraint.

Demassey, Pesant and Rousseau proposed the cost REGULAR constraint [DPR06] that shares a lot of advantages of the weighted GRAMMAR constraint. The propagator is based on the same idea as Pesant's propagator for the REGULAR constraint. The propagator builds an unfolded layered automaton. In addition, it associates a cost with each transition in the unfolded graph based on the cost matrix. It was shown that there is a bijection between solutions of COSTREGULAR and paths in the corresponding weighted unfolded automaton of weight at most $ub(z)$. However, as Example 5.3 demonstrated, there exist languages that can be expressed by polynomial sized context free grammars but the smallest regular

language that accepts these languages is exponential in size.

5.5.6 Experimental results

In this section, we evaluate the impact of the WEIGHTEDGRAMMAR constraint on some shift scheduling benchmarks. We show that the weighted GRAMMAR constraint is efficient for solving these problems, because it allows propagating the conjunction of GRAMMAR and the objective function of a shift scheduling problem. We used GeCode 2.0.1 for our experiments and ran them on an Intel Xeon 2.0Ghz machine with 4Gb of RAM.

We use the same shift scheduling problems that we introduced in Section 5.3.7 and use the same CP basic model. We replace the GRAMMAR constraint with the WEIGHTEDGRAMMAR constraint.

In the first set of experiments, we used WEIGHTEDGRAMMAR$(G, z_j, \mathbf{X})$, $j = 1, \ldots, m$ with zero weights. We investigate whether the filtering algorithm for the WEIGHTEDGRAMMAR constraint introduces an overhead compared to the unweighted GRAMMAR constraint. For this purpose, we define the weight function with conditional weights as follows

$$W(P, i, j) = \begin{cases} 0 & if \;\; f(i,j) = 1 \\ +\infty & if \;\; f(i,j) = 0 \end{cases}$$

Our monolithic propagator gave similar results to the unweighted GRAMMAR propagator from [QW07]. Decompositions of the WEIGHTEDGRAMMAR constraint were slower than decompositions of the unweighted GRAMMAR constraint as the former uses integers instead of Booleans. However, using the WEIGHTEDGRAMMAR constraint did not lead to a significant slowdown even if it does not achieve any extra pruning compared to GRAMMAR.

In the second set of experiments, we investigate whether propagating the conjunction of the GRAMMAR constraint and the objective function can achieve additional pruning. In order to do this, we augment the first model with additional constraints. We assigned weights to the productions of the grammar as described in Section 5.5.1. Specifically, we assigned weight 1 to activity productions, like $A_i \rightarrow a_i$. The objective function is thus reduced to $\sum_{j=1}^{m} z_j$, where z_j is the cost variables for the WEIGHTEDGRAMMAR$(G, z_j, \mathbf{X})$ constraint, $j = 1, \ldots, m$. Note $\sum_{j=1}^{m} z_j = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{a} b(i, j, a_k)$ is the number of slots in which employees worked. Results are presented in Table 5.3.

As can be seen from Table 5.3, we improved on the best solution found in the first model in 4 benchmarks and proved optimality in one. This shows that propagating the conjunction of the GRAMMAR constraints and the objective function pays off in terms of the runtime

improvements. The decomposition of the WEIGHTEDGRAMMAR constraint was slightly slower than the monolithic propagator. As was shown in Section 5.5.2, the time complexity of the decomposition is the same as the complexity of the monolithic propagator if the decomposition constraints are invoked in a particular order. However, we cannot enforce this ordering in the solver. It was also pointed out that some constraints become irrelevant during the search, but they are not entailed. Therefore, they present an overhead. We eliminate this redundant computation by annotating these constraints with an additional entailment test. If this test succeeds the solver ignores them in the remainder of the current branch (Section 5.5.2). Table 5.3 shows that redundant computation contributes a significant overhead and the additional entailment test improves performance in most cases, making the decomposition slightly faster than the monolithic propagator.

5.6 Conclusions

In this chapter we addressed three important problems for the GRAMMAR constraint. First of all, we showed that, unlike parsing, restrictions on context-free grammars such as determinism do not improve the efficiency of propagation of the corresponding global GRAMMAR constraint. This negatively resolves an open question stated by Meinolf Sellmann [Sel06] regarding unambiguous GRAMMAR constraint. On the other hand, one specific syntactic restriction, that of linearity, allows propagation in quadratic time. We demonstrated an application of such a restricted grammar in encoding the EDITDISTANCE constraint and more complex constraints. Second, we have shown how to transform a GRAMMAR constraint into a REGULAR constraint. In the worst case, the transformation may increase the space required to represent the constraint. However, in practice, we observed that such transformation reduces the space required to represent the constraint and speeds up propagation. We argued that transformation also permits us to compress the representation using standard techniques for automaton minimisation. We proved that minimising such automata after they have been unfolded and domains initially reduced can give automata that are exponentially more compact than those obtained by minimising before unfolding and reducing. Experimental results demonstrated that such transformations can improve the size of rostering problems that can be solved. Thirdly, we have introduced a weighted form of the GRAMMAR constraint. The GRAMMAR constraint with weights permits us to model over-constrained problems and problems with preferences. We proposed propagators for the WEIGHTEDGRAMMAR constraint based on the CYK parser. We also proposed a decomposition of the WEIGHTEDGRAMMAR constraint into

Table 5.3: All benchmarks have a one-hour time limit. $|A|$ is the number of activities,m is the number of employees, $cost$ shows the total number of slots in which employees worked in the best solution, $time$ is the time to find the best solution, bt is the number of backtracks to find the best solution, BT is the number of backtracks in one hour, Opt shows if optimality is proved, Imp shows if a lower cost solution is found by the second model

			Monolithic				Decomposition				Decomposition+entailment					
$\|A\|$	#	m	cost	time	bt	BT	cost	time	bt	BT	cost	time	bt	BT	Opt	Imp
1	2	4	**107**	**5**	**0**	8652	**107**	7	**0**	5926	**107**	7	**0**	**11521**		
1	3	6	**148**	**7**	**1**	5917	**148**	34	**1**	1311	**148**	9	**1**	**8075**		
1	4	6	**152**	1836	**5831**	11345	**152**	**1379**	**5831**	**14815**	**152**	1590	**5831**	13287		
1	5	5	**96**	6	**0**	8753	**96**	6	**0**	2660	**96**	**3**	**0**	**45097**		
1	6	6	–	–	–	10868	**132**	3029	**11181**	13085	**132**	**2367**	**11181**	**16972**		
1	7	8	**196**	16	**16**	10811	**196**	18	**16**	6270	**196**	**15**	**16**	**10909**		
1	8	3	**82**	11	**9**	**66**	**82**	13	**9**	**66**	**82**	**5**	**9**	**66**	√	√
1	10	9	–	–	–	10871	–	–	–	9627	–	–	–	**18326**		
2	1	5	**100**	523	**1109**	7678	**100**	634	**1109**	6646	**100**	**90**	**1109**	**46137**		
2	2	10	–	–	–	**11768**	–	–	–	10725	–	–	–	6885		
2	3	6	**165**	3517	**9042**	9254	168	2702	4521	6124	**165**	**2856**	**9042**	**11450**		√
2	4	11	–	–	–	**8027**	–	–	–	6201	–	–	–	5579		
2	5	4	**92**	**37**	**118**	**12499**	**92**	59	**118**	6332	**92**	49	**118**	10329		
2	6	5	**107**	**9**	**2**	6288	**107**	22	**2**	1377	**107**	14	**2**	**7434**		
2	8	5	**126**	422	**1282**	12669	**126**	1183	**1282**	3916	**126**	**314**	**1282**	**16556**		√
2	9	3	**76**	1458	**3588**	8885	**76**	2455	**3588**	5313	**76**	**263**	**3588**	**53345**		√
2	10	8	–	–	–	3223	–	–	–	3760	–	–	–	**8827**		

simple arithmetic constraints. Experiments on a shift-scheduling benchmark suggest that the WEIGHTEDGRAMMAR constraint has promise for solving real-world problems.

Chapter 6

The ALL-DIFFERENT constraint and Generalisations

6.1 Introduction

In modelling timetabling, scheduling and other problems we often need to specify the restriction that a set of variables takes pairwise distinct values. Consider, for example, a timetabling problem where a number of students have to take exams during an exam session period. We want to construct an exam timetable such that none of the students has to take two exams on the same day. Another example is a job-shop scheduling problem, where there exists a set of tasks and a set of machines that can process these tasks. We want to ensure that at any time point one machine processes at most one job. To encode this type of restrictions the ALL-DIFFERENT constraint was introduced [Lau78].

Definition 6.1 *The* ALL-DIFFERENT$([X_1, \ldots, X_n])$ *constraint is satisfied if and only if* $X_i \neq X_j$*, for* $i, j = 1, \ldots, n$*,* $i \neq j$*.*

A natural generalisation of the ALL-DIFFERENT constraint is the overlapping ALL-DIFFERENT constraint which allows to reason about two ALL-DIFFERENT constraints simultaneously. For example, in timetabling problems, we might have two students who have some common exams and some unique exams. The overlapping ALL-DIFFERENT constraint can be used to encode a timetabling problem for both students .

Definition 6.2 *The* OVERLAPPINGALLDIFF$([X_1, \ldots, X_n], S, T)$ *constraint where* $S \subseteq X$*,* $T \subseteq X$*,* $S \cup T = X$ *holds if and only if* ALL-DIFFERENT$([S])$ *and* ALL-DIFFERENT$([T])$ *hold simultaneously .*

Another generalisation of the ALL-DIFFERENT constraint is the GCC constraint, that restricts the number of occurrences of some values to be within an interval [Reg96]. The GCC constraint is useful in modelling shift scheduling problems, job shop scheduling problems and in many other domains. For example, in the shift scheduling problem described in Section 5.3.7, the GCC constraint can be used to model labour demand constraints, such as between two and three employees have to perform a certain activity .

Definition 6.3 *The* $\text{GCC}([X_1, \ldots, X_n], [l_1, \ldots, l_d], [u_1, \ldots, u_d])$ *constraint holds if and only if* $|\{i|X_i = j\}| \in [l_j, u_j]$, $j = 1, \ldots, m$.

A third generalisation of the ALL-DIFFERENT constraint is the NVALUE constraint. Pachet and Roy proposed this global constraint to model a combinatorial problem in selecting musical play-lists [PR99]. The NVALUE constraint restricts the number of occurrences of different values taken by a set of variables :

Definition 6.4 *The* $\text{NVALUE}([X_1, \ldots, X_n], N)$ *constraint holds if and only if* $N = |\{X_i \mid 1 \leq i \leq n\}|$.

The NVALUE constraint was also used to model the problem of species differentiation [Bue10].

ALL-DIFFERENT and its generalisations are among the most useful constraints in constraint programming toolkits. Therefore, considerable effort has been invested in developing efficient propagation algorithms for these constraints [Reg94, Lec96, Pug98, MT00, LOQTvB03, Reg96, QLOvBG04, PR99].

In this chapter we study the decomposability of these constraints and their filtering algorithms. Our main contribution is to show that existing range consistency and bounds consistency filtering algorithms for these constraints can be reformulated using a set of primitive arithmetic constraints, like ternary sum constraints or linear arithmetic constraints. We prove that these decompositions do not hinder propagation. Chapter 7 shows a complementary result that domain consistency algorithms for the ALL-DIFFERENT and GCC constraints cannot be polynomially decomposed into a set of primitive constraints. Enforcing domain consistency on the OVERLAPPINGALLDIFF and NVALUE constraints is NP-hard which, assuming P$\neq$NP, implies that a polynomial size decomposition for these global constraints cannot exist without hindering propagation [EKKM05, BHH$^+$05]. -

In this chapter we make the following contributions:

- propose range consistency and bounds consistency decompositions of the ALL-DIFFERENT constraint (Section 6.3).
- draw a connection between ALL-DIFFERENT and SEQUENCE constraints (Section 6.4), that shows that the ALL-DIFFERENT constraint can be encoded as the SEQUENCE constraint.
- propose a decomposition that detects bounds disentailment for the OVERLAPPINGALLDIFF constraint (Section 6.7).
- propose range consistency and bounds consistency decompositions of the GCC constraint (Section 6.8.2)
- present an extension of Hall's theorem for maximum matching in a graph (Section 6.9.1.3).
- propose range consistency and bounds consistency decompositions of the NVALUE constraint (Section 6.9.2).
- experimentally evaluate performance of these decompositions on some combinatorial and random problems and analyse their advantages and limitations (Section 6.5, Section 6.7.4 and Section 6.9.5).

6.2 Filtering algorithms for the ALL-DIFFERENT constraint

The ALL-DIFFERENT constraint is one of the oldest and most useful global constraints available to the constraint programmer. A large number of filtering algorithms have been proposed for the ALL-DIFFERENT constraint, including propagators that enforce domain consistency [Reg94, GMN08], range consistency [Lec96] and bounds consistency [Pug98, MT00, LOQTvB03]. In this section we show that many of these monolithic filtering algorithms can be decomposed into a set of primitive constraints. Since a domain consistency propagator for the ALL-DIFFERENT constraint cannot be polynomially decomposed into a set of primitive constraints (Chapter 7), we focus on decomposing range consistency and bounds consistency filtering algorithms , . Let us introduce a running example:

Example 6.1 (Running example (ALL-DIFFERENT)) *Suppose we have five conference tracks and five large rooms in a conference centre. Each conference track has some specific requirements for a conference room, like capacity, availability of video and audio systems, etc, so that the first track can be held only in rooms number* $\{3, 4, 5\}$*, the second in rooms* $\{1, 2, 3, 4, 5\}$*, the third in rooms* $\{3, 4\}$*, the fourth in rooms* $\{2, 3, 4, 5\}$ *and the fifth track can be held only in the 1-st room. The goal is to construct a schedule for the conference so that all tracks run in different rooms. To encode this problem we introduce five finite domain variables,* $[X_1, \ldots, X_5]$*, where the* i*th variables represents* i*th conference track. The domain of* X_i *contains all suitable rooms where the* i*th track can be held. The following matrix shows the domains of the variables.*

	1	2	3	4	5
X_1			$*$	$*$	
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

◇

The ALL-DIFFERENT$[X_1, X_2, X_3, X_4, X_5]$ constraint ensures that each conference track runs in a separate room.

As we only work with range consistency and bounds consistency filtering algorithms and bounds supports, we assume that the domains of all variables are intervals. We also denote the set of lower bounds of variables X as $LB = \bigcup_{i=1}^{n} \{lb(D(X_i))\}$ and the set of upper bounds of variables X as $UB = \bigcup_{i=1}^{n} \{ub(D(X_i))\}$

Existing bounds consistency and range consistency algorithms for the ALL-DIFFERENT constraint are based on the notion of a *Hall interval* , which is a special case of a *Hall set* [Hal35]. A Hall set is a set of d domain values that completely contains the domains of d variables. Then a Hall interval is a Hall set that contains consecutive values. More formally,

Definition 6.5 (Hall set [Hal35]) *A set* S *is a Hall set if and only if* $|\{i \mid D(X_i) \subseteq S\}| = |S|$.

Definition 6.6 (Hall interval) *An interval* $[l, u]$ *is a Hall interval if and only if* $|\{i \mid D(X_i) \subseteq [l, u]\}| = u - l + 1$.

Based on the notion of Hall interval we can derive necessary and sufficient conditions for range consistency and bounds consistency for the ALL-DIFFERENT constraint. Informally, the following theorems state that variables whose domains are contained within the Hall interval consume all the values in the Hall interval, whilst any other variables ranges of values (bounds) must find their support outside the Hall interval.

Theorem 6.1 (Bounds disentailment [Pug98]) *The* ALL-DIFFERENT *constraint is satisfiable if and only if*

1. $D(X_i) \neq \emptyset$, $i = 1, \ldots, n$ *and*
2. *for each interval* $[l, u]$, $|D(X_i) \subseteq [l, u]| \leq u - l + 1$,

where $l \in LB$ *and* $u \in UB$.

Theorem 6.2 (Range consistency) *The* ALL-DIFFERENT *constraint is range consistent if and only if*

1. *Theorem 6.1 conditions 1– 2 hold and*
2. *for each Hall interval of values* $[l, u]$ *and for each* X_i*:*

   if $D(X_i)$ *is not fully contained in the interval* $[l, u]$ *then* $D(X_i) \cap [l, u] = \emptyset$, $i = 1, \ldots, n$.

Theorem 6.3 (Bounds consistency) *The* ALL-DIFFERENT *constraint is bounds consistent if and only if*

1. *Theorem 6.1 conditions 1– 2 hold and*
2. *for each Hall interval of values* $[l, u]$ *and for each* X_i*:*

   if $D(X_i)$ *is not fully contained in the interval* $[l, u]$ *then* $\{lb(D(X_i)), ub(D(X_i))\} \notin [l, u]$, $i = 1, \ldots, n$.

Leconte [Lec96] proposed the first range consistency filtering algorithm for the ALL-DIFFERENT constraint that is based on identification of Hall intervals. Algorithm 6.1 shows the pseudocode for this algorithm (the pseudocode is from [Qui06]). This algorithm sorts variables twice, by the upper bounds and by the lower bounds. The outer loop iterates over variables sorted by the upper bounds. For each variable the algorithm iterate over variables sorted by the lower bounds to detect Hall intervals. The values in Hall intervals are removed from domains of variables that are not completely contained inside one of these intervals. The time complexity of the algorithm is $\Theta(n^2)$.

Algorithm 6.1 Leconte's algorithm

```
1:  procedure ALLDIFF-RC([X_1, ..., X_n])
2:      for X_j in decreasing order of ub(D(X_j)) do
3:          count = 0;
4:          k = -1;
5:          for X_i in decreasing order of lb(D(X_i)) do
6:              if k ≥ 0 then
7:                  D(X_i) = D(X_i) \ [lb(D(X_k), ub(D(X_j))] ;
8:              if ub(D(X_i) ≤ ub(D(X_j)) then
9:                  count = count + 1;
10:                 if count = ub(D(X_j)) - lb(D(X_i)) + 1 then
11:                     k = i ;
12:         for X_i such that ub(D(X_j)) < ub(D(X_i)) do
13:             if lb(D(X_k) ≤ lb(D(X_i)) then
14:                 D(X_i) = D(X_i) \ [inf, lb(D(X_k)) + 1] ;
```

Puget [Pug98] proposed a *BC* filtering algorithm for the ALL-DIFFERENT constraint that is also based on identification of Hall intervals. First, Puget observed that the problem can be simplified by achieving bounds consistency on the lower bounds independently of bounds consistency on the upper bounds. To achieve bounds consistency on the lower bounds (the upper bounds are similar), Puget sorts all variables by their upper bounds and processes variables in this order. For each pair of variables X_j and X_i, j is less than i in the order, the algorithm keeps tracks of the number of variables where domains are inside the interval $[lb(D(X_j)), ub(D(X_i))]$ and updates this information using dynamic programming. If the algorithm detects a Hall interval then these values are pruned according to Theorem 6.3. The naive implementation of the algorithm runs in $O(n^3)$ time, because we need to keep track of $O(n^2)$ intervals on each step. The complexity can be improved to $O(n \log n)$ per invocation by observing that we do not need to keep track of nested Hall intervals and by using a balanced binary tree to store the Hall intervals. Lopez-Ortiz *et al.* proposed an improvement of this algorithm that runs in $O(n)$ time per invocation [LOQTvB03].

Consider how to achieve range consistency and bounds consistency on our running example.

Example 6.2 *Recall that domains of the variables are:*

	1	2	3	4	5
X_1			$*$	$*$	
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

Bounds consistency. *We consider the interval* $[1, 1]$. *This interval is a Hall interval of size 1 that contains* $D(X_5)$. *Therefore, we prune value* 1 *from* $D(X_2)$.

	1	2	3	4	5
X_1			$*$	$*$	
X_2		$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

The constraint is now bounds consistent . At this point, there are two Hall intervals $[3, 4]$ *and* $[2, 5]$ *formed by domains of variables* X_1, X_3 *and* X_1, X_2, X_3, X_4, *respectively. However, none of the variables bounds, that are not contained inside these intervals overlap* $[3, 4]$ *or* $[2, 5]$. *Therefore, by Theorem 6.3, the constraint is bounds consistent .*

Range consistency. *The interval* $[3, 4]$ *is a Hall interval of size 2 as it completely contains the domains of 2 variables,* X_1 *and* X_3. *By Theorem 6.2, we can remove values* $[3, 4]$ *from the domains of* X_2 *and* X_4. *This leaves the following domains:*

	1	2	3	4	5
X_1			$*$	$*$	
X_2		$*$			$*$
X_3			$*$	$*$	
X_4		$*$			$*$
X_5	$*$				

By Theorem 6.2 the constraint is now range consistent . $\diamond$

6.3 A decomposition of the ALL-DIFFERENT constraint

In this section we propose a simple decomposition of the ALL-DIFFERENT constraint that allows enforcing range consistency. Following Theorems 6.2–6.3, the decomposition en-

sures that no interval can contain more variables than its size using a set of arithmetic constraints over a set of Boolean variables. We introduce $O(n^2d)$ new 0/1 variables A_{ilu} to represent whether X_i takes a value in the interval $[l, u]$. For $1 \leq i \leq n$, $1 \leq l \leq u \leq d$ and $u - l < n$, we post the following constraints:

$$A_{ilu} = 1 \iff X_i \in [l, u] \tag{6.1}$$

$$\sum_{i=1}^{n} A_{ilu} \leq u - l + 1 \tag{6.2}$$

Informally, constraints (6.1) provide channelling between variables A_{ilu} and the original variables X_i, and constraints (6.2) count how many original variables are inside each interval $[l, u]$.

If the number of original variables X whose domains are completely contained inside an interval $[l, u]$ becomes equal to the length of the interval, $u-l+1$, then constraints (6.2) force unset variables A_{ilu} to take value zero. In turn, setting a variable A_{ilu} to zero propagates pruning to the original variables X_i through the channelling constraints (6.1).

We will prove that enforcing *DC* on the decomposition enforces *RC* on the original ALL-DIFFERENT constraint. Interestingly, a simple decomposition like this can simulate a complex propagation algorithm like Leconte's. In addition, the overall complexity of reasoning with the decomposition is similar to Leconte's propagator.

Theorem 6.4 *Enforcing domain consistency on constraints* (6.1) *and* (6.2) *enforces range consistency on the corresponding* ALL-DIFFERENT *constraint in* $O(n^2d^2)$ *down any branch of the search tree.*

Proof: By Theorem 6.2 every Hall interval should be removed from the domain of variables whose domains are not fully contained within that Hall interval. Let $[a, b]$ be a Hall interval. That is, $|H| = b-a+1$ where $H = \{X_i \mid D(X_i) \subseteq [a, b]\}$. Constraint (6.1) fixes $A_{iab} = 1$ for all $D(X_i) \in H$. Constraint (6.2) with $l = a$ and $u = b$ becomes tight, fixing $A_{iab} = 0$ for all $X_i \notin H$. The channelling constraint (6.1) for $l = a$, $u = b$, and $X_i \notin H$ removes the interval $[a, b]$ from the domain of X_i as required for range consistency.

Complexity argument: There are $O(n^2d)$ constraints (6.1). Each constraint that can be invoked $O(d)$ times on variable X_i bound change and once on an assignment of the Boolean variable A_{ilu} down the branch of the search tree. In the first case each propagation requires $O(1)$ time. In the second case the single propagation requires $O(d)$ time. This gives $O(d)$ time down the branch of the search tree.

Constraints (6.1) therefore take $O(n^2d^2)$ down the branch of the search tree to propagate. There are $O(d^2)$ constraints (6.2) that each take $O(n)$ time to propagate down the

branch of the search tree for a total of $O(nd^2)$ time. The total running time is given by $O(n^2d^2) + O(nd^2) = O(n^2d^2)$. $\diamond$

We illustrate this decomposition on our running example.

Example 6.3 *Consider again the running example. Recall that domains of the variables are :*

	1	2	3	4	5
X_1			$*$	$*$	
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

Range consistency. *First take the interval* $[1, 1]$. *Since* $X_5 \in [1, 1]$, *constraint* (6.1) *implies* $A_{511} = 1$. *Now from the linear equation* (6.2) *we get that* $\sum_{i=1}^{5} A_{i11} \leq 1$. *That is, at most one variable can take the value* 1 *within this interval. This forces that* $A_{211} = A_{311} = A_{411} = A_{511} = 0$. *Using the channelling constraint* (6.1) *and* $A_{211} = 0$, *we get* $X_2 \notin [1, 1]$. *Since* $X_2 \in [1, 5]$, *this leaves* $X_2 \in [2, 5]$.

	1	2	3	4	5
X_1			$*$	$*$	
X_2		$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

Now take the interval $[3, 4]$. *From the channelling constraint* (6.1), *we obtain* $A_{134} = A_{334} = 1$. *Now from constraint* (6.2), $\sum_{i=1}^{5} A_{i34} \leq 2$. *That is, at most 2 variables can take a value within this interval. This means that* $A_{234} = A_{434} = A_{534} = 0$. *Using constraint* (6.1) *we get* $X_2 \notin [3, 4]$, $X_4 \notin [3, 4]$. *Since* $X_2 \in [2, 5]$ *and* $X_4 \in [2, 5]$, *this*

leaves $X_2 = \{2, 5\}$ *and* $X_4 \in \{2, 5\}$.

	1	2	3	4	5
X_1			$*$	$*$	
X_2		$*$			$*$
X_3			$*$	$*$	
X_4		$*$			$*$
X_5	$*$				

By Theorem 6.4 the ALL-DIFFERENT *constraint is range consistent .* $\diamond$

By using a decomposition that can only prune bounds of variable domains, we can give a decomposition that achieves bounds consistency in a similar way. In addition, we can reduce the overall complexity in the case that constraints are woken whenever their bounds change.

Following a linear encoding (Section 2.4.4), we introduce new 0/1 variables, B_{ik}, $1 \leq k \leq d$ and replace (6.1) by the following constraints, $1 \leq l \leq u \leq d$ and $u - l < n$,:

$$B_{il} = 1 \iff X_i \leq l \tag{6.3}$$

$$A_{ilu} = 1 \iff (B_{i(l-1)} = 0 \wedge B_{iu} = 1) \tag{6.4}$$

Theorem 6.5 *Enforcing bounds consistency on constraints* (6.2) *to* (6.4) *enforces bounds consistency on the corresponding* ALL-DIFFERENT *constraint in* $O(nd^2)$ *down any branch of the search tree.*

Proof: We first observe that bounds consistency is equivalent to domain consistency on constraints (6.2) because A_{ilu} are Boolean variables. So, the proof follows the proof for Theorem 6.4 except that fixing $A_{ilu} = 0$ prunes the bounds of $D(X_i)$ if and only if $B_{i(l-1)} = 0$ or $B_{iu} = 1$, that is, if and only if exactly one bound of the domain of X_i intersects the interval $[l, u]$. Only the bounds that do not have a bound support are shrunk.

Complexity argument: The complexity reduces compared to Theorem 6.4 as (6.3) appears $O(nd)$ times and is woken $O(d)$ times, whilst (6.4) appears $O(n^2d)$ times and is woken just $O(1)$ time. The total time complexity is $O(n^2d) + O(nd^2) = O(nd^2)$ $\diamond$

Example 6.4 *Consider again the running example. We start with initial domains:*

	1	2	3	4	5
X_1			$*$	$*$	
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

Bounds consistency. *Consider the interval* $[1, 1]$. *Since* $X_5 \in [1, 1]$,(6.3) *implies* $B_{50} = 0$ *and* $B_{51} = 1$. *The linear constraint* (6.4) *implies that* $A_{511} = 0$. *By the same reasoning as in Example 6.3, the value* 1 *is pruned for* X_2.

	1	2	3	4	5
X_1			$*$	$*$	
X_2		$*$	$*$	$*$	$*$
X_3			$*$	$*$	
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

If we consider the interval $[3, 4]$ *we can derive that* $A_{234} = A_{434} = A_{534} = 0$ *the same way as in Example 6.3. However, this does not give an additional pruning on the original variables* X. *Consider, for example, the second variable* X_2. *The variable* A_{234} *is fixed to 0. This implies that either* B_{24} *has to be* 0 *or* B_{22} *has to be* 1. *However, none of these Boolean variables is fixed and no propagation occurs on the variable* X_2. $\diamond$

A special case of ALL-DIFFERENT is PERMUTATION when we have the same number of values as variables, and the values are ordered consecutively. A decomposition of PERMUTATION just needs to replace (6.2) with the following equality where (as before) $1 \leq l \leq u \leq d$, and $u - l < n$:

$$\sum_{i=1}^{n} A_{ilu} \quad = \quad u - l + 1 \tag{6.5}$$

This can increase propagation. In some cases, *DC* on constraints (6.1) and (6.5) will prune values that a *RC* propagator for PERMUTATION would miss.

Example 6.5 *Consider a* PERMUTATION *constraint over the following variables and val-*

ues:

	1	2	3
X_1	$*$		$*$
X_2	$*$		$*$
X_3	$*$	$*$	$*$

These domains are range consistent . However, take the interval $[2, 2]$*. By enforcing DC on constraint* (6.1)*,* $A_{122} = A_{222} = 0$*. Now, from constraint* (6.5)*, we have* $\sum_{i=1}^{3} A_{i22} = 1$*. Thus* $A_{322} = 1$*. By constraint* (6.1)*, this sets* $X_3 = 2$*. On this particular problem instance, DC on constraints* (6.1) *and* (6.5) *has enforced domain consistency on the original* ALL-DIFFERENT *constraint.* $\diamond$

6.4 Other decompositions of the ALL-DIFFERENT constraint

In this section we consider two other decompositions of the ALL-DIFFERENT constraint. The first decomposition is an existing naive decomposition into a clique of binary inequality constraints [Lau78]. This decomposition does not enforce bounds consistency but it can be also easily implemented in a constraint solver. We present it for completeness of the material. We also use this model in an experimental evaluation. The second decomposition is a new alternative decomposition of the ALL-DIFFERENT constraint to the decomposition that is described in Section 6.3. This decomposition does not achieve bounds consistency, but it detects bounds disentailment and provides an interesting connection to the GEN-SEQUENCE constraint (Section 4.2.4) . Moreover, this decomposition is useful to construct a bounds consistency filtering algorithm for the OVERLAPPINGALLDIFF constraint .

6.4.1 Decomposition into clique of binary inequalities

A natural way to decompose the ALL-DIFFERENT constraint is to encode the constraint using a set of binary inequalities.

$$X_i \neq X_j \quad i, j = 1, \ldots, n; i \neq j \tag{6.6}$$

Clearly, constraints (6.6) are logically equivalent to the ALL-DIFFERENT constraint. However, this decomposition provides only weak inference as the following theorem shows:

Theorem 6.6 *Enforcing domain consistency on the decomposition* (6.6) *does not detect bounds disentailment of the* ALL-DIFFERENT *constraint.*

Proof: Consider an ALL-DIFFERENT$([X_1, X_2, X_3])$ with the following domains $D(X_1) = D(X_2) = D(X_3) = \{1, 2\}$. Enforcing domain consistency on each binary constraint does not cause any pruning. However, the ALL-DIFFERENT constraint is bounds disentailed. $\diamond$

6.4.2 Decomposition using the linear encoding

Gent and Nightingale [GN04] proposed a decomposition of the ALL-DIFFERENT constraint that is based on ladder encoding that they proposed in the same paper. We call this the linear encoding to be consistent with the rest of the thesis.

Exactly one encoding(EO) Consider a set of S Boolean variables. We want to encode that exactly one Boolean variable in the set S is true. We introduce $p = n - 1$ additional Boolean variables and a set of clauses. The first set of clauses are validity clauses:

$$\bigwedge_{i=1}^{p-1} (\bar{y}_{i+1} \vee y_i) \tag{6.7}$$

$$\tag{6.8}$$

The second set of constraints contains channeling constraints:

$$\bigwedge_{i=1}^{|S|} [(y_{i-1} \wedge \bar{y}_i) \iff x_i] \tag{6.9}$$

Finally, we set $y_0 = 1$ and $y_{|S|} = 0$.

At most one encoding(AMO) We add an extra variable to the EO encoding to obtain the at most one encoding (AMO). This variable indicates that all variables from S are set to false.

To use the linear encoding to encode the ALL-DIFFERENT constraint we consider direct encoding Boolean variables. We use EO encoding to make sure that a variable takes a value. We use AMO encoding to ensure that each value is taken by at most one variable. It was shown in [GN04] that the proposed encoding is as strong as the decomposition into a set of binary inequalities.

Theorem 6.7 *Unit propagation on the decomposition* (6.7)–(6.9) *does not detect bounds disentailment for the* ALL-DIFFERENT *constraint.*

Proof: Consider ALL-DIFFERENT$([X_1, X_2, X_3, X_4, X_5])$ with the following domains $D(X_1) = D(X_2) = D(X_3) = \{1, 4\}$ and $D(X_4) = D(X_5) = \{2, 3\}$. We introduce

Table 6.1: Boolean variables to encode $X_1, \ldots, X_5$.

	X_1	X_2	X_3	X_4	X_5
1	x_1^1	x_1^2	x_1^3	$x_1^4 = 0$	$x_1^5 = 0$
2	$x_2^1 = 0$	$x_2^2 = 0$	$x_2^3 = 0$	x_2^4	x_2^5
3	$x_3^1 = 0$	$x_3^2 = 0$	$x_3^3 = 0$	x_3^4	x_3^5
4	x_4^1	x_4^2	x_4^3	$x_4^4 = 0$	$x_4^5 = 0$

20 Boolean variables x_v^i, $v = 1, \ldots, 4$, $i = 1, \ldots, 5$ (Table 6.1). The variable x_v^i is set to true iff the variable X_i is assigned to v.

Consider the EO constraints for the 1st column (the second and the third are symmetric). We have the following validity constraints: $\bar{y}_1^1 \vee y_0^1$, $\bar{y}_2^1 \vee y_1^1$, $\bar{y}_3^1 \vee y_2^1$ and $\bar{y}_4^1 \vee y_3^1$. These can be simplified to $\bar{y}_2^1 \vee y_1^1$ and $\bar{y}_3^1 \vee y_2^1$.

We have the following channeling constraints: $\left(y_0^1 \wedge \bar{y}_1^1\right) \iff x_1^1$, $\left(y_1^1 \wedge \bar{y}_2^1\right) \iff x_2^1$, $\left(y_2^1 \wedge \bar{y}_3^1\right) \iff x_3^1$ and $\left(y_3^1 \wedge \bar{y}_4^1\right) \iff x_4^1$.

These constraints can be simplified to $\left(\bar{y}_1^1\right) \iff x_1^1$, $\left(y_1^1 \wedge \bar{y}_2\right) \iff$ FALSE, $\left(y_2^1 \wedge \bar{y}_3\right) \iff$ FALSE and $\left(y_3^1\right) \iff x_4^1$.

Unit propagation cannot make further inference.

Consider the EO constraints for the 4st column (the fifth column is symmetric.) We have the following validity constraints: $\bar{y}_1^4 \vee y_0^4$, $\bar{y}_2^4 \vee y_1^4$, $\bar{y}_3^4 \vee y_2^4$ and $\bar{y}_4^4 \vee y_3^4$. These can be simplified to $\bar{y}_2^4 \vee y_1^4$ and $\bar{y}_3^4 \vee y_2^4$.

We have the following channeling constraints: $\left(y_0^4 \wedge \bar{y}_1^4\right) \iff x_1^4$, $\left(y_1^4 \wedge \bar{y}_2^4\right) \iff x_2^4$, $\left(y_2^4 \wedge \bar{y}_3^4\right) \iff x_3^4$ and $\left(y_3^4 \wedge \bar{y}_4^4\right) \iff x_4^4$.

These constraints can be simplified to $\left(\bar{y}_1^4\right) \iff$ FALSE, $\left(y_1^4 \wedge \bar{y}_2^4\right) \iff x_2^4$, $\left(y_2^4 \wedge \bar{y}_3^4\right) \iff x_2^4$ and $\left(y_3^4\right) \iff$ FALSE.

Unit propagation derives: $y_1^4 = true$ and $y_3^4 = false$. All validity constraints are satisfied.

AMO constraints for any row cannot derive any inference as none of the variables is set to a value. Hence, the unit propagation cannot make further inference while the constraint is bounds disentailed. $\diamond$

6.4.3 Partial sums decomposition of the ALL-DIFFERENT constraint.

The next decomposition that we consider is a decomposition into partial sums that is based on encoding into the GEN-SEQUENCE constraint (Section 4.2.4). This decomposition can be seen as complementary to the *BC* decomposition (Section 6.3) because it provides an

alternative view on the ALL-DIFFERENT constraint. We also use this decomposition to build the first polynomial time propagator for the overlapping ALL-DIFFERENT constraint (Section 6.7).

Reduction to the GEN-SEQUENCE constraint. First we show that detecting bounds disentailment of the ALL-DIFFERENT constraint can be polynomially reduced to detecting bounds disentailment of the GEN-SEQUENCE constraint. This reduction reveals an interesting connection between two classes of constraints: counting constraints, like the ALL-DIFFERENT constraint, and sliding constraints, like the GEN-SEQUENCE constraint. We find this connection interesting and show that it is useful to construct a polynomial time propagator for the overlapping ALL-DIFFERENT constraint.

We recall that to fulfil conditions of Theorems 6.1–6.3, we count how many **variables** are completely inside each interval of values (Section 6.3). In our new decomposition we show that it is enough to keep track of how many **values** are taken by variables X in each interval of values. Informally, we say that if we can find a set of n values so that the number of selected values in each interval is between the number of variables whose domains are inside this interval and the length of this interval, then we can construct a solution of the ALL-DIFFERENT constraint. To encode this observation we use the GEN-SEQUENCE constraint. We introduce Boolean variables a_v, $v = 1, \ldots, d$ which indicate whether the value v was taken by one of the variables X_i:

$$a_v = 1 \Leftrightarrow |\exists i.\, X_i = v| > 0 \quad v = 1, \ldots, d \tag{6.10}$$

Then the ALL-DIFFERENT constraint can be encoded as GEN-SEQUENCE over the new variables a_v in the following way. For each interval of values $[l, u]$, where l is a lower bound of one of variables X and u is an upper bound of one of variables X, we introduce an $\text{AMONG}_{lu}([a_l, \ldots, a_u], b_{lu}, c_{lu})$ constraint, where $b_{lu} = |\{X_i | D(X_i) \subseteq [l, u]\}|$, $c_{lu} = u - l + 1$, to ensure that Hall's condition is satisfied (Theorem 6.1). Note that the value b_{lu} is constructed from the domains of variables X to pass information between the ALL-DIFFERENT and the $\text{GEN-SEQUENCE}_{alldiff}$ constraints. We also introduce the $\text{AMONG}([a_1, \ldots, a_d], n, n)$ constraint to make sure that exactly n values will be used. Then the GEN-SEQUENCE constraint is a conjunction of these AMONG constraints:

Definition 6.7

$$\text{GEN-SEQUENCE}_{alldiff} \Leftrightarrow \text{AMONG}([a_1, \ldots, a_d], n, n) \bigwedge \left(\bigwedge_{l \leq u} \text{AMONG}_{lu}\left([a_l, \ldots, a_u], b_{lu}, c_{lu}\right) \right),$$

where $b_{lu} = |\{X_i | D(X_i) \subseteq [l,u]\}|$, $c_{lu} = u - l + 1$, $l \in LB$ *and* $u \in UB$.

Note that the upper bound parameter c_{lu} is redundant for each AMONG_{lu} constraint but we keep it for consistency with our results for the OVERLAPPINGALLDIFF constraint.

Example 6.6 *Consider the following example with 5 variables:*

	1	2	3	4	5	6
X_1			$*$	$*$		
X_2	$*$	$*$	$*$	$*$	$*$	$*$
X_3			$*$	$*$		
X_4		$*$	$*$	$*$	$*$	$*$
X_5	$*$					

We construct the $\text{GEN-SEQUENCE}_{alldiff}$ *constraint according to Definition 6.7. We introduce six Boolean variables* $a_1, \ldots, a_6$. *The set of lower bounds is* $\{1,2,3\}$ *and the set of upper bounds is* $\{1,4,6\}$. *Therefore, we introduce constraints*

$$\text{AMONG}_{16}([a_1,\ldots,a_6],5,5) \qquad \text{AMONG}_{11}([a_1],1,1)$$
$$\text{AMONG}_{14}([a_1,a_2,a_3,a_4],3,4) \qquad \text{AMONG}_{16}([a_1,\ldots,a_6],5,6)$$
$$\text{AMONG}_{24}([a_2,a_3,a_4],2,3) \qquad \text{AMONG}_{26}([a_2,\ldots,a_6],3,5)$$
$$\text{AMONG}_{34}([a_3,a_4],2,2) \qquad \text{AMONG}_{36}([a_3,\ldots,a_6],2,4)$$

Note that some constraints are redundant.

Theorem 6.8 *The* $\text{GEN-SEQUENCE}_{alldiff}$ *constraint as defined by Definition 6.7 is satisfiable if and only if the* ALL-DIFFERENT *constraint is satisfiable.*

Proof: Any solution of the ALL-DIFFERENT constraint is a solution of the $\text{GEN-SEQUENCE}_{alldiff}$ constraint, because Theorem 6.1 holds for this solution. Therefore, we only have to show the converse.

Shrinking domains. Consider a solution of the $\text{GEN-SEQUENCE}_{alldiff}$ constraint $[a_1,\ldots,a_d]$. The first $\text{AMONG}([a_1,\ldots,a_d],n,n)$ constraint ensures that exactly n Boolean variables a_v are set to true. We shrink domains of variables X in the following way. If $a_v = 0$ we remove the value v from domains of all variables X. This leaves n values in the universe of possible values $\cup_{i=1}^{n} D(X_i)$. We renumber these values consecutively from 1 to n and obtain new domains of variables $D'(X_i)$, $i = 1,\ldots,n$. We renumber the values to point out that new domains D' are intervals, as the original domains are intervals. This will allow us to work with Hall intervals rather than Hall sets in the remaining proof.

Consider the shrinking procedure on Example 6.6. Consider a solution over variables a, $a_1 = 1, a_2 = 0, a_3 = 1, a_4 = 1, a_5 = 1$ and $a_6 = 1$. As $a_2 = 0$ then we remove value $\{2\}$ from domains of $X_1, \ldots, X_5$ and renumber 3 to 2, 4 to 3, 5 to 4 and 6 to 5. We get the following domains D':

	1	2	3	4	5
X_1		$*$	$*$		
X_2	$*$	$*$	$*$	$*$	$*$
X_3		$*$	$*$		
X_4		$*$	$*$	$*$	$*$
X_5	$*$				

Solution of ALL-DIFFERENT**.** We prove that ALL-DIFFERENT$([D'(X_1), \ldots, D'(X_n)])$ has a solution. Suppose there is no solution of the ALL-DIFFERENT constraint. Then there is an interval $[e, f]$ and a set of variables $P = \{X_i | D'(X_i) \subseteq [e, f]\}$ such that $|P| > f - e + 1$. Consider the tightest interval, $[l, u]$ that contains all variables in P before shrinking the intervals. The AMONG$_{lu}([a_l, \ldots, a_u], b_{lu}, u - l + 1)$ constraint, where $b_{lu} = |\{X_i | D(X_i) \subseteq [l, u]\}|$ and $c_{lu} = u - l + 1$, is satisfied, hence $|\{X_i | D(X_i) \subseteq [l, u]\}| \leq \sum_{i=l}^{u} a_i$. Consider all values V that were pruned from the interval $[l, u]$. As the value v, $v \in V$ was pruned, the corresponding variable $a_v, v \in V$, is set to 0. Therefore, after shrinking, we have

$$|\{X_i | D'(X_i) \subseteq [e, f]\}| = |\{X_i | D(X_i) \subseteq [l, u]\}| \leq \sum_{i=l}^{u} a_i = \sum_{i=l; i \notin V}^{u} a_i.$$

The interval $[e, f]$ contains all renamed values from the original interval $[l, u]$ such that $v \notin V$, hence $\sum_{i=l; i \notin V}^{u} a_i = f - e + 1$. This gives

$$|\{X_i | D'(X_i) \subseteq [e, f]\}| \leq \sum_{i=l}^{u} a_i = f - e + 1.$$

This contradicts our assumption that $|\{X_i | D'(X_i) \subseteq [e, f]\}| > f - e + 1$. $\diamond$

Theorem 6.7 provides an interesting connection between the ALL-DIFFERENT constraint and the GEN-SEQUENCE constraint. However, the encoding into the GEN-SEQUENCE constraint is not enough to enforce bounds consistency on the ALL-DIFFERENT constraint.

Theorem 6.9 *Enforcing DC on the* GEN-SEQUENCE$_{alldiff}$ *constraint specified by Definition* (6.7) *and constraints* (6.10) *does not enforce bounds consistency on the* ALL-DIFFERENT *constraint.*

Proof: Consider the ALL-DIFFERENT$([X_1, X_2, X_3])$ constraint with $D(X_1) = D(X_2) = \{1, 2\}$ and $D(X_3) = \{1, 2, 3\}$. We introduce 3 Boolean variables a_1, a_2 and a_3 and 3 AMONG constraints according Definition 6.7: and AMONG$_{13}([a_1, a_2, a_3], 3, 3)$, AMONG$_{12}([a_1, a_2], 2, 2)$ and AMONG$_{13}([a_1, a_2, a_3], 3, 3)$. The last constraint is redundant. The only solution of the GEN-SEQUENCE constraint is $a_1 = a_2 = a_3 = 1$. Constraints (6.10) are domain consistent . Hence, the values 1 and 2 are not pruned from $D(X_3)$. $\diamond$

Decomposition into the partial sums. We recall that to obtain the partial sums encoding of the GEN-SEQUENCE constraint, we introduce integer variables P_{lu}, $P_{lu} = \sum_{v=l}^{u} a_v$ each with domain $[0, n]$ (Section 4.3.4). We connect P_{lu} variables through a set of ternary constraints: $P_{lu} = P_{lk} + P_{(k+1)u}$, $1 \leq l \leq k < u \leq d$ and $u - l < n$. To satisfy the AMONG$_{lu}([a_l, \ldots, a_u], b_{lu}, c_{lu})$ constraint, we need to make sure that $b_{lu} \leq P_{lu} \leq c_{lu}$ The upper bound parameter is a constant and is equal to $u - l + 1$, so we can express this restriction with a constraint $P_{lu} \leq u - l + 1$. The lower bound parameter $b_{lu} = |\{X_i | D(X_i) \subseteq [l, u]\}|$ varies during the search and can be updated using the *BC* decomposition of the ALL-DIFFERENT constraint. This decomposition introduces variables A_{ilu} and constraints (6.3)–(6.4). Hence, we can easily update the lower bound of the P_{lu} variables, because $b_{lu} = \sum_{i=1}^{n} lb(A_{ilu}) = |\{X_i | D(X_i) \subseteq [l, u]\}|$. Then a combination of the PS decomposition of the GEN-SEQUENCE constraint and the *BC* composition of the ALL-DIFFERENT constraint is the following:

$$B_{il} = 1 \iff X_i \leq l \tag{6.11}$$

$$A_{ilu} = 1 \iff (B_{i(l-1)} = 0 \wedge B_{iu} = 1) \tag{6.12}$$

$$P_{lu} = \sum_{i=1}^{n} A_{ilu} \tag{6.13}$$

$$P_{lu} \leq u - l + 1 \tag{6.14}$$

$$P_{lu} = P_{lk} + P_{(k+1)u}, \tag{6.15}$$

where $1 \leq i \leq n$, $1 \leq l \leq k < u \leq d$ and $u - l < n$.

The decomposition (6.11)–(6.15) does enforce bounds consistency on the ALL-DIFFERENT constraint as it subsumes the decomposition (6.3)–(6.4). Note that the decomposition (6.11)–(6.15) encapsulates the AMONG constraints over the all possible intervals of values, $[l, u]$, $l \in LB$ and $u \in UB$ while it is sufficient to consider only $O(n)$ AMONG constraints at any point of the search. Since, domains of the variables are pruned and their lower and upper bounds are changed, we might need to consider any $O(n^2)$ of $O(d^2)$ intervals. As we do not know in advance which $O(n^2)$ intervals to

consider, we have to keep track of all $O(d^2)$ intervals with decomposition.

6.5 Experimental Results

To test these decompositions, we ran experiments on pseudo-Boolean encodings (PB) of *CSP*s containing ALL-DIFFERENT and PERMUTATION constraints. Using SAT solvers we can evaluate whether dynamic branching heuristic and learning schemes can take advantage of the auxiliary variables in the decompositions. This is not possible within publicity available CSP solvers. We used the MiniSat+ 1.13 solver on an Intel Xeon 4 CPU, 2.0 Ghz, 4G RAM machine with a timeout of 600 seconds for each experiment. Our decompositions contain two types of constraints: SUM constraints like (6.2) and MEMBER constraints like (6.1). The SUM constraints can be posted directly to the MiniSat+ solver. To encode MEMBER constraints in the pseudo-Boolean form, we use literals B_{ij} for the truth of $X_i \leq j$ (Section 2.4.4), and clauses of the form $(A_{ilu} = 1) \Leftrightarrow (B_{i(l-1)} = 0 \wedge B_{iu} = 1)$. This encoding achieves bounds consistency as stated in Theorem 6.5. To increase propagation, we use a direct encoding with literals Z_{ij} for the truth of $X_i = j$ and clauses $(A_{ilu} = 0) \Rightarrow (Z_{ij} = 0)$, $j \in [l, u]$. The overall level of consistency achieved is therefore between bounds consistency and range consistency. We denote this encoding HI. To explore the impact of small Hall intervals, we also tried HI_k, a PB encoding of our decomposition with only the subset of constraints (6.2) for which $u - l + 1 \leq k$. This detects Hall intervals of size at most k. Finally, we also decomposed the ALL-DIFFERENT constraint into a clique of binary inequalities, and used a direct encoding to convert this into *SAT*. We denote it by BI.

Pigeon Hole Problems. Table 6.2 gives results on pigeon hole problems (PHP) with n pigeons and $n - 1$ holes. Our decomposition is both faster and gives a smaller search tree compared to the BI decomposition. On such problems, detecting large Hall intervals is essential.

Double-Wheel Graceful Graphs. The second set of experiments uses double-wheel graceful graphs [PS03]. We converted the *CSP* model given in [PS03] into a PB formula. The model contains an ALL-DIFFERENT constraint on node labels and a PERMUTATION constraint on edge labels. For the PERMUTATION constraint we use (6.5). We strengthen the BI decomposition in this case with clauses to ensure that every value appears at least once. Table 6.3 show that our decomposition outperforms the augmented BI decomposi-

Table 6.2: PHP problems. t is time and bt is the number of backtracks to solve the problem.

n	BI $bt/\ t$	HI_1 $bt/\ t$	HI_3 $bt/\ t$	HI_5 $bt/\ t$	HI_7 $bt/\ t$	HI_9 $bt/\ t$
5	30/ **0.0**	28/ **0.0**	**4**/ **0.0**			
7	622/ **0.0**	539/ **0.0**	47/ **0.0**	**6**/ **0.0**		
9	16735/ 0.3	18455/ 0.7	522/ **0.0**	122/ **0.0**	**8**/ **0.0**	
11	998927/ 29.3	665586/ 44.8	5681/ 0.3	171/ **0.0**	180/ **0.0**	**10**/ 0.1
13	-/ -	-/ -	13876/ 0.9	2568/ 0.2	247/ **0.1**	**195**/ **0.1**
15	-/ -	-/ -	1744765/ 188.6	24109/ 2.6	1054/ 0.2	**165**/ **0.1**
17	-/ -	-/ -	-/ -	293762/ 48.0	8989/ 1.1	**4219**/ **0.6**
19	-/ -	-/ -	-/ -	107780/ 21.8	857175/ 368.0	**39713**/ **9.9**
21	-/ -	-/ -	-/ -	-/ -	550312/ 426.2	**57817**/ **33.5**

tion on many instances. Whilst detecting large Hall intervals can reduce the search space dramatically, in some cases MiniSat+'s branching heuristics and clause learning features appear to be fooled by the many extra variables introduced in the encodings.

Overall these experiments suggest that detecting Hall intervals reduces search significantly, and focusing on small Hall intervals may be best except on problems where large Hall intervals occur frequently.

Table 6.3: Double-wheel graceful graphs. t is time and bt is the number of backtracks to solve the problem

DW_n	BI $bt/\ t$	HI_1 $bt/\ t$	HI_3 $bt/\ t$	HI_5 $bt/\ t$	HI_7 $bt/\ t$	HI_9 $bt/\ t$
3	176/ 0.1	90/ 0.1	**63**/ **0.1**			
4	30/ **0.1**	**14**/ 0.1	212/ 0.2			
5	**22**/ **0.2**	526/ 0.4	87/ 0.3	1290/ 1.7		
6	1341/ 1.0	873/ 0.9	**318**/ **0.7**	1212/ 2.9		
7	2948/ 3.6	2047/ 4.2	1710/ 3.6	1574/ 4.0	**27**/ **0.9**	
8	2418/ 5.5	724/ **2.2**	643/ 2.8	**368**/ 2.4	3955/ 19.5	
9	3378/ 8.6	1666/ 5.7	1616/ 9.0	**30**/ **1.8**	10123/ 129.7	405/ 6.5
10	19372/ 118.3	9355/ 66.2	14120/ 85.9	**10**/ **2.1**	4051/ 35.0	5709/ 71.2
11	839/ 5.4	12356/ 84.2	1556/ 13.9	**14**/ **2.4**	7456/ 105.2	5552/ 92.7

6.6 Generalisations of ALL-DIFFERENT

In the following sections we consider the three useful generalisations of the ALL-DIFFERENT constraint. We show that bounds consistency and range consistency filtering algorithms for these constraints can be decomposed in a way similar to decompo-

sitions of the ALL-DIFFERENT constraint. In Section 6.7 we consider the overlapping ALL-DIFFERENT constraint. We propose a decomposition that detects bounds disentailment for the constraint that it based on a connection between the ALL-DIFFERENT constraint and the GEN-SEQUENCE constraint. constraints. In Section 6.8–6.9 we focus on range consistency and bounds consistency decompositions of the GCC constraint and NVALUE constraint, respectively.

6.7 The overlapping ALL-DIFFERENT constraint

In this section we consider the OVERLAPPINGALLDIFF constraint and show that enforcing bounds consistency on this constraint is polynomial. We recall that the OVERLAPPINGALLDIFF$([X_1, \ldots, X_n], S, T)$ constraint where $S \subseteq X$, $T \subseteq X$, $S \cup T = X$ holds if and only if ALL-DIFFERENT(S) and ALL-DIFFERENT(T) hold simultaneously. This constraint is more difficult to propagate compared to ALL-DIFFERENT because enforcing domain consistency on this constraint is NP-hard [KEKM08].

Let us introduce a running example:

Example 6.7 *We have 7 exams and 2 students that each have to take 5 of the 7 exams over a 5 day period. We number the exams from* 1 *to* 7*. The first student has to take the first 5 exams and the second student has to take the last 5 exams. Due to the availability of examiners, not every exam is offered each day. For example, the* 1^{st} *exam is not offered on the final day of the week. Only one exam can be sat each day by each student. This problem can be encoded as an* OVERLAPPINGALLDIFF *constraint. We introduce seven integer variables to represent exams,* X_1 *to* X_7*. As the first student has to take the first five exams, the set* S *includes variables* $\{X_1, X_2, X_3, X_4, X_5\}$*. As the second student has to take the last five exams, the set* T *includes variables* $\{X_3, X_4, X_5, X_6, X_7\}$*. The following matrix shows domains of all variables that take into account availability restrictions:*

		1	2	3	4	5
$S \setminus T$	X_1	$*$	$*$	$*$	$*$	
	X_2	$*$	$*$			
$S \cap T$	X_3	$*$	$*$	$*$		
	X_4	$*$	$*$	$*$		
	X_5	$*$	$*$	$*$	$*$	$*$
$T \setminus S$	X_6		$*$	$*$		
	X_7		$*$	$*$	$*$	$*$

Finding a solution for this OVERLAPPINGALLDIFF *constraint is equivalent to solving the timetabling problem.* $\diamond$

We denote as P the subset of variables $\{X_1, \ldots, X_n\}$, $P^S = P \cap (S \setminus T)$, $P^T = P \cap (T \setminus S)$ and $P^{ST} = P \cap S \cap T$. Consider the set of variables $P = \{X_1, X_4, X_5, X_7\}$ from the running example 6.7. Then, $P^S = \{X_1\}$, $P^T = \{X_7\}$ and $P^{ST} = \{X_4, X_5\}$.

6.7.1 Decomposition of the OVERLAPPINGALLDIFF constraint

Similar to Section 6.4.3, we show that detecting bounds disentailment of the ALL-DIFFERENT constraint can be polynomially reduced to detecting bounds disentailment of the GEN-SEQUENCE constraint . In the case of OVERLAPPINGALLDIFF this reduction is more sophisticated compared to the ALL-DIFFERENT constraint. Based on this reduction, we construct a decomposition of the OVERLAPPINGALLDIFF constraint.

This section also corrects a mistake in the paper [BKN$^+$10]. First we recall Theorem 2 that we rewrite using the notation of this thesis:

Theorem 6.10 (Simultaneous Hall Condition [BKN$^+$10])
Let OVERLAPPINGALLDIFF$([X_1, \ldots, X_n], S, T)$ *be the overlapping* ALL-DIFFERENT *constraint. There exists a solution of the constraint if and only if* $|\mathbf{D}(P)| + |\mathbf{D}(P^S) \cap \mathbf{D}(P^T)| \geq |P|$ *for* $P \subseteq \{X_1, \ldots, X_n\}$, *where* $\mathbf{D}(P) = \cup_{X_i \in P} D(X_i)$.

The following example shows that this condition is insufficient.

Example 6.8

		1	2	3	4	5
$S \setminus T$	X_1		$*$	$*$	$*$	
$S \cap T$	X_2		$*$	$*$		
	X_3		$*$	$*$		
	X_4	$*$	$*$	$*$	$*$	$*$
	X_5	$*$	$*$	$*$	$*$	$*$
$T \setminus S$	X_6	$*$	$*$	$*$		

Consider, for example, $P = \{X_1, \ldots, X_6\}$ *and compute the condition for this set.* $\mathbf{D}(P) = 5$ *and* $|\mathbf{D}(P^S) \cap \mathbf{D}(P^T)| = 2$. *Hence,* $7 = |\mathbf{D}(P)| + |\mathbf{D}(P^S) \cap \mathbf{D}(P^T)| \geq |P| = 6$ *Consider another example,* $P = \{X_1, X_2, X_3, X_6\}$. $\mathbf{D}(P) = 4$, $|\mathbf{D}(P^S) \cap \mathbf{D}(P^T)| = 2$ *and* $6 = |\mathbf{D}(P)| + |\mathbf{D}(P^S) \cap \mathbf{D}(P^T)| \geq |P| = 4$. *It is easy to see that the condition holds for any set of variables. However, the constraint does not have a solution as values* $\{2, 3\}$

are taken by X_2 and X_3. Hence, X_1 and X_6 cannot share any value. Note that the simple decomposition into 2 ALL-DIFFERENT *constraints will detect disentailment.*

The proof makes the following false claim (the last paragraph): "If $u \in A^{S'}$ ($u \in A^{T'}$ is similar) then v can be in $N(P^{T*})$, so v is a shared vertex with $N(P^*)$. There are two cases to consider:".

There can be three cases here. $\diamond$

Unfortunately, Theorem 2 was used to prove the main result of the paper that bounds consistency on the OVERLAPPINGALLDIFF constraint is polynomial. However, the result holds and we give an alternative proof here.

Reduction to the GEN-SEQUENCE **constraint.** Our reduction is based on a partitioning of the OVERLAPPINGALLDIFF constraint into 3 ALL-DIFFERENT constraints over overlapping sets of variables — ALL-DIFFERENT$([S \setminus T])$, ALL-DIFFERENT$([S \cap T])$ and ALL-DIFFERENT$([T \setminus S])$. We show that it is sufficient to find a solution of a single ALL-DIFFERENT over the shared variables between S and T that is subject to extra restrictions. We want a solution of the 'middle' ALL-DIFFERENT$([S \cap T])$ constraint that leaves enough values in each interval to satisfy the other two 'side' ALL-DIFFERENT constraints. More precisely, if we have a solution of ALL-DIFFERENT$([S \cap T])$ that *does not* use at least $\max(|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l,u]\}|, |\{X_i \mid X_i \in T \setminus S, D(X_i) \subseteq [l,u]\}|)$ values in each interval $[l,u]$, then we can always extend this solution to a solution of the OVERLAPPINGALLDIFF constraint.

We show that we can reformulate ALL-DIFFERENT$([S \cap T])$ using the GEN-SEQUENCE that takes into account the information about the number of values that have to be unused in each interval of values. We introduce Boolean variables a_v, $v = 1, \ldots, d$ that indicate whether there exists a variable $X_i \in S \cap T$ that equals v.

$$a_v = 1 \Leftrightarrow |\exists i.\, X_i = v, X_i \in S \cap T| > 0 \quad v = 1, \ldots, d \tag{6.16}$$

For each interval of values $[l,u]$, where l is a lower bound of one of variables X and u is an upper bound of one of variables X, we introduce an AMONG$_{lu}([a_l, \ldots, a_u], b_{lu}, c_{lu})$ constraint, where $b_{lu} = |\{X_i | X_i \in S \cap T, D(X_i) \subseteq [l,u]\}|$ and $c_{lu} = u - l + 1 - \max(|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l,u]\}|, |\{X_i \mid X_i \in T \setminus S, D(X_i) \subseteq [l,u]\}|)$. The parameter b_{lu} ensures that Hall's condition is satisfied for the ALL-DIFFERENT$([S \cap T])$ constraint (Theorem 6.1). The parameter c_{lu} ensures that Hall's condition is satisfied for ALL-DIFFERENT$([S \setminus T])$ and ALL-DIFFERENT$([T \setminus S])$. In addition, we introduce

AMONG$([a_1, \ldots, a_d], |S \cap T|, |S \cap T|)$ to make sure that exactly $|S \cap T|$ values will be used. The GEN-SEQUENCE constraint is a conjunction of these AMONG constraints:

Definition 6.8

$$\text{GEN-SEQUENCE}_{oalldiff} \Leftrightarrow \text{AMONG}([a_1, \ldots, a_d], |S \cap T|, |S \cap T|) \bigwedge$$

$$\left(\bigwedge_{l \leq u} \text{AMONG}_{lu}\left([a_l, \ldots, a_u], b_{lu}, c_{lu}\right) \right),$$

where $b_{lu} = |\{X_i | X_i \in S \cap T, D(X_i) \subseteq [l,u]\}|$, $c_{lu} = u - l + 1 - \max(|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l,u]\}|, |\{X_i \mid X_i \in T \setminus S, D(X_i) \subseteq [l,u]\}|)$, $l \in LB$ *and* $u \in UB$.

Example 6.9 *Consider our running example.*

		1	2	3	4	5
$S \setminus T$	X_1	$*$	$*$	$*$	$*$	
	X_2	$*$	$*$			
$S \cap T$	X_3	$*$	$*$	$*$		
	X_4	$*$	$*$	$*$		
	X_5	$*$	$*$	$*$	$*$	$*$
$T \setminus S$	X_6		$*$	$*$		
	X_7		$*$	$*$	$*$	$*$

We construct the GEN-SEQUENCE$_{oalldiff}$ *constraint according to Definition 6.8. We introduce five Boolean variables* $a_1, \ldots, a_5$. *The set of lower bounds is* $\{1,2\}$ *and the set of upper bounds is* $\{2,3,4,5\}$. *Therefore, we introduce constraints*

$$\text{AMONG}_{15}([a_1, \ldots, a_5], 3, 3) \qquad \text{AMONG}_{12}([a_1, a_2], 0, 2 - \max(1,0))$$
$$\text{AMONG}_{13}([a_1, a_2, a_3], 2, 3 - \max(1,1)) \qquad \text{AMONG}_{14}([a_1, \ldots, a_4], 2, 4 - \max(2,1))$$
$$\text{AMONG}_{15}([a_1, \ldots, a_5], 3, 5 - \max(2,2)) \qquad \text{AMONG}_{22}([a_2], 0, 1 - \max(0,0))$$
$$\text{AMONG}_{23}([a_2, a_3], 0, 2 - \max(0,1)) \qquad \text{AMONG}_{24}([a_2, a_3, a_4], 0, 3 - \max(0,1))$$
$$\text{AMONG}_{25}([a_2, \ldots, a_5], 0, 4 - \max(0,2))$$

Theorem 6.11 *The* GEN-SEQUENCE$_{oalldiff}$ *constraint as defined by Definition* (6.8) *is satisfiable if and only if the* OVERLAPPINGALLDIFF *constraint is satisfiable.*

Proof: Consider a solution of the overlapping ALL-DIFFERENT constraints over Boolean variables a_v. Since every interval satisfies the Hall condition for the variables in $S \cap T$ then the number of a_v that are set to one is greater or equal to $|\{X_i | X_i \in S \cap T, D(X_i) \subseteq$

$[l, u]\}|$. On the other hand the Hall condition holds for ALL-DIFFERENT$([S \setminus T])$ and ALL-DIFFERENT$([T \setminus S])$. Hence, the number of value that are taken by variables in $(S \setminus T) \cup (T \setminus S)$ is greater than or equal to $\max(|\{X_i \mid i \in S \setminus T, D(X_i) \subseteq [l, u]\}|, |\{X_i \mid i \in T \setminus S, D(X_i) \subseteq [l, u]\}|)$. Therefore, the upper bound constraint is satisfied too for each AMONG constraint.

Conversely, consider a solution of the GEN-SEQUENCE$_{oalldiff}$ constraint.

Shrinking domains. First, we shrink domains of variables to construct three independent ALL-DIFFERENT constraints. If $a_v = 1$ we remove v from domains of variables in $S \setminus T$ and from domains of variables in $T \setminus S$. If $a_v = 0$ we remove v from domains of variables in $S \cap T$. This shrinks the universe of values to $|S \cap T|$ different values for the variables in $S \cap T$ and to $d - |S \cap T|$ different values for variables in $(S \setminus T) \cup (T \setminus S)$. Consider the shrinking procedure on Example 6.9. Consider a solution $a_1 = 1, a_2 = 0, a_3 = 1, a_4 = 0$ and $a_5 = 1$. Then we remove value $\{1, 3, 5\}$ from domains of X_1, X_2, X_6, X_7 and values $\{2, 4\}$ from domains of X_3, X_4, X_5. We get the following domains:

		1	2	3	4	5
$S \setminus T$	X_1		*		*	
	X_2		*			
$S \cap T$	X_3	*		*		
	X_4	*		*		
	X_5	*		*		*
$T \setminus S$	X_6		*			
	X_7		*		*	

Now three partitions of variables are independent: no values appears in both $S \setminus T$ and $S \cap T$ or in both $T \setminus S$ and $S \cap T$, while $S \setminus T$ and $T \setminus S$ are free to share values. Therefore, we only need to show that the Hall condition is satisfied in each partition.

We renumber these values consecutively from 1 to $|S \cap T|$ for the variables $S \cap T$ and from 1 to $d - |S \cap T|$ for variable in $S \setminus T$ and $T \setminus S$. We denote new domains of variables $D'(X_i)$, $i = 1, \ldots, n$. Note that domains D' are intervals, because the original domains are intervals and the values that we remove are moved from all variables of a each partition.

In our example, $|S \cap T| = 3$, $d - |S \cap T = 2|$. We renumber 3 to 2 and 5 to 3 which

leads to the following domains of variables in $|S \cap T|$

		1	2	3
$S \cap T$	X_3	$*$	$*$	
	X_4	$*$	$*$	
	X_5	$*$	$*$	$*$

We renumber 2 to 1 and 4 to 2 for variables in $S \setminus T$ and $T \setminus S$ and get

		1	2
$S \setminus T$	X_1	$*$	$*$
	X_2	$*$	
$T \setminus S$	X_6	$*$	
	X_7	$*$	$*$

Solution of ALL-DIFFERENT$([S \cap T])$**.** Consider the partition $S \cap T$ and the ALL-DIFFERENT$([S \cap T])$ constraint.The proof is identical to the proof of Theorem 6.8, because this proof does not depend on the parameter c_{lu}.

Solution of ALL-DIFFERENT$([S \setminus T])$**.** Consider the partition $S \setminus T$ ($T \setminus S$ is symmetric). Note that values in domains of variables in this partition are values v such that $a_v = 0$ in the solution of the GEN-SEQUENCE$_{oalldiff}$ constraint.

Suppose there exists an interval $[e, f]$ which violates the Hall condition, so that $P = \{X_i | X_i \in S \setminus T, D'(X_i) \subseteq [e, f]\}$ such that $|P| > f - e + 1$. Consider the tightest interval, $[l, u]$ that contains all variables in P before the shrinking procedure. As the AMONG$_{lu}([a_l, \ldots, a_u], b_{lu}, c_{lu})$, where $b_{lu} = |\{X_i | X_i \in S \cap T, D(X_i) \in [l, u]\}|$ and $c_{lu} = u - l + 1 - \max(|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l, u]\}|, |\{X_i \mid X_i \in T \setminus S, D(X_i) \subseteq [l, u]\}|)$, is satisfied, then $\sum_{i=l}^{u} a_i \leq c_{lu}$. W.l.o.g. we assume that $|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l, u]\}| > |\{X_i \mid X_i \in T \setminus S, D(X_i) \subseteq [l, u]\}|$. Hence,

$$\sum_{i=l}^{u} a_i \leq u - l + 1 - |\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l, u]\}|$$

$$|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l, u]\}| \leq u - l + 1 - \sum_{i=l}^{u} a_i.$$

Note that $u - l + 1 - \sum_{i=l}^{u} a_i = |\{i | (a_i = 0, i \in [l, u]\}|$ in the solution of the GEN-SEQUENCE constraint. Hence, $|\{X_i \mid X_i \in S \setminus T, D(X_i) \subseteq [l, u]\}| \leq |\{i | (a_i = 0, i \in [l, u]\}|$. Note that

$$|\{X_i | X_i \in S \setminus T, D(X_i) \subseteq [l, u]\}| = |\{X_i | X_i \in S \setminus T, D'(X_i) \subseteq [e, f]\}|,$$

as the interval $[l, u]$ was the interval in the original that contained all variables in the violated set P. Moreover, $|\{i|(a_i = 0, i \in [l, u]\}|$ is exactly the number of values that are left in the interval $[l, u]$ after the shrinking procedure. Therefore, $|\{X_i \mid X_i \in S \setminus T, D'(X_i) \subseteq [e, f]\}| \leq \{i|(a_i = 0, i \in [l, u]\} = f - e + 1$. This contradicts our assumption that $|\{X_i|X_i \in S \setminus T, D'(X_i) \in [e, f]\}| > f - e + 1$. $\diamond$

From Theorem 6.11 it follows that

Corollary 6.1 *Detecting bounds disentailment of the* OVERLAPPINGALLDIFF *constraint can be done in* $O(dn^2)$ *time.*

Proof: Detecting domain disentailment of the GEN-SEQUENCE$_{oalldiff}$ constraint can be done in $O(nm)$ time, where n is the number of variables and m is the number of AMONG constraints (Theorem 4.4). In our case, $n = d$ and $m = O(n^2)$. Therefore, we can detect bounds disentailment in $O(dn^2)$ time. $\diamond$

Using the failed literal test procedure together with Theorem 6.11 we get that :

Corollary 6.2 *Enforcing bounds consistency on the* OVERLAPPINGALLDIFF *constraint can be done in* $O(d^2n^3)$ *time.*

We can improve the complexity of enforcing bounds consistency on the OVERLAPPINGALLDIFF constraint if we do a binary search on variable domains in the following way. Consider, for example, a variable X with the domain $D(X) = [l, u]$. We are looking for a support for $lb(X)$. At the first step we temporarily fix the domain of X to the first half so that $D(X) = [l, (u - l)/2]$ and run the bounds disentailment detection algorithm. If this algorithm fails, we halved the search and repeat with the other half. If this algorithm does not fail, we know that there is a value in $[l, (u - l)/2]$ that has a bounds support. Hence, we continue with the binary search within this half. As each test takes $O(dn^2)$ time and there are n variables to prune, the total running time is $O(n^3 d \log d)$.

Example 6.10 *Consider our running example. Suppose we set* X_6 *to 3 and the domains of*

the other variables are not changed.

		1	2	3	4	5
$S \setminus T$	X_1	$*$	$*$	$*$	$*$	
	X_2	$*$	$*$			
$S \cap T$	X_3	$*$	$*$	$*$		
	X_4	$*$	$*$	$*$		
	X_5	$*$	$*$	$*$	$*$	$*$
$T \setminus S$	X_6			$*$		
	X_7		$*$	$*$	$*$	$*$

We construct the GEN-SEQUENCE$_{oalldiff}$ *constraint according to Definition 6.8. We introduce five Boolean variables* $a_1, \ldots, a_5$*. The set of lower bounds is* $\{1, 2, 3\}$ *and the set of upper bounds is* $\{2, 3, 4, 5\}$*. Therefore, we introduce constraints*

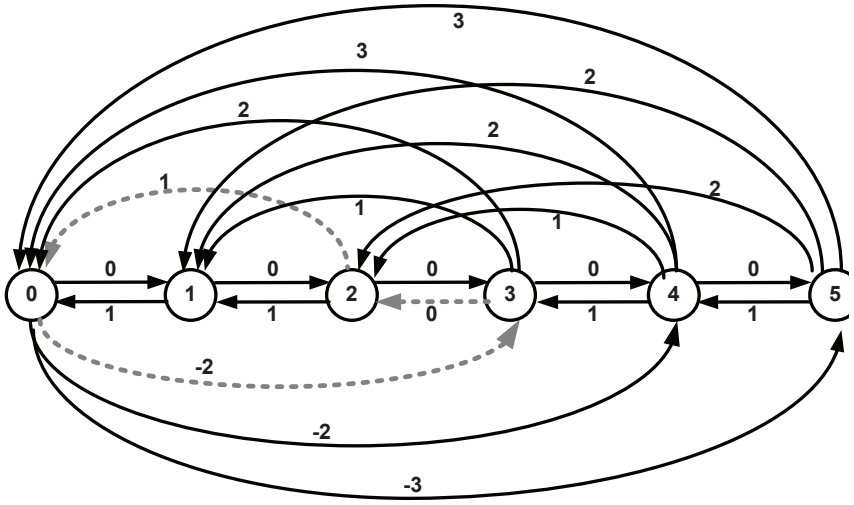
$$\text{AMONG}_{15}([a_1, \ldots, a_5], 3, 3) \qquad \text{AMONG}_{12}([a_1, a_2], 0, 2 - \max(1, 0))$$
$$\text{AMONG}_{13}([a_1, a_2, a_3], 2, 3 - \max(1, 1)) \qquad \text{AMONG}_{14}([a_1, \ldots, a_4], 2, 4 - \max(2, 1))$$
$$\text{AMONG}_{15}([a_1, \ldots, a_5], 3, 5 - \max(2, 2)) \qquad \text{AMONG}_{22}([a_2], 0, 1 - \max(0, 0))$$
$$\text{AMONG}_{23}([a_2, a_3], 0, 2 - \max(0, 1)) \qquad \text{AMONG}_{24}([a_2, a_3, a_4], 0, 3 - \max(0, 1))$$
$$\text{AMONG}_{25}([a_2, \ldots, a_5], 0, 4 - \max(0, 2)) \qquad \text{AMONG}_{33}([a_3], 0, 1 - \max(0, 1))$$
$$\text{AMONG}_{34}([a_3, a_4], 0, 2 - \max(0, 1)) \qquad \text{AMONG}_{35}([a_3, a_4, a_5], 0, 3 - \max(0, 1)))$$

Using the dual formulation of the ILP encoding of the GEN-SEQUENCE$_{oalldiff}$ *constraint we construct a flow graph (Figure 6.1). We omit several edges that correspond to parameter* b_{lu}*, where* $b_{lu} = 0$*. Gray dashed edges highlight a negative cycle in the graph. This implies that the* OVERLAPPINGALLDIFF *is unsatisfiable. Indeed, if* $X_6 = 3$*, then* X_3 *and* X_4 *cannot take value* 3*. Therefore, 3 variables* X_2, X_3, X_4 *in* S *are contained inside the interval* $[1, 2]$*. This leads to a violation.* $\diamond$

Theorem 6.12 *Enforcing DC on the* GEN-SEQUENCE$_{oalldiff}$ *constraint specified by Definition 6.8 and constraints* (6.16) *does not enforce bounds consistency on the* OVERLAPPINGALLDIFF *constraint*

Proof: From Theorem 6.9 it follows that the decomposition does not enforce bounds consistency on individual ALL-DIFFERENT constraints, therefore, it does not enforce bounds consistency on the OVERLAPPINGALLDIFF constraint. $\diamond$

Figure 6.1 The flow graph that corresponds to the dual *ILP* model of the GEN-SEQUENCE$_{oalldiff}$ constraint (Example 6.10)



Decomposition of the OVERLAPPINGALLDIFF **constraint** We show that we can encode bounds disentailment detection of the OVERLAPPINGALLDIFF constraint using a reformulation based on partial sums encoding of the GEN-SEQUENCE constraint. As in Section 6.4.3, we cannot use the decomposition immediately, because parameters of the GEN-SEQUENCE constraint are changing during the search (Definition 6.8). Therefore, we combine the partial sums encoding and the bounds consistency (or range consistency) decomposition of the ALL-DIFFERENT constraint. We introduce Boolean variables A_{ilu}, B_{il} to represent whether X_i takes a value in the interval $[l, u]$, $(\inf, l]$, respectively, and the variables C^S, C^{ST} and C^T, which represent bounds on the number of variables from each set $S \setminus T$, $T \setminus S$ and $S \cap T$ that may take values in the interval $[l, u]$. We introduce the following set of constraints for $1 \leq i \leq n$, $1 \leq l \leq u \leq d$ and $u - l < n$:

$$B_{il} = 1 \iff X_i \leq l \tag{6.17}$$

$$A_{ilu} = 1 \iff (B_{i(l-1)} = 0 \wedge B_{iu} = 1) \tag{6.18}$$

$$C^{ST}_{lu} = \sum_{i \in S \cap T} a_{ilu} \tag{6.19}$$

$$C^{S}_{lu} = \sum_{i \in S \setminus T} a_{ilu} \tag{6.20}$$

$$C^{T}_{lu} = \sum_{i \in T \setminus S} a_{ilu} \tag{6.21}$$

$$C^{ST}_{1u} = C^{ST}_{1l} + C^{ST}_{(l+1)u} \tag{6.22}$$

$$C^{ST}_{lu} \leq u - l + 1 - C^{S}_{lu} \tag{6.23}$$

$$C^{ST}_{lu} \leq u - l + 1 - C^{T}_{lu} \tag{6.24}$$

Theorem 6.13 *Decomposition* (6.17)–(6.24) *detects bounds disentailment of the* OVERLAPPINGALLDIFF *constraint.*

Proof: Constraints (6.22) subsume the partial sums decomposition of the GEN-SEQUENCE constraint that encodes the ALL-DIFFERENT$([S \cap T])$ constraint. Constraints (6.19)–(6.24) and (6.23)–(6.24) correctly adjust parameters b_{lu} and c_{lu}, $1 \leq l \leq u \leq d$ and $u - l < n$ for each AMONG$_{lu}$ constraint, because the lower bound of C_{lu}^{ST} computes $|\{X_i | X_i \in S \cap T\}|$, the lower bound of C_{lu}^{S} computes $|\{X_i | X_i \in S \setminus T\}|$ and the lower bound of C_{lu}^{T} computes $|\{X_i | X_i \in T \setminus S\}|$. $\diamond$

Next we show that the decomposition (6.17)–(6.24) does not enforce bounds consistency on the OVERLAPPINGALLDIFF constraint. However, unlike the algorithm proposed in Section 6.4.3, it does enforce bounds consistency on individual constraints — ALL-DIFFERENT$([S])$ and ALL-DIFFERENT$([T])$.

Theorem 6.14 *Decomposition* (6.17)–(6.24) *does not enforce bounds consistency of the* ALL-DIFFERENT$([S])$ *and* ALL-DIFFERENT$([T])$ *constraint but does enforce bounds consistency on* OVERLAPPINGALLDIFF(S, T) *constraints.*

Proof: To show that the decomposition does not enforce bounds consistency we consider the following example of the OVERLAPPINGALLDIFF$([X_1, \ldots, X_4], S, T)$, $S = \{X_1, X_2, X_3\}$ and $T = \{X_2, X_3, X_4\}$ with the following domains:

	1	2	3	4
X_1	$*$	$*$		
X_2	$*$	$*$	$*$	
X_3		$*$	$*$	$*$
X_4		$*$	$*$	

From constraints (6.23)–(6.24) we derive $C_{1,2}^{ST} = [0, 1]$ and $C_{2,3}^{ST} = [0, 1]$ and no more propagation happens. However, assigning $X_3 = 2$ falsifies the constraint.

The decomposition enforces bounds consistency on ALL-DIFFERENT$([S])$ (ALL-DIFFERENT$([T])$ is similar) because constraints (6.17)–(6.20) and (6.23) subsume the *BC* decomposition of the ALL-DIFFERENT constraint (6.2)–(6.4). $\diamond$

Example 6.11 *We demonstrate how the constraints* (6.17)–(6.24) *work on our running example 6.7. Consider the interval* $[1, 4]$ *that contains variables* $\{X_1, X_2, X_3, X_4\}$. $lb(C_{14}^{ST}) \geq 2$ *and* $lb(C_{14}^{S}) \geq 2$. *Therefore, the constraint* $C_{14}^{ST} + C_{14}^{S} \leq 4$ *forces* $C_{14}^{ST} = 2$

and $C_{14}^{S} = 2$. *The interval* $[1, 4]$ *is saturated, as* $lb(C_{14}^{ST}) = ub(C_{14}^{ST})$. *Hence, by* (6.17)–(6.19)*, values* $[1, 4]$ *are removed from* $D(X_5)$.

		1	2	3	4	5
$S \setminus T$	X_1	*	*	*	*	
	X_2	*	*			
$S \cap T$	X_3	*	*	*		
	X_4	*	*	*		
	X_5					*
$T \setminus S$	X_6		*	*		
	X_7		*	*	*	*

As $lb(C_{55}^{ST}) = ub(C_{55}^{ST}) = 1$*, by channelling constraints* (6.17)–(6.18) *the value 5 is removed from the domain of the variable* X_7.

		1	2	3	4	5
$S \setminus T$	X_1	*	*	*	*	
	X_2	*	*			
$S \cap T$	X_3	*	*	*		
	X_4	*	*	*		
	X_5					*
$T \setminus S$	X_6		*	*		
	X_7		*	*	*	

Consider the interval $[1, 3]$ *and variables* $\{X_2, X_3, X_4\}$. $ub(C_{13}^{S}) \leq 1 \Rightarrow$ (6.18) $\Rightarrow$ $ub(C_{12}^{S}) \leq 1$. *The interval* $[1, 2]$ *is saturated, as* $lb(C_{12}^{S}) = ub(C_{12}^{ST})$. *Hence, by* (6.17)-(6.18),(6.20)*,* $[1, 2]$ *is removed from* $D(X_1)$. *Similarly,* $[2, 3]$ *is removed from* $D(X_7)$.

		1	2	3	4	5
$S \setminus T$	X_1				*	
	X_2	*	*			
$S \cap T$	X_3	*	*	*		
	X_4	*	*	*		
	X_5					*
$T \setminus S$	X_6		*	*		
	X_7				*	

Bounds disentailment. *To show that the decomposition detects bounds disentailment we set X_3 to 2. By the constraint $C_{22}^{ST} + C_{22}^{S} \leq 1$ we get $C_{22}^{S} = 0$ and by channelling constraints* (6.17)–(6.18) *the value 2 is removed from the domain of the variable X_2. This sets X_2 to 1 and causes pruning of the value 1 from X_3.*

		1	2	3	4	5
$S \setminus T$	X_1				$*$	
	X_2	$*$				
$S \cap T$	X_3		$*$	$*$		
	X_4		$*$			
	X_5					$*$
$T \setminus S$	X_6		$*$	$*$		
	X_7				$*$	

Finally, consider the interval $[2, 3]$. We have that $lb(C_{23}^{ST}) \geq 2$ and $lb(C^{T}) \geq 1$. This violates the constraint that $C_{23}^{ST} + C_{23}^{T} \leq 2$. $\diamond$

6.7.2 Exponential separation

We now give an artificial problem on which our new propagator does exponentially less work than existing methods to propagate two individual ALL-DIFFERENT constraint separately.

Theorem 6.15 *There exists a class of problems such that enforcing bounds disentailment on* OVERLAPPINGALLDIFF *detects unsatisfiability without search, while a search method that enforces bounds consistency on the decomposition into* ALL-DIFFERENT *constraints explores an exponential sized search tree regardless of the branching heuristic.*

Proof:

$$\begin{aligned}
\mathcal{I}_n \quad = \quad & \text{ALL-DIFFERENT}([X_1, \ldots, X_n, Y_1, \ldots, Y_{2n}]) \wedge \\
& \text{ALL-DIFFERENT}([Y_1, \ldots, Y_{2n}, Z_1, \ldots, Z_n]) \wedge \\
& D(X_i) = [1, 2n-1],\ \forall i = 1, \ldots, n \wedge \\
& D(Y_i) = [1, 4n-1],\ \forall i = 1, \ldots, 2n \wedge \\
& D(Z_i) = [2n, 4n-1],\ \forall i = 1, \ldots, n
\end{aligned}$$

The OVERLAPPINGALLDIFF **constraint.** We will use the decomposition (6.17)– (6.24) for bounds disentailment detection.

Consider the interval $[1, 4n-1]$. $lb(C^{ST}_{1,4n-1}) \geq 2n$, $lb(C^{S}_{1,2n-1}) \geq n$ and $lb(C^{T}_{2n,4n-1}) \geq n$. The constraint $C^{ST}_{1,2n-1}+C^{S}_{1,2n-1} \leq 2n-1$ implies that $C^{ST}_{1,2n-1} \leq n-1$. The constraint $C^{ST}_{2n,4n-1}+C^{T}_{2n,4n-1} \leq 2n$ implies that $C^{ST}_{2n,4n-1} \leq n$. Finally, the constraint $C^{ST}_{1,4n-1} = C^{ST}_{1,2n-1} + C^{ST}_{2n,4n-1}$ is violated because $ub(C^{ST}_{1,2n-1}) + ub(C^{ST}_{2n,4n-1}) \leq 2n-1$ and $lb(C^{ST}_{1,4n-1}) \geq 2n$.

Decomposition. Consider any ALL-DIFFERENT constraint. A subset of variables of size at most n has at least $2n-1$ different values in their domains and a subset of size between $n+1$ and $3n$ has $4n-1$ values in their domains. Therefore, to obtain a Hall interval and prune, we must instantiate at least $n-1$ variables. Reasoning about the second ALL-DIFFERENT is similar. Therefore, the search tree has $\Omega(n!)$ nodes. $\diamond$

6.7.3 Other related work

Little work has been done for propagating conjunctions of ALL-DIFFERENT constraints. Enforcing domain consistency on the constraint was shown to be NP-hard in [KEKM08]. In the same work, the authors proposed approximation algorithms for the simultaneous matching problem. It is unclear, however, if they can be used for building constraint propagation algorithms.

In [AHHT07], it was proposed that communication between different constraints can be improved by generalising pruning to removing paths from a multi-valued decision diagram. Propagating conjunctions of ALL-DIFFERENT constraints was proposed as an application, but the level of consistency enforced in this setting is not established in [AHHT07]. The authors demonstrated that their approach works well on problems with three overlapping PERMUTATION constraints where the PERMUTATION constraints overlap on most of the variables. As all problems were solved without backtracks I conjecture that for these random problems the authors achieve a level of consistency which is close to domain consistency.

In [MMA10], a system of ALL-DIFFERENT constraints was investigated from the integer linear programming point of view. In particular, the authors presented a system of inequalities that describes the convex hull of a system of ALL-DIFFERENTs if the system of ALL-DIFFERENTs satisfies the inclusion property. For each ALL-DIFFERENT$_i$, $i = 1, \ldots, n$ we distinguish between variables that belong to the scope of the ALL-DIFFERENT$_i$ constraint and the remaining variables that we denote T_i. The inclusion property requires that for any two constraints, ALL-DIFFERENT$_i(\mathbf{X}_i)$ and ALL-DIFFERENT$_j(\mathbf{X}_j)$, we have that either $T_i \subset T_j$ or $T_j \subset T_i$.

Table 6.4: Random problems. n is the size of X, Y and Z, o is the size of W, d is the size of variable domains. Number of instances solved in 300 seconds out of 50 runs / average backtracks/average time to solve.

n,d,o	BC	DC	oBC
	#s / #bt / t	#s / #bt / t	#s / #bt / t
4, 15, 10	14 /2429411 / 61.8	41 /1491341 / 52.1	**42** / **17240** / **32.5**
4, 16, 11	6 /5531047 / 153.7	22 /1745160 / 67.9	**31** / **8421** / **19.5**
4, 17, 12	1 / 17 / 0	6 /2590427 / 100.9	**24** / **8185** / **21.5**
5, 16, 10	11 /3052298 / 82.0	37 /1434903 / 58.2	**42** / **20482** / **48.5**
5, 17, 11	2 /3309113 / 94.5	19 /2593819 / 114.6	**26** / **4374** / **15.8**
5, 18, 12	1 / 17 / 0	4 /2666556 / 133.1	**22** / **3132** / **12.2**
6, 17, 10	11 /2845367 / 79.1	31 /1431671 / 66.3	**40** / **6796** / **21.9**
6, 18, 11	4 / 199357 / 6.6	16 /1498128 / 80.2	**31** / **4494** / **17.5**
6, 19, 12	4 /3183496 / 110.0	5 /1035126 / 66.2	**27** / **3302** / **15.5**
TOTALS			
sol/total	54 /450	181 /450	**285** /450
avg time for sol	78.072	70.551	**24.689**
avg bt for sol	2818926	1666568	**9561**

Finally, [LMS$^+$09] proposed several rules to improve propagation between multiple ALL-DIFFERENT constraints. The authors also proposed an encoding of the multiple ALL-DIFFERENT constraints into CNF that requires an exponential size in the worst case.

6.7.4 Experimental results

To evaluate the performance of our decomposition we carried out a series of experiments on random problems. We ran experiments with Ilog 6.2 solver on an Intel Xeon 4 CPU, 2.0 Ghz, 4Gb RAM. We compare performance of the domain consistency, bounds consistency [LOQTvB03] propagators and our decomposition into constraints (6.17)-(6.24) for the OVERLAPPINGALLDIFF constraint (OBC). We use the following problems. We have three global constraints: ALL-DIFFERENT$(X \cup W)$, ALL-DIFFERENT$(Y \cup W)$ and ALL-DIFFERENT$(Z \cup W)$. We also randomly post a linear number of binary ordering relations between variables in X, Y and Z. We post three OVERLAPPINGALLDIFF constraints in OBC model instead of three ALL-DIFFERENTS: OVERLAPPINGALLDIFF$([X \cup W, Y \cup W])$, OVERLAPPINGALLDIFF$([X \cup W, Z \cup W])$ and OVERLAPPINGALLDIFF$([Z \cup W, Y \cup W])$. We use a random variable ordering and run each instance with 50 different seeds. As can be seen from Table 6.4, our decomposition reduces the search space significantly, is much faster and solves more instances overall.

6.8 The GCC constraint

Another generalisation of the ALL-DIFFERENT constraint is the global cardinality constraint , $\text{GCC}([X_1, \ldots, X_n], [l_1, \ldots, l_d], [u_1, \ldots, u_d])$. This constraint ensures that the value i occurs between l_i and u_i times in X_1 to X_n. The GCC constraint is useful in resource allocation problems where values represent resources. Another example is the car sequencing problem (prob001 at *CSP*Lib.org), where we can post a GCC constraint to ensure that the correct number of cars of each type is put on the assembly line. Let us introduce a running example:

Example 6.12 (Running example (GCC)) *Consider a resource allocation problem in a goods store. There are five activities that need to be performed in the shop on daily basis, like monitoring security, assist customers, etc. There are five employees in the store which are available resources in this problem. Each employee is qualified to perform one or more activities. During a day an employee can be assigned to a single activity. We need to construct a schedule for a day so that at least one employee is assigned to each of the activities* $1, 2, 4$ *and* 5 *and at most one employee is assigned to the* 3*rd activity. We can encode this problem using a* GCC *constraint.*

We introduce five variables, $X_1, \ldots, X_5$*, one for each employee. Domains of the variables show activities that each employee is qualified to work on. Using the values* $[l_1, \ldots, l_5]$ *and* $[u_1, \ldots, u_5]$ *we make sure that each of activities* $1, 2, 4, 5$ *is performed during a day and the third activity is assigned to at most one person. The following tables show domains of variables and upper and lower bounds on the occurrences of values:*

v	1	2	3	4	5
X_1	$*$				
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$		
X_4	$*$	$*$	$*$	$*$	$*$
X_5	$*$	$*$	$*$	$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	1	5	5

$\diamond$

6.8.1 Filtering algorithm for the GCC constraint

The GCC constraint is one of the most useful constraints. Therefore, a number of filtering algorithms that achieve domain consistency [Reg96], range consistency and bounds consistency [KT03, QLOvBG04] have been proposed for it. In this section we recall the bounds consistency algorithm for the GCC constraint that was proposed by Quimper *et al.* [QLOvBG04], because our decomposition is based on this algorithm .

The filtering algorithm is based on the observation that the $\text{GCC}([X_1, \ldots, X_n], [l_1, \ldots, l_d], [u_1, \ldots, u_d])$ constraint can be split into two constraints

1. the upper bound constraint (GCC^U) that only takes into account the upper bounds parameters – $\text{GCC}([X_1, \ldots, X_n], [0, \ldots, 0], [u_1, \ldots, u_d])$ and
2. the lower bound constraint (GCC^L) that only takes into account the lower bounds parameters – $\text{GCC}([X_1, \ldots, X_n], [l_1, \ldots, l_d], [n, \ldots, n])$.

Quimper*et al.* proved that it is sufficient to enforce *BC* on the GCC^U and GCC^L independently.

Before we describe the algorithm we introduce notations. Let S be a set of values. We use I_S for the number of variables X whose domains $D(X_i)$ intersect the set S,

$$I_S = |\{X_i | D(X_i) \cap S \neq \emptyset\}|, \tag{6.25}$$

and U_S for the number of variables X_i whose domains are subsets of S,

$$U_S = |\{X_i | D(X_i) \subseteq S\}|. \tag{6.26}$$

We recall that we denote the set of all lower bounds of variables X as $LB = \bigcup_{i=1}^{n} \{lb(D(X_i))\}$ and the set of all upper bounds of variables X as $UB = \bigcup_{i=1}^{n} \{ub(D(X_i))\}$. We point out that we slightly change notation from [QLOvBG04] to differentiate from Definition 6.6.

6.8.1.1 Filtering algorithm for the GCC^U constraint.

First we consider the GCC^U constraint. The bounds consistency and range consistency filtering algorithms are based on the notion of an extended Hall interval:

Definition 6.9 (Extended Hall interval) *An interval $[l, u]$ is an extended Hall interval if and only if $U_{[l,u]} = \sum_{v=l}^{u} u_v$.*

Theorems 6.17 and 6.18 provide necessary and sufficient conditions for range consistency and bounds consistency, respectively. These theorems can be seen as generalisations of Theorem 6.2 and Theorem 6.3 for the ALL-DIFFERENT constraint.

Theorem 6.16 (Range/Bounds disentailment, Lemma 5.1 [Qui06]) *The* $\text{GCC}^U([X_1, \ldots, X_n], [0, \ldots, 0], [u_1, \ldots, u_d])$ *constraint is satisfiable if and only if for any interval* $[l, u]$ *it holds that:*

1. $D(X_i) \neq \emptyset$, $i = 1, \ldots, n$ *and*
2. *for each extended interval* $[l, u]$, $U_{[l,u]} \leq \sum_{v=l}^{u} u_i$,

where $l \in LB$ *and* $u \in UB$.

Theorem 6.17 (Range consistency, follows from Lemma 3 [QLOvBG04]) *The* $\text{GCC}^U([X_1, \ldots, X_n], [0, \ldots, 0], [u_1, \ldots, u_d])$ *constraint is range consistent if and only if*

1. *Theorem 6.16 conditions 1– 2 hold and*
2. *for each extended Hall interval of values* $[l, u]$ *and each* X_i*:*

   if $D(X_i)$ *is not fully contained in the interval* $[l, u]$ *then* $D(X_i) \cap [l, u] = \emptyset$, $i = 1, \ldots, n$.

Theorem 6.18 (Bounds consistency, follows from Lemma 3 [QLOvBG04]) *The* $\text{GCC}^U([X_1, \ldots, X_n], [0, \ldots, 0], [u_1, \ldots, u_d])$ *constraint is bounds consistent if and only if*

1. *Theorem 6.16 conditions 1– 2 conditions hold and*
2. *for each extended Hall interval of values* $[l, u]$ *and each* X_i*:*

   if $D(X_i)$ *is not fully contained in the interval* $[l, u]$ *then* $\{lb(X_i), ub(X_i)\} \cap [l, u] = \emptyset$, $i = 1, \ldots, n$.

The bounds consistency algorithm for the GCC constraint runs in $(n + n \log n)$ time. The range consistency algorithm is more expensive and runs in $O(n \log n + n|HI|)$ time, where $|HI|$ is the number of Hall intervals. This algorithm enforces bounds consistency on the GCC^U constraint, then finds Hall intervals and prunes variable domains according to Theorem 6.17 to achieve range consistency.

Consider how to achieve range consistency and bounds consistency on our running example if we ignore the information about lower bound parameters l_v, $v = 1, \ldots, d$.

Example 6.13 ***Bounds consistency***. *We consider the interval* $[3,3]$. *This interval is an extended Hall interval because* $U_{[3,3]} = 1$ *and* $\sum_{v=3}^{3} u_v = 1$. *However, none of the variables bounds overlaps the interval* $[3,3]$. *Therefore, by Theorem 6.18,* GCC^U *is bounds consistent .*

v	1	2	3	4	5
X_1	$*$				
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$		
X_4	$*$	$*$	$*$	$*$	$*$
X_5	$*$	$*$	$*$	$*$	$*$
l_v	0	0	0	0	0
u_v	5	5	1	5	5

Range consistency. *Consider again the interval* $[3,3]$ *that is an extended Hall interval. By Theorem 6.18, we can remove value* 3 *from the domains of* X_2, X_4 *and* X_5. *This leaves the following domains:*

v	1	2	3	4	5
X_1	$*$				
X_2	$*$	$*$		$*$	$*$
X_3			$*$		
X_4	$*$	$*$		$*$	$*$
X_5	$*$	$*$		$*$	$*$
l_v	0	0	0	0	0
u_v	5	5	1	5	5

By Theorem 6.17 the constraint is range consistent . $\diamond$

6.8.1.2 Filtering algorithm for the GCC^L constraint.

Next we consider the GCC^L constraint. The bounds consistency and range consistency filtering algorithms are based on the notion of an unstable set:

Definition 6.10 (Unstable set) *A set* S *is an unstable set if and only if* $I_S = \sum_{v \in S} l_v$.

Theorems 6.20 and 6.21 provide necessary and sufficient conditions for range consistency and bounds consistency, respectively.

Theorem 6.19 (Range/Bounds disentailment, Lemma 5.2 [Qui06]) *The* $\mathrm{GCC}^L([X_1,\ldots,X_n],[l_1,\ldots,l_m],[n,\ldots,n])$ *constraint is satisfiable if and only if for any interval* $[l,u]$ *it holds that:*

1. $D(X_i) \neq \emptyset$, $i = 1,\ldots,n$ *and*
2. *for each set of values* S, $I_S \geq \sum_{v\in S} l_v$.

Theorem 6.20 (Range consistency, Lemma 4 [QLOvBG04]) *The* $\mathrm{GCC}^L([X_1,\ldots,X_n],$ $[l_1,\ldots,l_m],[n,\ldots,n])$ *constraint is range consistent if and only if*

1. *Theorem 6.19 conditions 1–2 hold and*
2. *for each unstable set* S*:*

   if $D(X_i)\cap S \neq \emptyset$ *then* $D(X_i)\subseteq S$, $i = 1,\ldots,n$.

Theorem 6.21 (Bounds consistency, follows from Lemma 4 [QLOvBG04]) *The* $\mathrm{GCC}^L([X_1,\ldots,X_n],[l_1,\ldots,l_m],[n,\ldots,n])$ *constraint is bounds consistent if and only if*

1. *Theorem 6.19 conditions 1–2 hold and*
2. *for each unstable set* S*:*

   if $D(X_i)\cap S \neq \emptyset$ *then* $\{lb(D(X_i)), ub(D(X_i))\}\subseteq S$, $i = 1,\ldots,n$.

The filtering algorithm proposed in [QLOvBG04] is based on detection of the unstable sets. The algorithm enforces bounds consistency on the GCC^L in $O(n\log n + n)$ time and range consistency in $O(n\log n + n|US|)$ time, where $|US|$ is the number of unstable sets. This algorithm enforces bounds consistency on the GCC^L constraint, then finds unstable sets and prunes variable domains according to Theorem 6.20.

Consider how to achieve range consistency and bounds consistency on our running example if we ignore the information about upper bound parameters u_v, $v = 1,\ldots,d$.

Example 6.14 . ***Bounds consistency****. We consider the set* $\{2,4,5\}$*. This set is an unstable set because* $I_{\{2,4,5\}} = 3$ *and* $\sum_{v\in\{2,4,5\}} l_v = 3$*. The lower bounds of variables* X_2, X_4 *and* X_5 *are outside the unstable set. Therefore, by Theorem 6.21, we remove value* 1 *from domains of variables* X_2, X_4 *and* X_5.

v	1	2	3	4	5
X_1	$*$				
X_2		$*$	$*$	$*$	$*$
X_3			$*$		
X_4		$*$	$*$	$*$	$*$
X_5		$*$	$*$	$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	5	5	5

We consider the set $\{1, 2, 4, 5\}$. *This set is an unstable set because* $I_{\{1,2,4,5\}} = 4$ *and* $\sum_{v\in\{1,2,4,5\}} l_v = 4$. *However, none of the variables* X_2, X_4 *and* X_5 *bounds are outside this set. By Theorem 6.21, the* GCC^L *constraint is bounds consistent .*

Range consistency*. Consider again the unstable set* $\{1, 2, 4, 5\}$. *By Theorem 6.20, we remove the value* 3 *from domains of variables* X_2, X_4 *and* X_5.

v	1	2	3	4	5
X_1	$*$				
X_2		$*$		$*$	$*$
X_3			$*$		
X_4		$*$		$*$	$*$
X_5		$*$		$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	5	5	5

We have processed all unstable sets and, by Theorem 6.20, the GCC^L *constraint is range consistent .* $\diamond$

6.8.2 Decompositions of the GCC constraint

We can decompose range consistency and bounds consistency filtering algorithm for the GCC constraint in a similar way to decomposing ALL-DIFFERENT using the GEN-SEQUENCE constraint (Section 6.4.3, the decomposition (6.11)–(6.15)). We introduce the additional $O(d^2)$ integer variables, N_{lu} to represent the number of variables whose domains are contained in the interval $[l, u]$. Clearly, $N_{lu} \in [\sum_{i=l}^{u} l_i, \sum_{i=l}^{u} u_i]$ and $N_{1d} = n$.

We then post the following constraints for $1 \leq i \leq n$, $1 \leq l \leq u \leq d$, $1 \leq k < u$:

$$A_{ilu} = 1 \iff X_i \in [l, u] \tag{6.27}$$

$$N_{lu} = \sum_{i=1}^{n} A_{ilu} \tag{6.28}$$

$$N_{1u} = N_{1k} + N_{(k+1)u} \tag{6.29}$$

Constraints (6.27) are channelling constraints between variable A and the original variables X. The integer variables N_{lu} can be seen as 'counter' variables because they keep track of the number of times a value from the interval $[l, u]$ is used (constraints (6.28)). Finally, constraints (6.29) provides communication between 'counter' variables N.

Range consistency. We next show that enforcing domain consistency on constraint (6.27) and bounds consistency on constraints (6.28) and (6.29) enforces range consistency on the GCC constraint.

Theorem 6.22 *Enforcing domain consistency on constraint* (6.27) *and bounds consistency on constraints* (6.28) *and* (6.29) *achieves range consistency on the corresponding* GCC *constraint in* $O(nd^3)$ *time down any branch of the search tree.*

Proof: Bounds disentailment. We first show that if range consistency fails on the GCC then enforcing domain consistency on (6.27) and bounds consistency on constraints (6.28) and (6.29) will fail. By Theorem 6.17 and Theorem 6.20 the GCC constraint fails on a GCC if and only if

1. there exists an violated extended Hall interval $[l, u]$ such that $U_{[l,u]} > \sum_{v=l}^{u} u_v$ or
2. there exists a violated unstable set S such that $I_S < \sum_{v \in S} l_v$.

Consider these two cases.

Violated extended Hall interval. Suppose, there exists a violated extended Hall interval $[l, u]$ with $U_{[l,u]} > \sum_{v=l}^{u} u_v$. By constraints (6.27) we have $\sum_{i=1}^{n} A_{ilu} > U_{[l,u]}$ and by (6.28) we have

$$N_{lu} = \sum_{i=1}^{n} A_{ilu} > U_{[l,u]}.$$

This implies that $N_{lu} > \sum_{v=l}^{u} u_v$, whereas the greatest value in the domain of N_{lu} was set to $\sum_{v=l}^{u} u_v$. So bounds consistency will fail on the constraint $N_{lu} = \sum_{i=1}^{n} A_{ilu}$.

Violated unstable set. Consider the second case. Suppose now that a set $S = \{v_1, \ldots, v_k\}$ is such that $I_S < \sum_{v_i \in S} l_{v_i}$. We denote P the set of variables $\{X_i | D(X_i) \cap$

$S \neq \emptyset\}$. The total number of values taken by variables X_i is equal to n, therefore, the number of variables in $P' = \{X_i | X_i \notin P\}$ is greater than $n - \sum_{v_i \in S} l_{v_i}$. Consider the set of intervals $S' = \{[1, v_1 - 1], \ldots, [v_k + 1, v_d]\}$ that contain variables in P'. Note that domain of any variable X_i, $X_i \in P'$ is contained in at most one interval in S', otherwise X_i would be in P. Therefore,

$$U_{[1,v_1-1]} + U_{[v_1+1,v_2-1]} + \ldots + U_{[v_k+1,d]} > n - \sum_{v_i \in S} l_{v_i}.$$

Thanks to constraints (6.28), we know that for any l, u, $N_{lu} \geq U_{[l,u]}$. So,

$$N_{1(v_1-1)} + N_{(v_1+1)(v_2-1)} + \ldots + N_{(v_k+1)d} \geq$$

$$U_{[1,v_1-1]} + U_{[v_1+1,v_2-1]} + \ldots + U_{[v_k+1,d]} > n - \sum_{v_i \in S} l_{v_i}.$$

The initial domains of N_{lu} variables also tell us that for every v_i in S, $N_{v_i v_i} \geq l_{v_i}$. Thus,

$$\begin{aligned} &lb(N_{1(v_1-1)}) + lb(N_{v_1 v_1}) + lb(N_{(v_1+1)(v_2-1)}) + \ldots \\ &\quad + lb(N_{v_k v_k}) + lb(N_{(v_k+1)d}) > n = ub(N_{1d}) \end{aligned} \tag{6.30}$$

Chaining lower bounds. Successively applying bounds consistency on $N_{1v_1} = N_{1(v_1-1)} + N_{v_1 v_1}$, then on $N_{1(v_2-1)} = N_{1v_1} + N_{(v_1+1)(v_2-1)}$, and so on until $N_{1d} = N_{1v_k} + N_{(v_k+1)d}$ will successively increase the lower bound of the variables so that

$$\begin{aligned}
N_{1v_1} &= N_{1(v_1-1)} + N_{v_1 v_1} \Rightarrow_{BC} \\
lb(N_{1v_1}) &\geq lb(N_{1(v_1-1)}) + lb(N_{v_1 v_1}) \\
&\vdots \\
N_{1v_i} &= N_{1(v_1-1)} + N_{v_i v_i} \Rightarrow_{BC} \\
lb(N_{1v_i}) &\geq lb(N_{1(v_1-1)}) + lb(N_{v_i v_i}) \Leftrightarrow_{chaining} \\
lb(N_{1v_i}) &\geq lb(N_{1(v_1-1)}) + lb(N_{v_1 v_1}) + \ldots + lb(N_{(v_{i-1}+1)(v_i-1)}) + lb(N_{v_i v_i}) \\
&\vdots \\
N_{1v_d} &= N_{v_k v_k} + N_{(v_k+1)d} \Rightarrow_{BC} \\
lb(N_{1v_i}) &\geq lb(N_{v_k v_k}) + lb(N_{(v_k+1)d}) \Leftrightarrow_{chaining} \\
lb(N_{1v_d}) &\geq lb(N_{1(v_1-1)}) + lb(N_{v_1 v_1}) + \ldots + lb(N_{v_k v_k}) + lb(N_{(v_k+1)d})
\end{aligned}$$

This leads to a failure on the variable N_{1d} as $lb(N_{1d}) > ub(N_{1d})$.

We call this successive application of constraints (6.29) to make sure the lower bound of a variable N_{1v_i} is greater than or equal to the sum of lower bounds of variables $N_{1,v_1}, N_{v_1,v_1}, \ldots, N_{(v_{i-1}+1)(v_i-1)}, N_{v_iv_i}$ as the *chaining lower bounds* procedure.

Range consistency. We now show that when domain consistency on (6.27) and bounds consistency on (6.28) and (6.29) do not fail, we prune all values that are pruned when enforcing range consistency on the GCC constraint. Consider a value $v \in D(X_j)$ for some $i \in [1, n]$ such that v does not have any bound support.

From Theorem 6.17 and Theorem 6.20 we conclude that there are two cases when this can happen:

1. there exists an extended Hall interval $[l, u]$ that contains v and $D(X_j)$ is not included in the interval $[l, u]$ or
2. there exists an unstable set S that *does not* contains v and $D(X_j)$ intersects S.

Consider the first case. Suppose there exists an extended Hall interval $[l, u]$, $v \in [l, u]$ such that $U_{[l,u]} = \sum_{v=l}^{u} u_v$. Then,

$$N_{lu} = \sum_{i=1}^{n} A_{ilu} \geq U_{[l,u]} = \sum_{v=l}^{u} u_v.$$

On the other hand, $ub(N_{lu}) = \sum_{v=l}^{u} u_v$ and the variable N_{lu} is fixed to this value. The constraint (6.28) makes sure that

$$\sum_{i=1}^{n} A_{ilu} = \sum_{v=l}^{u} u_v.$$

As there are $\sum_{v=l}^{u} u_v$ variables A_{ilu} that are equal to 1, the remaining unset variables are fixed to 0. In particular, A_{jlu} was an unset variable because $D(X_j) \nsubseteq [l, u]$ and constraint (6.28) ensures that it takes the value 0. As $v \in [l, u]$, v is pruned from $D(X_j)$.

Consider the second case. Let $S = \{v_1, \ldots, v_k\}$ be an unstable set such that $v \notin S$ and $D(X_i) \cap S \neq \emptyset$. W.l.o.g. we assume that $v_i < v < v_{i+1}$ As S is an unstable set we have:

$$I_S = \sum_{v_i \in S} l_{v_i}.$$

We again denote P the set of variables $\{X_i | D(X_i) \cap S \neq \emptyset\}$. The total number of values taken by X_i variables being equal to n, the number of variables in $P' = \{X_i | X_i \notin P\}$ is equal to $n - \sum_{v_i \in S} l_{v_i}$. Consider the set of intervals $S' = \{[1, v_1 - 1], \ldots, [v_k + 1, v_d]\}$ that contain variables in P'. By the above argument we have the following equation for intervals in S':

$$U_{[1,v_1-1]} + U_{[v_1+1,v_2-1]} + \ldots + U_{[v_{k+1},d]} = n - \sum_{v_i \in S} l_{v_i} \tag{6.31}$$

Thanks to constraints (6.28), we know that for any l, u, $N_{lu} \geq U_{[l,u]}$. So,

$$N_{1(v_1-1)} + N_{(v_1+1)(v_2-1)} + \ldots + N_{(v_k+1)d} \geq n - \sum_{v_i \in S} l_{v_i}.$$

The initial domains of N_{lu} variables also tell us that for every v_i in S, $N_{v_i v_i} \geq l_{v_i}$.

$$\begin{aligned} &lb(N_{1(v_1-1)}) + lb(N_{v_1 v_1}) + lb(N_{(v_1+1)(v_2-1)}) + \ldots \\ &\quad + lb(N_{v_k v_k}) + lb(N_{(v_k+1)d}) \geq n = ub(N_{1d}) \end{aligned} \tag{6.32}$$

By the chaining lower bounds procedure we derive that $lb(N_{1d}) \geq n = ub(N_{1d})$. This implies that the variable N_{1d} is fixed to n. Moreover, chaining lower bounds makes sure that

$$lb(N_{1d}) = lb(N_{1(v_i-1)}) + \sum_{i=1}^{k} \left(lb(N_{(v_{i-1}+1)(v_i-1)}) + lb(N_{v_i v_i}) \right) + lb(N_{(v_k+1)d}) \tag{6.33}$$

Chaining upper bounds. As $lb(N_{1d}) = ub(N_{1d})$, equation (6.33) suggests that none of the variables in the RHS of this equation can take a value that is greater than its lower bound. Indeed if we consider the chaining lower bounds procedure in the backward direction: $N_{1d} = N_{1v_k} + N_{(v_k+1)d}$, on $N_{1v_k} = N_{1(v_k-1)} + N_{v_k v_k}$, and so on until $N_{1v_1} = N_{1(v_1-1)} + N_{v_1 v_1}$, this will successively decrease the upper bounds of the variables to their lower bounds:

$$\begin{aligned} N_{1d} &= N_{1v_k} + N_{(v_k+1)d} \\ \text{using } ub(N_{1d}) = lb(N_{1d}) \text{ and } lb(N_{1d}) &\geq lb(N_{1v_k}) + lb(N_{(v_k+1)d}) \Rightarrow_{BC} \\ lb(N_{1v_k}) = ub(N_{1v_k}), &\quad lb(N_{(v_k+1)d}) = ub(N_{(v_k+1)d}) \\ &\vdots \\ N_{1v_1} &= N_{1(v_1-1)} + N_{v_1 v_1} \\ \text{using } ub(N_{1v_1}) = lb(N_{1v_1}) \text{ and } lb(N_{1v_1}) &\geq lb(N_{1(v_1-1)}) + lb(N_{v_1 v_1}) \Rightarrow_{BC} \\ lb(N_{1(v_1-1)}) = ub(N_{1(v_1-1)}), &\quad lb(N_{v_1 v_1}) = ub(N_{v_1 v_1}) \end{aligned}$$

This leads to fixing the value of the variables $N_{1,v_1}, N_{v_1,v1}, \ldots, N_{(v_{i-1}+1)(v_i-1)}, N_{v_i v_i} \ldots, N_{1d}$ to their lower bounds. We call this the *chaining upper bounds* procedure. In particular, $N_{(v_i+1)(v_{i+1}-1)} = U_{[v_i+1, v_{i+1}-1]}$. As $D(X_j) \in S$, $D(X_j)$ is not contained inside the interval $[(v_i + 1)(v_{i+1} - 1)]$ and $A_{j(v_i+1)(v_{i+1}-1)}$ is unset. The constraint

$$N_{(v_i+1)(v_{i+1}-1)} = \sum_{i=1}^{n} A_{i(v_i+1)(v_{i+1}-1)}$$

is tight because $N_{(v_i+1)(v_{i+1}-1)} = U_{[v_i+1,v_{i+1}-1]}$. Therefore all unset variables, including $A_{j(v_i+1)(v_{i+1}-1)}$ are fixed to 0 and (6.27) prunes v from $D(X_j)$.

Complexity argument: There are $O(nd^2)$ constraints (6.27) that can be woken $O(d)$ times down the branch of the search tree. Each invocation costs either $O(1)$ if domain of a variable X_i is changed or $O(d)$ if A_{ilu} is assigned. This amounts to the total of $O(nd^3)$ down the branch of the search tree.

There are $O(d^2)$ constraints (6.28), which can be woken $O(n)$ times each down the branch of the search tree for a total cost of $O(n)$ time down the branch. Thus a total of $O(nd^2)$ down the branch of the search tree. There are $O(d^2)$ constraints (6.29) that can be woken $O(n)$ times down the branch. Each propagation takes $O(1)$ time to execute for a total of $O(nd^2)$ time down the branch. The final complexity down the branch of the search tree is therefore $O(nd^3) + O(nd^2) + O(nd^2) = O(nd^3)$. $\diamond$

Example 6.15 *Consider how the decomposition* (6.27)–(6.29) *works on our running example. We recall that initial domains of the variables and upper and lower bounds on the occurrences of values are:*

v	1	2	3	4	5
X_1	$*$				
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$		
X_4	$*$	$*$	$*$	$*$	$*$
X_5	$*$	$*$	$*$	$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	1	5	5

Consider the interval $[3, 3]$. *This is an extended Hall interval:*

$$N_{3,3} = A_{133} + A_{233} + A_{333} + A_{433} + A_{533}$$

and $N_{3,3} = [0, 1]$, $A_{333} = 1$. *This makes the constraint tight and bounds propagation fixes* $A_{133} = A_{233} = A_{433} = A_{533} = 0$. *By channelling constraints value* 3 *is pruned from variables* X_2, X_4 *and* X_5.

v	1	2	3	4	5
X_1	$*$				
X_2	$*$	$*$		$*$	$*$
X_3			$*$		
X_4	$*$	$*$		$*$	$*$
X_5	$*$	$*$		$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	1	5	5

Consider the set $S = \{2, 4, 5\}$. This is an unstable set (see Example 6.14).

Following the proof of Theorem 6.22 we consider a set of intervals $S' = \{[1, 1], [3, 3]\}$ and perform the chaining lower bounds procedure taking into account that $lb(N_{11}) = lb(N_{22}) = lb(N_{33}) = lb(N_{44}) = lb(N_{55}) = 1$:

$$lb(N_{12}) \geq lb(N_{11}) + lb(N_{22}) \mid\Rightarrow lb(N_{12}) = 2$$
$$lb(N_{13}) \geq lb(N_{12}) + lb(N_{33}) \mid\Rightarrow lb(N_{13}) = 3$$
$$lb(N_{14}) \geq lb(N_{13}) + lb(N_{44}) \mid\Rightarrow lb(N_{14}) = 4$$
$$lb(N_{15}) \geq lb(N_{14}) + lb(N_{55}) \mid\Rightarrow lb(N_{15}) = 5$$

Now we have $lb(N_{15}) = ub(N_{15}) = 5$. We apply the chaining upper bounds procedure:

$$ub(N_{15}) = lb(N_{15}) = lb(N_{14}) + lb(N_{55}) \mid\Rightarrow N_{14} = 4; N_{55} = 1$$
$$\vdots$$
$$ub(N_{12}) = lb(N_{12}) \geq lb(N_{11}) + lb(N_{22}) \mid\Rightarrow N_{11} = N_{22} = 1$$

Hence, all variables N_{ll}, $l = 1, \ldots, 5$ are fixed to 1. By constraints (6.27)*, $A_{111} = A_{333} = 1$, so by constraints* (6.28)*, $A_{i11} = 0, i, \in [2, \ldots, 5]$ and $A_{i33} = 0, i, \in \{1, 2, 4, 5\}$. By constraints* (6.27)*, we remove 1 and 3 from X_2, X_4, X_5. This makes the constraint range consistent .*

v	1	2	3	4	5
X_1	$*$				
X_2		$*$		$*$	$*$
X_3			$*$		
X_4		$*$		$*$	$*$
X_5		$*$		$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	1	5	5

$\diamond$

Bounds consistency. We show that using our reformulation we can enforce *BC* of the GCC constraint. As in the case of ALL-DIFFERENT, by replacing constraints (6.27) by constraints (6.3) and (6.4), the decomposition achieves *BC*.

Theorem 6.23 *Enforcing bounds consistency on constraints* (6.3), (6.4), (6.28)–(6.29) *achieves bounds consistency on the corresponding* GCC *constraint in* $O(nd^2)$ *time down any branch of the search tree.*

Proof: The proof follows that for Theorem 6.22 except that fixing $A_{ilu} = 0$ prunes the bounds of $D(X_i)$ if and only if exactly one bound of the domain of X_i intersects the interval $[l, u]$. The complexity reduces to $O(nd^2)$ as bounds consistency on (6.3) and (6.4) is in $O(nd^2)$ (see Theorem (6.5)) and bounds consistency on (6.28) and (6.29) is in $O(nd^2)$ (see Theorem 6.22). $\diamond$

The best known algorithm for bounds consistency on GCC runs in $O(n)$ time at each call [QLOvBG04] and can be awaken $O(nd)$ times down a branch of the search tree . This gives a total of $O(n^2d)$, which is greater than the $O(nd^2)$ here when $n > d$.

Example 6.16 *Consider how the BC decomposition works on our running example. We recall that initial domains of the variables and upper and lower bounds on the occurrences*

of values are:

v	1	2	3	4	5
X_1	$*$				
X_2	$*$	$*$	$*$	$*$	$*$
X_3			$*$		
X_4	$*$	$*$	$*$	$*$	$*$
X_5	$*$	$*$	$*$	$*$	$*$
l_v	1	1	0	1	1
u_v	5	5	1	5	5

Consider the set $S = \{1, 2, 4, 5\}$. *This is an unstable set:*

$$I_S = \sum_{i \in S} l_i = 4$$

Moreover, we have that $A_{333} = 1$, *which forces* $lb(N_{3,3}) = 1$ *by constraint* (6.28). *Following the proof of Theorem 6.22 we consider a set of intervals* $S' = \{[3, 3]\}$ *and perform the chaining lower bounds procedure taking into account that* $lb(N_{11}) = lb(N_{22}) = lb(N_{33}) = lb(N_{44}) = lb(N_{55}) = 1$ *(similar to Example 6.15). This sets lower bounds of variables* N_{ll}, $l = 1, \ldots, 5$ *to* 1. *Then we apply the chaining upper bounds procedure (similar to Example 6.15), which fixes* N_{ll}, $l = 1, \ldots, 5$ *to 1. Finally, by constraints* (6.27), $A_{111} = A_{333} = 1$, *so by constraints* (6.28), $A_{i11} = 0, i, \in 2..5$ *and* $A_{i33} = 0, i, \in \{1, 2, 4, 5\}$. *By constraints* (6.3) *and* (6.4), *we remove 1 from* X_2, X_4, X_5. *Note that we cannot remove the value* 3 *as it is not a bound of any variable. This makes the constraint bounds consistent .*$\diamond$

6.8.3 Other decompositions of the GCC constraint

In this section we consider three decompositions of the GCC constraint. The first decomposition is a naive decomposition into a set of AMONG constraints [Lau78]. This decomposition does not enforce bounds consistency or range consistency but it can be also easily implemented in a constraint solver. We present it for completeness of the material. The second is a decompositions that was recently proposed in [FS09]. It was shown that this decomposition is useful in practice. We prove here that this decomposition does not detect bounds disentailment (Section 6.8.3.2).

6.8.3.1 Decomposition into AMONG constraint

A natural way to decompose the $\text{GCC}([X_1, \ldots, X_n], [l_1, \ldots, l_d], [u_1, \ldots, u_d])$ constraint is to encode it using a set of AMONG constraints.

$$\text{AMONG}([X_1, \ldots, X_n], l_v, u_v, \{v\}) \quad v = 1, \ldots, d; \tag{6.34}$$

Clearly, constraints (6.34) are logically equivalent to the GCC constraint. This decomposition provides a weak pruning as the following theorem shows:

Theorem 6.24 *Enforcing domain consistency on the decomposition* (6.34) *does not detect bounds disentailment for the* GCC *constraint.*

Proof: Consider $\text{GCC}([X_1, X_2, X_3], [2, 2], [2, 2])$ with the following domains $D(X_1) = D(X_2) = D(X_3) = \{1, 2\}$. Enforcing domain consistency on each $\text{AMONG}([X_1, X_2, X_3], 2, 2, \{1\})$ and $\text{AMONG}([X_1, X_2, X_3], 2, 2, \{2\})$ constraint does not cause any pruning because any of three variables that can potentially take value 1 to satisfy the first AMONG constraint and value 2 to satisfy the second AMONG. However, the GCC constraint is bounds disentailed. $\diamond$

6.8.3.2 Decomposition using linear encoding of domains

The next decomposition that we consider is a decomposition proposed in [FS09]. The decomposition uses cardinality variables instead of cardinality bounds values $[l_1, \ldots, l_d], [u_1, \ldots, u_d]$. However, we replace the cardinality variables with the cardinality values here to keep all decompositions consistent.

To construct a decomposition we introduce $d + 1$ cumulative sum variables $A_0, \ldots, A_d$ such that $A_v = \sum_{i=1}^{n} B_{iv}$, where B_{iv} is defined according to equation (6.3).

We then post the following constraints for $1 \leq i \leq n$, $1 \leq v \leq d$,

$$c_{iv} \iff (X_i == v) \tag{6.35}$$

$$B_{iv} \iff X_i \leq v \tag{6.36}$$

$$A_0 = 0 \tag{6.37}$$

$$A_v = \sum_{i=1}^{n} B_{iv} \tag{6.38}$$

$$l_v \leq \sum_{i=1}^{n} c_{iv} \leq u_v \tag{6.39}$$

$$l_v \leq A_v - A_{v-1} \leq u_v \tag{6.40}$$

$$\sum_{v=1}^{n} l_v \leq A_d - A_0 \leq \sum_{v=1}^{n} u_v \tag{6.41}$$

Theorem 6.25 *Enforcing bounds consistency on the decomposition* (6.35)–(6.41) *does not detect bounds disentailment for the* GCC *constraint.*

Proof: Consider GCC$([X_1, X_2, X_3, X_4, X_5], [1, 0, 0, 1], [2, 1, 1, 2])$ with the following domains $D(X_1) = D(X_2) = [1, 4]$ and $D(X_3) = D(X_4) = D(X_5) = [2, 3]$. Initial domains of variables: $D(A_1) = [0, 2]$ $D(A_2) = [0, 5]$ $D(A_3) = [3, 5]$ and $D(A_4) = [5]$.

$$\begin{aligned} A_4 - A_3 &\geq 1 \mid\Rightarrow D(A_3) = [3, 4] \\ A_3 - A_2 &\geq 0 \mid\Rightarrow D(A_2) = [0, 4] \\ A_3 - A_2 &\leq 1 \mid\Rightarrow D(A_2) = [2, 4] \\ A_2 - A_1 &\leq 1 \mid\Rightarrow D(A_2) = [2, 3] \\ A_1 - A_0 &\geq 1 \mid\Rightarrow D(A_1) = [1, 2] \end{aligned}$$

So after propagation reaches fix point we have $D(A_1) = [1, 2]$ $D(A_2) = [2, 3]$ $D(A_3) = [3, 4]$, $D(A_4) = [5]$ and $D(c_{iv}) = [0, 1]$, $1 \leq i \leq n$, $1 \leq v \leq d$. However, enforcing bounds consistency detects a failure. $\diamond$

6.8.4 Other related work

The global cardinality constraint was first introduced in the CHARME constraint programming language [OMT89]. Regin proposed a domain consistency propagator based on network flow that runs in $O(n^2)$ time [Reg99]. Katriel and Thiel proposed a bounds consistency propagator for the extended form of the GCC constraint that has variables representing the cardinalities [KT03]. Quimper *et al.* proved that enforcing domain consistency on such an extended GCC constraint is NP-hard [QLOvBG04]. They also improved the time complexity to enforce domain consistency and gave the first propagator for enforcing range consistency on GCC .

6.9 The NVALUE constraint

Another generalisation of the ALL-DIFFERENT constraint is the NVALUE$([X_1, \ldots, X_n], N)$ constraint . NVALUE counts the number of values used by a set of variables. Pachet and Roy proposed the global NVALUE constraint to model a combinatorial problem for selecting musical play-lists [PR99]. From a theoretical perspective, the NVALUE constraint is similar to the OVERLAPPINGALLDIFF constraint

and is more difficult to propagate than the ALL-DIFFERENT and GCC constraints since enforcing domain consistency on it is known to be NP-hard [BHH+06a]. Moreover, as NVALUE is a generalisation of ALL-DIFFERENT, there exists no polynomial sized decomposition of NVALUE which achieves domain consistency (Chapter 7).

Nevertheless, we show that decomposition can simulate the polynomial time algorithm for enforcing bounds consistency on NVALUE. Our range consistency propagator has the same worst case complexity as the best known bounds consistency propagator. This contrasts with the ALL-DIFFERENT constraint where the best known bounds consistency propagator has better worst case complexity to the best known range consistency propagator.

Example 6.17 (Running example (NVALUE)) *Beldiceanu et al. give an interesting application of the* NVALUE *constraint to encode a resource allocation problem for virtual machine management on a cluster of computers [BHLP10]. The problem consists of assigning virtual machines to nodes of the cluster subject to two types of constraints:*

- *each machine has CPU time or memory resource demand and*
- *there are restrictions on the number of nodes that can be used by virtual machines.*

The second restriction is modelled using the NVALUE *constraint [BHLP10].*

Suppose we have 6 *virtual machines and nodes of the cluster have different characteristics. Therefore, each virtual machine can be run on nodes that satisfy its resource demand constraints. We introduce one variable for each virtual machine,* $X_1, \ldots, X_6$*. Domains of the variables show compatibility between virtual machines and nodes of the cluster. We also introduce a variable* N *that represents restrictions on the number of nodes that we can use. The following matrix shows domains of the variables.*

	1	2	3	4	5	6
X_1	$*$	$*$	$*$	$*$	$*$	$*$
X_2		$*$				
X_3		$*$				
X_4		$*$	$*$	$*$		
X_5				$*$	$*$	
X_6				$*$	$*$	
N	$*$	$*$	$*$	$*$	$*$	$*$

In our example N *is unrestricted. We will vary this parameter in the following sections to demonstrate our filtering algorithms and decompositions.* $\diamond$

6.9.1 Filtering algorithms for the NVALUE constraint

The NVALUE constraint can be seen as a conjunction of two constraints : an ATMOSTNVALUE constraint and an ATLEASTNVALUE constraint .

Definition 6.11 *The* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is satisfied if and only if* $N \leq |\{X_i | 1 \leq i \leq n\}|$.

Definition 6.12 *The* ATMOSTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is satisfied if and only if* $|\{X_i | 1 \leq i \leq n\}| \leq N$.

We show that decomposing the NVALUE constraint into these two parts does not hinder propagation in general if we enforce range consistency or bounds consistency. Note that it has been shown that this decomposition hinders propagation if we enforce domain consistency [BHH$^+$06a].

In both these cases we are only looking for a bound support, therefore, we assume that domains of all variables are intervals. We also use the following notations. Given an assignment X of values, $card(X)$ denotes the number of distinct values in X. Given a vector of variables $X = [X_1 \ldots X_n]$, $card_{\uparrow}(X)$ is the maximum cardinality assignment and $card_{\downarrow}(X)$ is the minimum cardinality assignment.

Theorem 6.26 (adapted from Lemma 1 [BHH$^+$06a]) *Consider the* NVALUE$([X_1, \ldots, X_n], N)$ *constraint. If* $D(N) \subseteq [card_{\downarrow}(X), card_{\uparrow}(X)]$*, then the bounds of* N *have bound supports.*

Proof: Let X_{min} be a minimum cardinality assignment of X with $card(X_{min}) = card_{\downarrow}(X)$ and X_{max} be a maximum cardinality assignment of X with $card(X_{max}) = card_{\uparrow}(X)$. Consider the sequence $X_{min} = X^0, X^1, \ldots, X^n = X_{max}$ where X_{k+1} is the same as X^k except that X^{k+1} has been assigned its value in X_{max} instead of its value in X_{min}. $|card(X^{k+1}) - card(X^k)| \leq 1$ because they only differ on X_{k+1}. Hence, for any $p \in [card_{\downarrow}(X), card_{\uparrow}(X)]$, there exists $k \in [1, \ldots, n]$ with $card(X^k) = p$. Thus, X^k with $card(X^k) = p$ is a bound support for p on NVALUE$([X_1, \ldots, X_n], N)$. Therefore, $lb(N)$ and $ub(N)$ have a bound support. $\diamond$

We now prove that decomposing the NVALUE constraint into ATMOSTNVALUE and ATLEASTNVALUE constraints does not hinder pruning when enforcing range consistency or bounds consistency.

Theorem 6.27 *Enforcing bounds consistency on the variable N for* NVALUE$([X_1, \ldots, X_n], N)$ *is equivalent to enforcing bounds consistency on the variable N for* ATMOSTNVALUE$([X_1, \ldots, X_n], N)$ *and for* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$.

Proof: Suppose the ATMOSTNVALUE and ATLEASTNVALUE constraints are bounds consistent on variable N. The ATMOSTNVALUE constraint guarantees that $card_{\downarrow}(X) \leq lb(N)$ and the ATLEASTNVALUE constraint guarantees that $card_{\uparrow}(X) \geq ub(N)$. Therefore, $D(N) \in [card_{\downarrow}(X), card_{\uparrow}(X)]$. By Theorem 6.26, the variable N is bounds consistent . $\diamond$

Theorem 6.28 *Suppose* NVALUE$([X_1, \ldots, X_n], N)$ *is bounds consistent on the variable N. Then enforcing range consistency on variables X for* NVALUE$([X_1, \ldots, X_n], N)$ *is equivalent to enforcing range consistency on variables X for* ATMOSTNVALUE$([X_1, \ldots, X_n], N)$ *and for* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$.

Proof: As NVALUE is bounds consistent on the variable N, ATMOSTNVALUE and ATLEASTNVALUE are bounds consistent on this variable by Theorem 6.27.

Suppose the ATMOSTNVALUE and ATLEASTNVALUE constraints are range consistent on variables X. Consider a variable-value pair $X_i = b$. Let X_L be a bound support of $X_i = b$ in the ATLEASTNVALUE constraint so that there exists a value $p_1 \in D(N)$ such that $card(X_L) \geq p_1$. Let X_M be a bound support of $X_i = b$ in the ATMOSTNVALUE constraint so that there exists a value $p_2 \in D(N)$ such that $card(X_M) \leq p_2$. Consider the sequence $X_L = X^0, X^1, \ldots, X^n = X_M$ where X^{k+1} is the same as X^k except that X_{k+1} has been assigned its value in X_M instead of its value in X_L. $|card(X^{k+1}) - card(X^k)| \leq 1$ because they only differ on X_{k+1}. Hence, there exists $k \in [1, \ldots, n]$ with $min(p_1, p_2) \leq card(X^k) \leq max(p_1, p_2)$. We know that p_1 and p_2 belong to $D(N)$ because they belong to bound supports. Thus, $card(X^k) \in D(N)$ and $card(X^k)$ is a bound support for $X_i = b$ on NVALUE$([X_1, \ldots, X_n], N)$. $\diamond$

Theorem 6.29 *Suppose* NVALUE$([X_1, \ldots, X_n], N)$ *is bounds consistent on the variable N. Then enforcing bounds consistency on variables X for* NVALUE$([X_1, \ldots, X_n], N)$ *is equivalent to enforcing bounds consistency on variables X for* ATMOSTNVALUE$([X_1, \ldots, X_n], N)$ *and for* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$.

Proof: Similar to Theorem 6.28. $\diamond$

When enforcing domain consistency, Bessiere *et al.* [BHH$^+$06a] noted that decomposing the NVALUE constraint into ATMOSTNVALUE and ATLEASTNVALUE constraints

does hinder propagation, but only when $D(N)$ contains just $card_{\downarrow}(X)$ and $card_{\uparrow}(X)$ and there is a gap in the domain in-between (see Theorem 1 in [BHH$^+$06a] and the discussion that follows). When enforcing bounds consistency or range consistency, any such gap in the domain for N is ignored and it is not surprising that the decomposition does not hinder propagation.

A number of polynomial propagation algorithms have been proposed that achieve bounds consistency and some closely related levels of local consistency [Bel01, BHH$^+$05, BHH$^+$06a]. In this chapter we consider a bounds consistency filtering algorithm for the ATMOSTNVALUE that was proposed in [Bel01] and a domain consistency filtering algorithm for the ATLEASTNVALUE that was proposed in [ZMP06], because our bounds consistency propagator is based on this algorithm. We use the theoretical results that these algorithms are based on in the following sections.

6.9.1.1 Filtering algorithm for the ATMOSTNVALUE constraint.

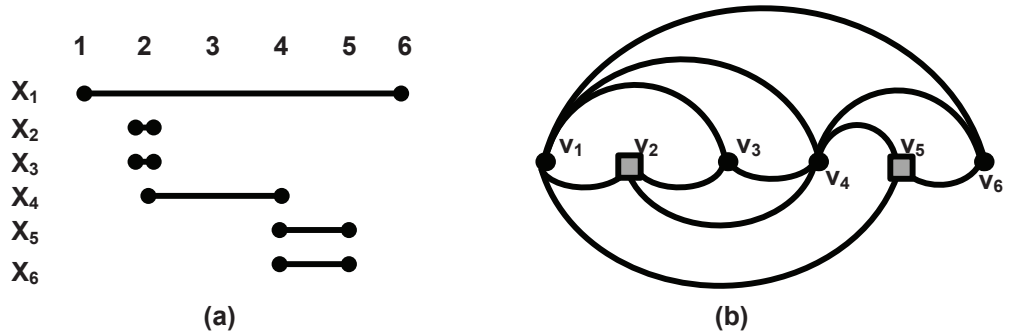
First we consider the ATMOSTNVALUE constraint . A bounds consistency algorithm for this constraint was proposed in [Bel01] and the proof that it enforces bounds consistency on the variable N was given [BHH$^+$05]. The algorithm is based on counting of the number of cliques in the interval graph formed by domains of variables that are also intervals. The number of cliques in the interval graph equals to the size of the maximum independent set. In order to explain the algorithm we recall the notions of interval graph and show how to construct this graph given variable domains.

We consider the construction of the interval graph from domains of variables on an example.

Example 6.18 *We recall domains of the X variables and suppose that the domain of the variable N is $[1, 2]$:*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N	*	*				

Figure 6.2 A set of intervals that correspond to domains of variables X (a) and an interval graph(b).



We construct the interval graph as follows. Let S be a set of intervals formed by domains of variables so that $S = \{D(X_1), \ldots, D(X_n)\}$. We introduce one vertex for each interval $\{v_1, \ldots, v_n\}$ and an edge (v_i, v_j) if and only if $D(X_i) \cap D(X_j) \neq \emptyset$. The set of intervals that corresponds to the domains of the variables is $S = \{[1, 6], [2, 2], [2, 2], [2, 4], [4, 5], [4, 5]\}$. Figure 6.2 shows an interval representation of domains. To construct an interval graph we introduce one vertex for each interval $v_1, \ldots, v_6$ and we connect v_i and v_j if their domains of the corresponding variables overlap. Figure 6.2 shows the corresponding interval graph.

$\diamond$

We recall that we denote $\alpha_I(G)$ (Section 2.1.3) the size of the maximum independent set. A maximum independent set is highlighted as gray squares on Figure 6.2(b) for the running example.

The bounds consistency algorithm proposed in [Bel01] sorts variables by their upper bounds. Then it process them in this order and outputs a sequence of maximal cliques, $C_1, \ldots, C_m$. The number of these cliques is the size of maximum independent set [BHH$^+$06a].

Proposition 6.1 (Proposition 1 [BHH$^+$06a]) *Let $\{C_1, ..., C_k\}$ be a partition of the intervals into maximal cliques. If $C = \{I_1, \ldots, I_p\}$ is the vector of intervals where I_j is the element of C_j with least maximum value, then all such elements have empty pairwise intersections.*

Note that not only elements of the cliques with least maximum values can form a pairwise independent set of intervals. Consider, for example, the ATMOSTNVALUE constraint with 3 variables $D(X_1) = [1, 2]$, $D(X_2) = [2, 3]$ and $D(X_4) = [5, 6]$. There are two maximal cliques. The first contains intervals $[1, 2]$ and $[2, 3]$. The second contains the interval

$[5,6]$. The sets $\{[1,2],[5,6]\}$ and $\{[2,3],[5,6]\}$ are two sets of disjoint intervals.

Definition 6.13 *Let $\mathcal{C}_j = \cap_{i=1}^{p} I_{i_j}$ be the intersection of intervals in the jth clique. We call $\mathcal{C}_j$ the jth intersection interval. We call set $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ maximum cardinality set of intersection intervals.*

Theorem 6.30 (Range/Bounds disentailment) *The* ATMOSTNVALUE $([X_1, \ldots, X_n], N)$ *constraint is satisfiable if and only if*

1. $D(X_i) \neq \emptyset$, $i = 1, \ldots, n$ *and*
2. $\alpha_I(G) \leq ub(N)$.

Theorem 6.31 (Bounds consistency on N) *The* ATMOSTNVALUE $([X_1, \ldots, X_n], N)$ *constraint is bounds consistent on N iff*

1. *Theorem 6.30 conditions 1–2 hold and*
2. $\alpha_I(G) \leq lb(N)$.

Theorem 6.32 (adapted from Lemma 2 [BHH+06a]) *Consider* ATMOSTNVALUE $([X_1, \ldots, X_n], N)$. *If N is bounds consistent and $lb(N) < ub(N)$, then X are range consistent .*

Proof: Take any assignment X_{min} such that $card(X_{min}) = card_{\downarrow}(X)$. Let X_b be the assignment where the value of X_i in X_{min} has been replaced by b, $b \in D(X_i)$. We know that $card(X_b) \in [card(X_{min}) - 1, card(X_{min}) + 1] = [card_{\downarrow}(X) - 1, card_{\downarrow}(X) + 1]$ because only one variable has been flipped. As $ub(N) > lb(N)$ and $lb(N) \geq \alpha_I(G) = card_{\downarrow}(X)$ then $card_{\downarrow}(X) + 1 \leq ub(N)$. Therefore, $X_i = b$ has a support. $\diamond$

Theorem 6.33 (adapted from Lemma 2 [BHH+06a]) *Consider* ATMOSTNVALUE $([X_1, \ldots, X_n], N)$. *If N is bounds consistent and $lb(N) < ub(N)$, then X are bounds consistent .*

Proof: Similar to Theorem 6.32. $\diamond$

Next we provide necessary and sufficient conditions to enforce range consistency and bounds consistency.

Theorem 6.34 (Range consistency on X) *The* ATMOSTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is range consistent on variables X if and only if*

1. *Theorems 6.30 and 6.31 conditions 1–2 hold and*

2. *(a)* $lb(N) < ub(N)$ *or*

   (b) $\alpha_I(G) = lb(N)$, $lb(N) = ub(N)$ *and* $D(X_i) \in \mathcal{C}$, $i = 1 \ldots, n$,

where $\mathcal{C}$ *is the maximum cardinality set of intersection intervals.*

Proof: Case 2a follows from Theorem 6.32. Case 2b is when the value N is fixed. As $\alpha_I(G) = lb(N) = ub(N)$ then there are at least N maximal cliques. We know that at least one value from each $\mathcal{C}_i$ has to appear in a solution. Hence, N values, one from each $\mathcal{C}_i$, $i = 1, \ldots, N$, are taken in any assignment. Therefore none of the values outside the $\cup_{i=1}^{N}\mathcal{C}_i$ can be taken by a variable X_i, $i = 1, \ldots, n$. On the other hand, any value inside $\mathcal{C}_i$ can be taken by a variable X as it belong to ith interval of any maximum cardinality set of disjoint intervals by construction of the set $\mathcal{C}$. $\diamond$

Theorem 6.35 (Bounds consistency on X) *The* ATMOSTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is bounds consistent on variables* X *if and only if*

1. *Theorems 6.30 and 6.31 conditions 1–2 hold and*

2. *(a)* $lb(N) < ub(N)$ *or*

   (b) $\alpha_I(G) = lb(N)$, $lb(N) = ub(N)$ *and* $\{lb(D(X_i)), ub(D(X_i)\} \in \mathcal{C}$, $i = 1 \ldots, n$,

where $\mathcal{C}$ *is the maximum cardinality set of intersection intervals.*

Proof: Similar to Theorem 6.34. $\diamond$

We also observe that changing the domains of the X variables cannot affect the upper bound of N by the ATMOSTNVALUE constraint and, conversely, changing the lower bound of N cannot affect the domains of the X variables.

Let us consider how to achieve range consistency and bounds consistency on our running example.

Example 6.19 *First we check disentailment of the constraint. By Theorem 6.30 we find a set of maximal cliques. In our example, there are two maximal cliques* C_1 *and* C_2. C_1 *contains a set of intervals that correspond to domains of variables* X_1, X_2, X_3 *and* X_4. C_2 *contains domains of variables* X_5 *and* X_6. *The size of a maximum independent set is two, which is equal to* $ub(N)$. *Hence, the constraint is satisfiable.*

Bounds consistency on N**.** *The lower bound of* N *is* 1 *which is smaller than the size of the maximum independent set. Hence, we remove* 1 *from* $D(N)$. *This fixes* N *to 2.*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N		*				

Bounds consistency on X**.** *Now we consider the maximum cardinality set of intersection intervals* $\mathcal{C}$*. As we have a single maximum set of disjoint intervals* $\{[2,2],[4,5]\}$ *then* $\mathcal{C}_1 = \{[2,2]\}$*,* $\mathcal{C}_2 = \{[4,5]\}$ *and* $\mathcal{C} = \{[2,2],[4,5]\}$*. Therefore, values* 1 *and* 6 *have to be pruned from* $D(X_1)$ *as* $\{lb(X_1), ub(X_1)\} \notin \{2,4,5\}$*.*

	1	2	3	4	5	6
X_1		*	*	*	*	
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N		*				

Range consistency on X**.** *To achieve range consistency we also need to prune the value* 3 *from domains of variables* X_1 *and* X_3*, as this value does not belong to any of the intervals in* $\mathcal{C}$*.*

	1	2	3	4	5	6
X_1		*		*	*	
X_2		*				
X_3		*				
X_4		*		*		
X_5				*	*	
X_6				*	*	
N		*				

6.9.1.2 Filtering algorithm for the ATLEASTNVALUE constraint.

Next we consider the ATLEASTNVALUE constraint. This constraint can be encoded as the soft ALL-DIFFERENT constraint with the variable-based violation measure [ZMP06] .

Definition 6.14 *The soft* ALL-DIFFERENT$([X_1, \ldots, X_n], Z)$ *constraint with variable-based violation measure holds if and only if* $n - \sum_{v=1}^{d} \max(|\{X_i = v \mid 1 \leq i \leq n\}| - 1, 0) \leq Z$.

Then the ATLEASTNVALUE$([X_1, \ldots, X_n], N)$ constraint is equivalent to ALL-DIFFERENT$([X_1, \ldots, X_n], Z)$ constraint, $Z = n - N$, because the variable N shows the number of distinct values in an assignment and the variable Z shows the total number of repetitions in an assignment.

A domain consistency filtering algorithm for the soft ALL-DIFFERENT constraint was proposed in [ZMP06]. The algorithm is based on the correspondence between solutions of the ALL-DIFFERENT constraint and maximum matching in the variable-value graph, G_v (Section 2.4.1). We denote a maximum matching in the variable-value graph $M(G)$.

Consider an encoding of the ATLEASTNVALUE constraint into the ALL-DIFFERENT constraint and a construction of the variable-value graph on the running example.

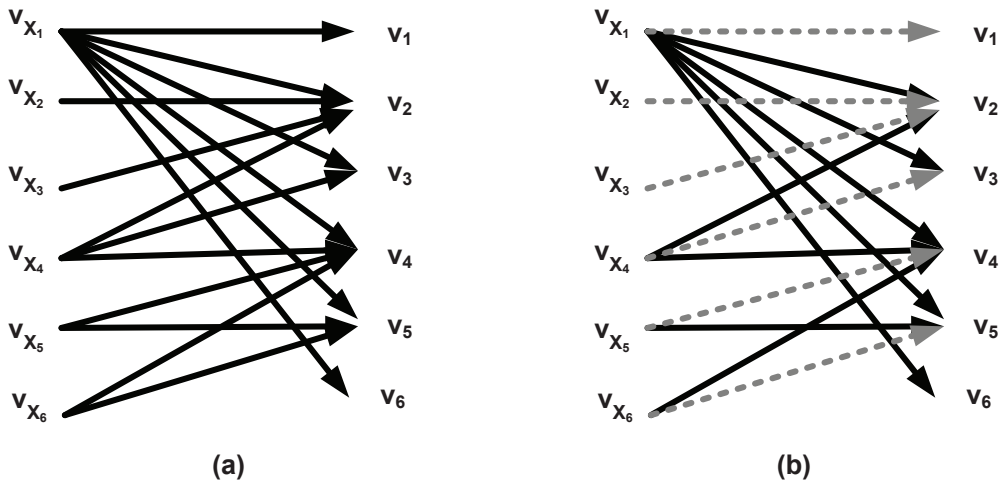
Example 6.20 *We recall domains of the variables* X *(Example 6.17) and suppose that the domain of the variable* N *is* $[5, 6]$*:*

	1	2	3	4	5	6
X_1	$*$	$*$	$*$	$*$	$*$	$*$
X_2		$*$				
X_3		$*$				
X_4		$*$	$*$	$*$		
X_5				$*$	$*$	
X_6				$*$	$*$	
N					$*$	$*$

An equivalent soft ALL-DIFFERENT *constraint has the same variables* X *in the scope and the domain of the variable* Z *is* $[0, 1]$.

Next we construct a variable-value graph. We introduce one vertex for a variable, v_{X_i}*,* $i = 1, \ldots, 6$ *and one vertex for a value,* v_j*,* $j = 1, \ldots, 6$*. We add an edge from* v_{X_i} *to* v_j *if and only if* $v_j \in D(X_i)$ *(Figure 6.3 (a)). The size of a maximum matching is* 5 *(Figure 6.3 (b)) (gray lines).* $\diamond$

Figure 6.3 (a) Variable-value graph that corresponds to the soft ALL-DIFFERENT constraint, (b) a maximum matching in the variable-value graph (gray dashed lines)



A domain consistency filtering algorithm for the soft ALL-DIFFERENT constraint along with necessary and sufficient conditions for achieving domain consistency is described in [ZMP06]. Due to equivalence of the soft ALL-DIFFERENT constraint and ATLEASTNVALUE we adjust these conditions to the interval domain variables:

Theorem 6.36 (Range/Bounds disentailment. Follows from Theorem 7 [ZMP06]) *The* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is satisfiable if and only if*

1. $D(X_i) \neq \emptyset$, $i = 1, \ldots, n$ *and*
2. $|M(G_v)| \geq lb(N)$.

Theorem 6.37 (Bounds consistency on N.) *The* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$ *constraint enforces bounds consistency on the variable N if and only if*

1. *Theorem 6.36 conditions 1–2 hold and*
2. $|M(G_v)| \geq ub(N)$.

Theorem 6.38 (adapted from Lemma 2 [BHH$^+$06a]) *Consider* ATLEASTNVALUE $([X_1, \ldots, X_n], N)$. *If N is bounds consistent and $lb(N) < ub(N)$, then X are range consistent .*

Proof: Take any assignment X_{max} such that $card(X_{max}) = card_{\uparrow}(X)$. Let X_b be the assignment where the value of X_i in X_{max} has been replaced by b, $b \in D(X_i)$. We know that $card(X_b) \in [card(X_{max}) - 1, card(X_{max}) + 1] = [card_{\uparrow}(X) - 1, card_{\uparrow}(X) + 1]$

because only one variable has been flipped. As N is bounds consistent then $ub(N) \leq card_{\uparrow}(X)$ and $lb(N) < card_{\uparrow}(X)$. Hence, $lb(N) \leq card(X_b)$. and the variable-value pair $X_i = b$ has a support. $\diamond$

Theorem 6.39 *Consider* ATLEASTNVALUE $([X_1, \ldots, X_n], N)$. *If N is bounds consistent and $lb(N) < ub(N)$, then X are bounds consistent .*

Proof: Similar to Theorem 6.38. $\diamond$

Theorem 6.40 (Range consistency. Follows from Theorem 7 [ZMP06]) *The* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is range consistent on variables X if and only if*

1. *Theorems 6.36 and 6.37 conditions 1–2 hold and*
   (a) $lb(N) < ub(N)$ *or*
   (b) $lb(N) = ub(N)$, $ub(N) = |M(G_v)|$ *and for each variable-value pair $X_i = v_j$, $i = 1, \ldots, n$, $j = 1 \ldots, d$, the corresponding edge (v_{X_i}, v_j) can be extended to a matching of size at least $lb(N)$.*

Theorem 6.41 (Bounds consistency. Follows from Theorem 7 [ZMP06])

The ATLEASTNVALUE$([X_1, \ldots, X_n], N)$ *constraint is bounds consistent on variables X if and only if*

1. *Theorems 6.36 and 6.37 conditions 1–2 hold and*
   (a) $lb(N) < ub(N)$ *or*
   (b) $lb(N) = ub(N)$, $ub(N) = |M(G_v)|$ *and for each variable-bound pair $X_i = lb(D(X_i))$/$X_i = ub(D(X_i))$, $i = 1, \ldots, n$, $j = 1 \ldots, d$, the corresponding edge $(v_{X_i}, lb(D(X_i)))$/$(v_{X_i}, ub(D(X_i)))$ can be extended to a matching of size at least $lb(N)$.*

We also observe that changing the domains of the variables X cannot affect the lower bound of N by the ATLEASTNVALUE constraint and, conversely, changing the upper bound of N cannot affect the domains of X.

Consider how to achieve range consistency and bounds consistency on our running example.

Example 6.21 *First we check bounds disentailment. Example 6.20 shows that the size of the maximum matching in the graph G_v is 5, which is greater than $lb(N)$. By Theorem 6.36 the constraint is satisfiable.*

Bounds consistency on N. *As the size of the maximum matching is 5, the value 6 has to be removed from $D(N)$ by Theorem 6.37. This fixes N to 5.*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N					*	

Bounds consistency on X. *Note that variables X_2, X_3 and X_5,X_6 can take only three distinct values, $D(X_i) \subseteq \{2,4,5\}$, $i = \{2,3,5,6\}$. As $N = 5$, X_4 cannot take values in the set $\{2,4\}$. If X_4 takes 2 or 4, then the size of the maximum matching will be 4.*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4			*			
X_5				*	*	
X_6				*	*	
N					*	

Range consistency on X.

We note that variables $X_2, \ldots, X_6$ can take only four distinct values, $D(X_i) \in \{2,3,4,5\}$, $i = \{2, \ldots, 6\}$. As $N = 5$, these value have to be pruned from the domain

of variable X_1.

	1	2	3	4	5	6
X_1	*					*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N					*	

◇

6.9.1.3 Hall theorem for the soft ALL-DIFFERENT constraint

Before we present our decompositions for the ATLEASTNVALUE constraint we derive necessary and sufficient conditions for the existence of the maximum matching in a convex graph that corresponds to the soft ALL-DIFFERENT constraint. We use these conditions in Section 6.9.2.3 where we present a reformulation of the ATLEASTNVALUE constraint into a set of primitive constraints.

Consider the soft ALL-DIFFERENT$([X_1, \ldots, X_n], N)$ constraint and the corresponding variable-value graph $G(V, E)$, $V = A \cup B$, $|A| = n$, $|B| = d$. We denote vertices in the partition A as $X_1, \ldots, X_n$ and vertices in the partition B as $1, \ldots, n$. We refer to vertices in the partition A as variables and vertices in the partition B as values of the interval $[1, d]$. Clearly, the neighbourhood of $N(X_i)$ is $D(X_i)$.

Suppose the graph G does not have a matching of size n. Then there exists a violated Hall interval. To determine the size of the maximum matching we need to distinguish between violated Hall intervals that contain and do not contain a violated unstable set. This allows us to determine whether there exists a matching that covers all values from a violated Hall interval. We recall definition of the set I_S (Equation (6.25): $I_S = |X_i|D(X_i) \cap S \neq \emptyset|$.

Definition 6.15 (Violated unstable set) *A set S is a violated unstable set if and only if $I_S < |S|$*

Consider an example of a violated Hall interval that contains a violated unstable set.

Example 6.22 *Consider the soft* ALL-DIFFERENT$([X_1, \ldots, X_5], N)$ *constraint with domains* $D(X_1) = \ldots = D(X_4) = [2, 3]$, $D(X_5) = [1, 4]$ *and* $N = 1$. *The interval* $[2, 3]$

is a violated Hall interval. However, it does not contain a violated unstable set. The interval $[1, 4]$ *is also a violated Hall interval. However, there exists a violated unstable set* $S = \{1, 4\}$*:* $I_S = 1$ *and* $|S| = 2$*. This means that there is no maximum matching that covers values* 1 *and* 4*.*$\diamond$

To distinguish between violated Hall intervals, we introduce the notion of violated-saturated Hall interval, which is a Hall interval such that we can construct a maximum matching that covers all values in this interval. More formally,

Definition 6.16 *Consider the soft* ALL-DIFFERENT$([X_1, \ldots, X_n], N)$ *constraint. A Hall interval* $[a, b]$ *is a* violated-saturated *Hall interval if and only if it is violated and for any subset* $S \subset [a, b]$*,* $I_S \geq |S|$*.*

The meaning of a *violated-saturated* Hall interval is that this is an violated Hall interval such that there is a matching that covers each value in this interval.

We use results for range consistency and bounds consistency for the GCC^L constraint (Section 6.8.1) to obtain the maximum matching.

Proposition 6.2 *Consider the soft* ALL-DIFFERENT$([X_1, \ldots, X_n], N)$ *constraint. Let* $[a, b]$ *be a violated Hall interval and* P *be a set of interval domain variables that are contained inside this interval,* $P = \{X_i | D(X_i) \subseteq [a, b]\}$*. The interval* $[a, b]$ *is a violated-saturated Hall interval if and only if* GCC$^L([P], [1, \ldots, 1], [|P|, \ldots, |P|])$ *is satisfiable.*

Proof: Suppose $[a, b]$ is a violated-saturated Hall interval. As variables in P have interval domains it is sufficient to show that the GCC$^L([P], [1, \ldots, 1])$ constraint has a bounds solution. Hence, conditions of Theorem 6.19 are applied. Note that all lower bounds l_v are ones in our case. Hence, $\sum_{v \in S} l_v = |S|$.

Suppose GCC$^L([P], [1, \ldots, 1], [|P|, \ldots, |P|])$ is violated. Then there exists a violated unstable set $S' \in [a, b]$ such that

$$I_{S'} < |S'|.$$

This contradicts the assumption that $[a, b]$ is a violated-saturated Hall interval.

As the GCC^L constraint has a solution, for any set of values

$$I_S \geq |S|.$$

Hence, $[a, b]$ is a violated-saturated Hall interval. $\diamond$

Theorem 6.42 *Let $G(V, E)$, $V = A \cup B$, $|A| = n$, $|B| = d$, $n \leq d$ be a convex bipartite graph, $M(G)$ be a maximum matching in the graph, $|M(G)| < n$. Let $[a, b]$ be a violated-saturated Hall interval and P be a set of variables that are contained inside this interval, $P = \{X_i | D(X_i) \subseteq [a, b]\}$. Let $G'(V', E')$, $V' = A' \cup B'$, $A' = A \cap P$, $B' = B \cap [a, b]$ be a graph induced by P and $[a, b]$. Then there exists a matching in G' that covers values $[a, b]$.*

Proof: As $[a, b]$ is a violated-saturated Hall interval, there exists a solution of the $\text{GCC}^L([P], [l_a, \ldots, l_b])$ constraint, where $l_a = \ldots = l_b = 1$ (Proposition 6.2). This solution is a required matching in G'. $\diamond$

We show that each violated Hall interval contains a violated-saturated Hall interval.

Theorem 6.43 *Let $[a, b]$ be a violated Hall interval and P be a set of interval domain variables that are contained inside this interval, $P = \{X_i | D(X_i) \subseteq [a, b]\}$. Then, there exists a violated-saturated Hall interval $[c, d]$ such that $[c, d] \subseteq [a, b]$.*

Proof: Consider a violated Hall interval $[a, b]$. We have

$$|P| \quad > \quad b - a + 1. \tag{6.42}$$

As $[a, b]$ is not a violated-saturated Hall interval, there is a violated unstable set $S = \{f_1, \ldots, f_p\}$, $S \subseteq [a, b]$ such that

$$I_S < |S|.$$

We denote P' a subset of variables that overlap S. We assume that S is a maximal violated unstable set. It is easy to see that there exists a unique maximum violated unstable set because a union of two violated unstable sets gives a violated unstable set.

Next we remove variables P' and values S from the set of variables P and the set of values $[a, b]$. This partitions the interval $[a, b]$ into a set of intervals:

$$F = \{[a, f_1 - 1], [f_1 + 1, f_2 - 1], \ldots, [f_p + 1, b]\}. \tag{6.43}$$

Note that domain of any variable X_i, $X_i \in P \setminus P'$ is completely contained inside a single interval in F, otherwise X_i overlaps with S and belongs to P'. As S is the maximum violated unstable set S, each interval $[f_j + 1, f_{j+1} - 1] \in F$ is a violated-saturated Hall interval. $\diamond$

We partition the interval of values $[1, d]$ into violated-saturated and non-violated Hall intervals. Let $H = \{[a_1, b_1], \ldots, [a_p, b_p]\}$ be a maximum set of all maximal-length violated-saturated Hall intervals and Hv_i be the amount of violation for H_i: $Hv_i = b_i - a_i + 1 - |D(X_i) \subseteq [a_i, b_i]|$. The intervals in H are disjoint, otherwise they are not maximal in length. The rest of the intervals we denote F, $F = [1, d] \setminus H$. Note that any interval $[a, b] \in F$ is not violated by Theorem 6.43. Otherwise, it would contain a violated-saturated Hall interval inside. The next theorem shows how to construct a maximum matching that uses this partitioning.

Theorem 6.44 *Let $G(V, E)$, $V = A \cup B$, $|A| = n$, $|B| = d$, $n \leq d$ be a convex bipartite graph, $M(G)$ be a maximum matching in the graph, $|M(G)| < n$. Let $H = \{[a_1, b_1], \ldots, [a_p, b_p]\}$ be a set of all maximal-length violated-saturated Hall intervals. Then there exists a maximal matching of size $n - \sum_{i=1}^{p} Hv_i$.*

Proof: The set H partitions the interval of values $[1, d]$ into maximal-length violated-saturated Hall intervals and non-violated intervals F, $F = [1, d] \setminus H$. We denote the set of variables that are contained inside $\cap_{i=1}^{p}[a_i, b_i]$ as V_1 and the rest of variables as V_2, $V_2 = V \setminus V_1$.

Matching inside violated-saturated Hall interval. Consider a maximal violated-saturated Hall interval, $H_i = [a_i, b_i]$. By Theorem 6.42 we can construct a matching of size $b_i - a_i + 1$ using variables whose domains are inside the interval $[a_i, b_i]$.

Let P_i be a set of variables that are completely contained inside the interval $[a_i, b_i]$. Note that $P_i \cap P_j = \emptyset$, $i \neq j$ by construction of maximal-length violated-saturated Hall intervals and $\bigcup_{i=1}^{p} P_i = V_1$. As intervals in H are disjoint, we can construct a matching that covers $\bigcup_{i=1}^{p}[a_i, b_i]$ using variables in V_1.

Matching outside violated-saturated Hall interval. We remove vertices $V_1 \cup (\cup_{i=1}^{p}[a_i, b_i])$ and the corresponding edges from the graph G. We denote the new graph $G'(V', E')$, $V = A' \cup B'$. The remaining graph is a convex graph because the original graph was convex. The size of the partition B' is $|B| - \sum_{i=1}^{p}(b_i - a_i + 1)$. We rename vertices in B' as $1, \ldots, |B| - \sum_{i=1}^{p}(b_i - a_i + 1)$ (similar to the shrinking procedure in Theorem 6.7). Suppose the renaming maps

$c_i \leftrightarrow c'_i$, ..., $d_i \leftrightarrow d'_i$, where $\{c_i, d_i\} \in B$, $\{c'_i, d'_i\} \in B'$,

The size of the partition A' is $|A| - |V_1| = |A| - \sum_{i=1}^{p}(b_i - a_i + 1) - \sum_{i=1}^{p} Hv_i$. We show that this graph has a matching of size A'.

Suppose there is no matching of size $|A'|$. Then, there exists a violated Hall interval $[a'_i, b'_i]$. By Theorem 6.43, $[a'_i, b'_i]$ contains a violated-saturated Hall sub-interval $[c'_i, d'_i]$.

We map values in $[c'_i, d'_i]$ back to original values in B. If $[c_i, d_i]$ form a continuous interval we reached a contradiction as $[c_i, d_i] \not\subseteq H$. Otherwise, these values form a set S'.

Let $H' = \{[a_{i_j}, b_{i_j}] | [a_{i_j}, b_{i_j}] \subseteq [c_i, d_i]\}$, $H' \subseteq H$, $H' \cup S' = H$ be a set of violated-saturated Hall intervals of maximum length that are contained inside $[c_i, d_i]$. Then, the interval $[c_i, d_i]$ does not contain violated unstable sets, because $[c_i, d_i]$ is a union of elements of S' and $\cup_{[a_{i_j}, b_{i_j}] \in H'} [a_{i_j}, b_{i_j}]$ that do contain such violated unstable sets by construction. Therefore, $[c_i, d_i]$ is a violated-saturated Hall interval. This contradicts that the intervals in H' are violated-saturated Hall intervals of *maximal length*.

Maximum matching construction. As V_1 and V_2 have a disjoint neighbourhood, we can construct a matching of size $[\sum_{i=1}^{p}(b_i - a_i + 1)] + [n - \sum_{i=1}^{p}(b_i - a_i + 1) - \sum_{i=1}^{p} Hv_i] = n - \sum_{i=1}^{p} Hv_i$.

This matching is a maximal matching because the only way to increase it is to use an unused vertex that is inside a violated-saturated Hall interval. However, all values from violated-saturated Hall intervals are used. $\diamond$

6.9.2 Decompositions of the NVALUE constraint

Section 6.9.1 shows that the decomposition of the NVALUE constraint into ATMOSTNVALUE and ATLEASTNVALUE does not hinder bounds propagation on the variable N and range and bounds propagation on the variables X. Therefore, we present a decomposition of the ATMOSTNVALUE and ATLEASTNVALUE into a set of primitive arithmetic constraints independently. Section 6.9.2.1 shows range consistency and bounds consistency decompositions of the ATMOSTNVALUE and Section 6.9.2.2 proposes complexity improvements of these decomposition. Section 6.9.2.3 shows range consistency and bounds consistency decompositions of the ATLEASTNVALUE.

6.9.2.1 ATMOSTNVALUE constraint

We now give a decomposition for the ATMOSTNVALUE constraint that does not hinder range consistency and bounds consistency. To decompose the ATMOSTNVALUE constraint, we introduce 0/1 variables, A_{ilu} to represent whether X_i uses a value in the interval $[l, u]$, and 'counter' variables, M_{lu} with domains $[0, \min(u - l + 1, n)]$ which count the number of values taken inside the interval $[l, u]$. Note that this construction

is similar to the alternative decomposition of the ALL-DIFFERENT constraint and the OVERLAPPINGALLDIFF constraint (Sections 6.4.3 and 6.7)

To constrain these introduced variables, we post the following constraints:

$$A_{ilu} = 1 \iff X_i \in [l,u] \qquad \forall\, 1 \le i \le n, 1 \le l \le u \le d \tag{6.44}$$

$$A_{ilu} \le M_{lu} \qquad \forall\, 1 \le i \le n, 1 \le l \le u \le d \tag{6.45}$$

$$M_{1u} = M_{1k} + M_{(k+1)u} \qquad \forall\, 1 \le k < u \le d \tag{6.46}$$

$$M_{1d} \le N \tag{6.47}$$

We now prove that this decomposition does not hinder propagation.

Theorem 6.45 *Enforcing domain consistency on constraints* (6.44) *and bounds consistency on* (6.45)–(6.47) *is equivalent to range consistency on variables X and bounds consistency on the variable N for* ATMOSTNVALUE $([X_1,\ldots,X_n],N)$*, and takes $O(nd^3)$ time to enforce down the branch of the search tree.*

Proof: We show that the decomposition detects disentailment. By Theorem 6.30 the ATMOSTNVALUE constraint is disentailment if $\alpha_I(G) = card_{\downarrow}(X) > ub(N)$.

Let $Y = \{X_{p_1},\ldots,X_{p_k}\}$ be a maximum cardinality subset of variables of X whose domains are pairwise disjoint (i.e., $D(X_{p_i}) \cap D(X_{p_j}) = \emptyset, \forall i,j \in 1..k, i \neq j$). Let $I_Y = \{[b_i,c_i] \mid b_i = lb(D(X_{p_i})),\ c_i = ub(D(X_{p_i})), X_{p_i} \in Y\}$ be the corresponding ordered set of disjoint ranges of the variables in Y. Proposition 6.1 shows that $|Y| = card_{\downarrow}(X)$.

Consider interval $[b_i,c_i] \in I_Y$. Constraints (6.45) ensure that variables $M_{b_ic_i}$ $i = [1,\ldots,k]$ are greater than or equal to 1 and constraints (6.46), by chaining the lower bounds procedure (introduced in proof of Theorem 6.22), ensure that

$$M_{1d} = M_{1,b_1-1} + M_{b_1,c_1} + M_{c_1+1,b_2-1} + \ldots + M_{b_N,c_N} + M_{c_N+1,d}. \tag{6.48}$$

and

$$lb(M_{1d}) \ge lb(M_{1,b_1-1}) + lb(M_{b_1,c_1}) + lb(M_{c_1+1,b_2-1}) + \ldots + lb(M_{b_N,c_N}) + lb(M_{c_N+1,d}).$$

Therefore, the lower bound of the variable N is greater than or equal to $card_{\downarrow}(X)$ which leads to the failure of the last constraint in the chaining lower bounds procedure.

Bounds consistency on N. The chaining lower bounds procedure ensures that $lb(N) \ge card_{\downarrow}(X)$. Hence, N is bounds consistent by Theorem 6.31.

Range consistency on X. By Theorem 6.34 the only case when pruning might occur is if the variable N is ground and $card_{\downarrow}(X) = N$. In this case $|Y| = card_{\downarrow}(X) = lb(N) = ub(N)$. By chaining upper bounds procedure (introduced in proof of Theorem 6.22) on equation (6.48) we get that the upper bounds of variables M_{b_i,c_i}, $[b_i, c_i] \in I_Y$ are set to their lower bounds and variables $M_{c_{i-1}+1,b_i-1}$ that correspond to intervals outside the set I_Y are forced to zero.

As this argument holds for any maximum set of pairwise disjoint variables all values outside the maximum cardinality set of intersection intervals $\mathcal{C}$ will be pruned. By Theorem 6.32 the constraint is range consistent .

Complexity argument: There are $O(nd^2)$ constraints (6.44) and constraints (6.45) that can be woken $O(d)$ times down the branch of the search tree. This requires $O(d)$ time down the branch for a total of $O(nd^3)$ down the branch. There are $O(d^2)$ constraints (6.46) which can be woken $O(n)$ times down the branch and all invocations down the branch take $O(d)$ time. This gives a total of $O(nd^2)$. The final complexity down the branch of the search tree is therefore $O(nd^3)$. $\diamond$

Similar to the ALL-DIFFERENT and GCC constraints we can easily modify the decomposition to enforce bounds consistency if we replace constraints (6.44) by constraints (6.3) and (6.4).

Theorem 6.46 *Enforcing bounds consistency on constraints* (6.3)–(6.4) *and bounds consistency on* (6.45)–(6.47) *is equivalent to bounds consistency on* ATMOSTNVALUE $([X_1, \ldots, X_n], N)$*, and takes $O(nd^2)$ time to enforce down the branch of the search tree.*

Proof: The proof is similar to Theorem 6.45. The only difference is that channelling constraints (6.3)–(6.4) only prunes variable bounds.

Complexity argument: The complexity reduces to $O(nd^2)$ as bounds consistency on (6.3) and (6.4) is in $O(nd^2)$ compared to $O(nd^3)$ on (6.44). This gives $O(nd^2)$ time complexity down a branch of the search tree. $\diamond$

Consider how the two decompositions work on our running example.

Example 6.23 . *We recall that domains of the variables are*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N	*	*				

Bounds consistency on N**.** *As* $X_2 = 2$ *and* $X_5 \in [4, 5]$*, we have* $lb(M_{2,2}) \geq 1$ *and* $lb(M_{4,5}) \geq 1$*. By the chaining lower bounds procedure, we get* $lb(M_{1,6}) = ub(M_{1,6}) = 2$*. This enforces bounds consistency on the variable* N*. As* N *is fixed we have to prune variables* X*.*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N		*				

Bounds consistency on X*. Now we consider the maximum cardinality set of intersection intervals* $\mathcal{C}$*. As we have a single maximum set of disjoint intervals,* $\mathcal{C} = \{[2, 2], [4, 5]\}$*. The implied constraint*

$$M_{1,6} = M_{1,1} + M_{2,2} + M_{3,3} + M_{4,5} + M_{6,6}$$

is tight as $ub(M_{1,6}) = lb(M_{2,2}) + lb(M_{4,5})$*. Therefore, by the chaining upper bounds procedure, the* $M_{1,1}$*,* $M_{3,3}$ *and* $M_{6,6}$ *are fixed to zero. By channelling constraints* (6.3)–(6.4) *the values 1 and 6 are pruned from* $D(X_1)$*.*

	1	2	3	4	5	6
X_1		$*$	$*$	$*$	$*$	
X_2		$*$				
X_3		$*$				
X_4		$*$	$*$	$*$		
X_5				$*$	$*$	
X_6				$*$	$*$	
N		$*$				

Range consistency on X. *To achieve range consistency we also need to enforce domain consistency on constraints* (6.44) *which prunes the value* 3 *from domains of variables* X_1 *and* X_3.

	1	2	3	4	5	6
X_1		$*$		$*$	$*$	
X_2		$*$				
X_3		$*$				
X_4		$*$		$*$		
X_5				$*$	$*$	
X_6				$*$	$*$	
N		$*$				

6.9.2.2 Faster decompositions

We can improve how the solver handles this decomposition of the ATMOSTNVALUE constraint by adding implied constraints and by implementing specialised propagators. The aim of the specialised propagators is to improve channeling between variables X and decomposition variables to speed up the enforcement of range consistency. Our first improvement is to add an implied constraint and enforce bounds consistency on it:

$$M_{1d} \quad = \quad \sum_{i=1}^{d} M_{ii} \tag{6.49}$$

This does not change the asymptotic complexity of reasoning with the decomposition, nor does it improve the level of propagation achieved. However, we have found that the fixpoint of propagation is reached quicker in practice with such an implied constraint.

Our second improvement is to bring the complexity of enforcing range consistency down to the same complexity as bounds consistency. Note that decomposition (6.44)–(6.47)

achieves range consistency in $O(dn^3)$ time. The time complexity is dominated by the channelling constraints (6.44). To improve the complexity we start from the bounds consistency decomposition that runs in $O(dn^2)$ time and extend it with $O(d \log d)$ Boolean variables $B_{il(l+2^k)} \in [0,1], 1 \leq i \leq n, 1 \leq l \leq d, 0 \leq k \leq \lfloor \log d \rfloor$. The following constraint ensures that $B_{ijj} = 1 \iff X_i = j$.

Definition 6.17 *The* DOMAINBITMAP$(X_i, [B_{i11}, \ldots, B_{idd}])$ *constraint holds if and only if* $B_{ijj} = 1 \iff X_i = j$, $j = 1, \ldots, d$,

We can then use the following clausal constraints to channel from variables M_{lu} to these variables and on to the X variables. These constraints are posted for every $1 \leq i \leq n, 1 \leq l \leq u \leq d, 1 \leq j \leq d$ and integers k such that $0 \leq k \leq \lfloor \log d \rfloor$:

$$B_{ij(j+2^{k+1}-1)} = 1 \vee B_{ij(j+2^k-1)} = 0 \tag{6.50}$$

$$B_{ij(j+2^{k+1}-1)} = 1 \vee B_{i(j+2^k)(j+2^{k+1}-1)} = 0 \tag{6.51}$$

$$M_{lu} \neq 0 \vee B_{il(l+2^k-1)} = 0 \qquad 2^k \leq u - l + 1 < 2^{k+1} \tag{6.52}$$

$$M_{lu} \neq 0 \vee B_{i(u-2^k+1)u} = 0 \qquad 2^k \leq u - l + 1 < 2^{k+1} \tag{6.53}$$

The variable $B_{il(l+2^k-1)}$, similarly to the variables A_{lu}, is true when $X_i \in [l, l+2^k-1]$, but instead of having one such variable for every interval, we only have them for intervals whose length is a power of two.

We assume that we do not branch on auxiliary variables in this work. Hence, the only case when enforcing bounds consistency on constraints (6.3) and (6.4) is not equivalent to enforcing range consistency on constraints (6.44) is when A_{ilu} is forced to zero by the constraint (6.45), $A_{ilu} \leq M_{lu}$. This can only happen if M_{lu} is forced to take the value zero by propagation. Hence, to enforce range consistency on the ATMOSTNVALUE constraint we only need to enhance bounds consistency decomposition with the following inference: if M_{lu} is forced to take the value zero then values $[l, u]$ have to be removed from domains ofvariables X .

Theorem 6.47 *Let* M_{lu} *be* 0. *Enforcing range consistency on constraints* (6.50)–(6.53), *the* DOMAINBITMAP$(X_i, [B_{i11}, \ldots, B_{idd}])$ *constraint removes values* $[l, u]$ *from domains of all variables* X. *This takes* $O(nd^2)$ *time to enforce down the branch of the search tree.*

Proof: When $M_{lu} = 0$, with $2^k \leq u - l + 1 < 2^{k+1}$, constraints (6.52) –(6.53) set to 0 the B variables that correspond to the two intervals of length 2^k that start at l and finish at u, respectively. In turn, constraints (6.50)–(6.51) set to 0 the B variables that correspond to

intervals of length 2^{k-1}, all the way down to intervals of size 1. These trigger constraints (6.17), so all values in the interval $[l, u]$ are removed from the domains of all variables.

Complexity argument: We can propagate DOMAINBITMAP in a constant time per variable change then we can enforce range consistency on this constraint in time $O(d)$ over a branch, and $O(nd)$ for all variables X_i. Note that range consistency can be enforced on each of constraints (6.50)– (6.53) in constant time over a branch. There exist $O(nd \log d)$ of the constraints (6.50)–(6.51) and $O(nd^2)$ of the constraints (6.52)–(6.53), so the total time to propagate them all down a branch is $O(nd^2)$. $\diamond$

Example 6.24 *Suppose* $X_1 \in [5, 9]$. *Then, by constraints* (6.3)–(6.4), $A_{14} = 0$, $A_{19} = 1$ *and by constraints* (6.45), $M_{59} > 0$. *Conversely, suppose* $M_{59} = 0$ *and* $X_1 \in [1, 10]$. *Then, by constraints* (6.52)–(6.53), *we get* $B_{158} = 0$ *and* $B_{169} = 0$. *From* $B_{158} = 0$ *and* (6.50)–(6.51) *we get* $B_{156} = 0$, $B_{178} = 0$, $B_{155} = B_{166} = B_{177} = B_{188} = 0$, *and by* (6.17), *the interval* $[5, 8]$ *is pruned from* X_1. *Similarly,* $B_{169} = 0$ *causes the interval* $[6, 9]$ *to be removed from* X_1, *so* $X_1 \in [1, 4] \cup \{10\}$. $\diamond$

6.9.2.3 ATLEASTNVALUE constraint

There is a similar decomposition for the ATLEASTNVALUE constraint. We introduce 0/1 variables, A_{ilu} to represent whether X_i uses a value in the interval $[l, u]$, and integer variables, E_{lu} with domains $[0, n]$ to count the number of times values in $[l, u]$ are *reused*, that is, the amount of violation H_v for the interval $[l, u]$. To constrain these introduced variables, we post the following constraints:

$$A_{ilu} = 1 \iff X_i \in [l, u] \qquad \forall\, 1 \le i \le n, 1 \le l \le u \le d \tag{6.54}$$

$$E_{lu} \ge \textstyle\sum_{i=1}^{n} A_{ilu} - (u - l + 1) \qquad \forall\, 1 \le l \le u \le d \tag{6.55}$$

$$E_{1u} = E_{1k} + E_{(k+1)u} \qquad \forall\, 1 \le k < u \le d \tag{6.56}$$

$$N \le n - E_{1d} \tag{6.57}$$

We now prove that this decomposition does not hinder propagation in general.

Theorem 6.48 *Enforcing domain consistency on constraints* (6.54) *and bounds consistency on* (6.55)–(6.57) *is equivalent to enforcing range consistency on variables* X *and bounds consistency on the variable* N *for* ATLEASTNVALUE$([X_1, \ldots, X_n], N)$, *and takes* $O(nd^3)$ *time down a branch of the search tree.*

Proof: The value $card_{\uparrow}(X)$ is equal to the size of a maximum matching $M(G_v)$ in the variable-value graph of the constraint. Since $N \le n - E_{1d}$, we show that the lower bound

of E_{1d} is equal to $n - |M(G_v)|$, where $|M(G_v)|$ is the size of the maximum matching.[1] Theorem 6.44 shows that the size of the maximum matching equals to $n - \sum_{i=1}^{p} Hv_i$, where $H = \{[a_1, b_1], \ldots, [a_k, b_k]\}$ is a maximum set of maximal length violated-saturated Hall intervals and $Hv_i = b_i - a_i + 1 - |D(X_i) \subseteq [a_i, b_i]|$, $i = 1, \ldots, p$. Therefore, it is sufficient to show that $lb(E_{1d})$ is greater than or equal to the total amount of violation: $\sum_{i=1}^{p} Hv_i$.

We partition the interval $[1, d]$ into a set of maximal violated-saturated and non-violated Hall intervals. Let $H = \{[b_j, c_j]\}$, $j = 1, \ldots, k$ be a set of maximal violated-saturated Hall intervals and $P = \{X_i | D(X_i) \subseteq \cup_{j=1}^{k} [b_j, c_j]\}$

As $[b_j, c_j]$ is a violated-saturated Hall interval we have

$$Hv_{[b_j,c_j]} = lb(E_{b_j,c_j}) = \sum_{i=1}^{n} lb(A_{ib_jc_j}) - (c_j - b_j + 1)$$

Consider the remaining intervals $\{[1, d] \setminus H\}$: $F = \{[1, b_1 - 1], [c_1 + 1, b_2 - 1], \ldots, [c_k, d]\}$.

Any interval in F is not a violated Hall interval by Theorem 6.42. Therefore, $lb(E_{b_j+1,c_{j+1}-1}) = 0$ for any interval in F. Constraints (6.55)–(6.57) imply

$$E_{1d} = E_{1,b_1-1} + E_{b_1,c_1} + E_{c_1+1,b_2-1} + \ldots + E_{b_k,c_k} + E_{c_k+1,d}.$$

and by the chaining lower bounds procedure we have that

$$lb(E_{1d}) \geq lb(E_{1,b_1-1}) + lb(E_{b_1,c_1}) + lb(E_{c_1+1,b_2-1}) + \ldots + lb(E_{b_k,c_k}) + lb(E_{c_k+1,d}).$$

As lower bounds of non-violated intervals are zeros we have:

$$lb(E_{1d}) \geq lb(E_{b_1,c_1}) + + \ldots + lb(E_{b_k,c_k}) = \sum_{i=1}^{k} Hv_i$$

Therefore, $lb(E_{1d}) \geq \sum_{i=1}^{k} Hv_i$ and bounds propagation on

$$N \leq n - E_{1d}$$

implies that $ub(N) \leq n - lb(E_{1d}) = n - \sum_{i=1}^{k} Hv_i = |M(G)|$.

Hence, the decomposition verifies that $lb(N) \leq |M(G_v)|$ and by Theorem 6.36 detects disentailment.

Bounds consistency on N. The chaining lower bounds procedure ensures that $ub(N) \leq |M(G)|$. By Theorem 6.37 the variable N is bounds consistent .

[1] We assume that E_{1d} is not pruned by other constraints.

Range consistency on X. By Theorem 6.34 the only case when pruning might occur is if the variable N is ground and $card_{\uparrow}(X) = N$. By Theorem 6.44 the size of the maximum matching is $card_{\uparrow}(X) = n - \sum_{i=1}^{k} Hv_i$. Therefore, the variable E_{1d} is fixed to $\sum_{i=1}^{k} Hv_i$.

Consider again the partitioning of the interval $[1, d]$ into violated-saturated Hall intervals and the non violated intervals. Constraints (6.55)–(6.57) imply

$$E_{1d} = E_{1,b_1-1} + E_{b_1,c_1} + E_{c_1+1,b_2-1} + \ldots + E_{b_k,c_k} + E_{c_k+1,d}. \tag{6.58}$$

and by the chaining upper bounds procedure, taking into account that $\sum_{[b_j,c_j]\in H} lb(E_{b_j,c_j}) = \sum_{i=1}^{k} Hv_i$, we have that upper bounds of the variables $E_{1(b_1-1)}, E_{b_1c_1}, E_{(c_1+1)(b_2-1)}, \ldots, E_{b_kc_k}, E_{(c_k+1)d}$ are fixed to their lower bounds. In particular, for any non-violated interval $[a_j, d_j]$, $E_{a_jd_j}$ is fixed to zero and for any violated-saturated interval $[b_j, c_j]$, $E_{b_jc_j}$ is fixed to $\sum_{i=b_j}^{c_j} Hv_i$.

Consider a variable-value pair $X_i = v$ that does not have a support. We consider two cases when $X_i = v$ does not have a support. We recall that $P = \{X_i | D(X_i) \subseteq \cup_{j=1}^{k} [b_j, c_j]\}$:

1. $D(X_i)$ is contained inside one of the violated-saturated Hall intervals, $X_i \in P$;
2. $D(X_i)$ is not contained inside one of the violated-saturated Hall intervals, $X_i \in X \setminus P$.

Case 1. Consider the interval $[b_j, c_j] \in H$, such that $D(X_i) \subseteq [b_j, c_j]$. Let $P' = \{X_i | D(X_i) \subseteq [b_j, c_j])\}$. First we show that variables outside of P' cannot take values in the interval $[b_j, c_j]$. This shows that the problem of constructing a matching of size $c_j - b_j + 1$ using variables in P' such that $X_i = v$ is isolated from the rest of the problem.

The variable $E_{b_jc_j}$ equals $\sum_{i=b_j}^{c_j} Hv_i$ and $\sum_{i=1}^{n} A_{ib_jc_j} \geq |P'| = c_j - b_j + 1 + \sum_{i=b_j}^{c_j} Hv_i$. Therefore, $\sum_{i=1}^{n} A_{ib_jc_j} = c_j - b_j + 1$ by constraints (6.55). Hence, the values $[b_j, c_j]$ are pruned from domains of variables $X \setminus P'$.

Next, we need to prove that there exists a matching of size $c_j - b_j + 1$ using variables in P' such that $X_i = v$.

Proposition 6.2 shows a connection between finding a matching that covers all values from the interval and the GCC$^L([P'], [1, \ldots, 1], [u_{b_j}, \ldots, u_{c_j}])$ constraint and $u_{b_j} = \ldots = u_{c_j} = |P'|$. We can use this connection to construct a matching such that $X_i = v$. By Theorem 6.20 GCC^L does not have a solution with $X_i = v$ if and only if

- there exists an unstable set S, $S \subseteq [b_j, c_j]$ that *does not* contain v and $D(X_j)$ intersects S,

We modify the proof of Theorem 6.4 to prove that this case is handled by the decomposition.

Let $S = \{v_1, \ldots, v_k\}$ be an unstable set such that $v \notin S$ and $D(X_i) \cap S \neq \emptyset$. W.l.o.g. we assume that $v_j < v < v_{j+1}$ As S is an unstable set we have:

$$I_S = |S|,$$

where $I_S = |X_i | D(X_i) \cap S \neq \emptyset|$ (Equation (6.25)).

The number of variables in $Q' = P' \setminus I_S = \{X_i | X_i \notin I_S\}$ is equal to $c_j - b_j + 1 - |S|$. Any variable X_i, $X_i \notin I_S$ can be contained inside only one of the intervals $[b_j, v_1 - 1], [v_1 + 1, v_2 - 1], \ldots, [v_k + 1, c_j]$. Therefore

$$U_{[b_j, v_1 - 1]} + U_{[v_1+1, v_2-1]} + \ldots + U_{[v_k+1, c_j]} = |Q'| = c_j - b_j + 1 - |S|, \tag{6.59}$$

where $U_{[l,u]} = |X_i | D(X_i) \subseteq [l, u]|$ (Equation (6.26)).

The amount of violation in the interval $[b_j, c_j]$ is

$$lb(E_{b_j c_j}) = ub(E_{b_j c_j}) = |I_S| + |Q'| - (c_j - b_j + 1). \tag{6.60}$$

Thanks to constraints (6.55), we know that for any interval $[l, u]$:

$$lb(E_{lu}) \geq U_{[l,u]} - (u - l + 1). \tag{6.61}$$

So, taking into account constraints (6.59)–(6.61):

$$\begin{aligned} lb(E_{b_j(v_1-1)}) + lb(E_{v_1 v_1}) + lb(E_{(v_1+1)(v_2-1)}) + \ldots + lb(E_{v_k v_k}) + lb(E_{(v_k+1)c_j}) \geq \\ U_{[b_j, v_1-1]} + U_{[v_1+1, v_2-1]} + \ldots + U_{[v_k+1, c_j]} - (c_j - b_j + 1 - |S|) + \sum_{v=1}^{k} lb(E_{vv}) = \\ |Q'| - (c_j - b_j + 1 - |S|) + \sum_{v=1}^{k} lb(E_{vv}) \end{aligned}$$

As S is unstable set we have $S = |I_S|$ which implies

$$\begin{aligned} lb(E_{b_j(v_1-1)}) + lb(E_{(v_1+1)(v_2-1)}) + \ldots + lb(E_{(v_k+1)c_j}) \\ \geq |I_S| + |Q'| - (c_j - b_j + 1) + \sum_{v=1}^{k} lb(E_{vv}). \end{aligned} \tag{6.62}$$

By Equation (6.60)

$$\begin{aligned} lb(E_{b_j(v_1-1)}) + lb(E_{(v_1+1)(v_2-1)}) + \ldots + lb(E_{(v_k+1)c_j}) \\ \geq ub(E_{b_j c_j}) + \sum_{v=1}^{k} lb(E_{vv}). \end{aligned} \tag{6.63}$$

On the other hand we have

$$ub(E_{b_j c_j}) \geq lb(E_{b_j(v_1-1)}) + \ldots + lb(E_{(v_k+1)c_j}) + \sum_{v=1}^{k} lb(E_{vv}). \tag{6.64}$$

Equations (6.63) and (6.64) have to hold simultaneously so we get

$$ub(E_{b_j c_j}) = lb(E_{b_j(v_1-1)}) + lb(E_{(v_1+1)(v_2-1)}) + \ldots + lb(E_{(v_k+1)c_j}). \tag{6.65}$$

By the chaining upper bound procedure on the implied constraint

$$E_{1d} = E_{1(b_j-1)} + E_{b_j(v_1-1)} + E_{v_1 v_1} + E_{(v_1+1)(v_2-1)} + \ldots + E_{v_k v_k} + E_{(v_k+1)c_j}$$

and taking into account equation (6.65) and that variables $E_{1,b_1-1}, E_{b_1,c_1}, E_{c_1+1,b_2-1}, \ldots, E_{b_k,c_k}, E_{c_k+1,d}$ are ground, the decomposition fixes lower bounds of variables $E_{b_j(v_1-1)}, E_{(v_1+1)(v_2-1)}, \ldots, E_{(v_k+1)c_j}$ to their upper bounds.

As X_i overlaps S and $v \notin S$, then $v \in [v_j + 1, v_{j+1} - 1]$ and $A_{i(v_j+1)(v_{j+1}-1)}$ is unset as $D(X_i) \not\subseteq [v_j + 1, v_{j+1} - 1]$. The variable $E_{(v_j+1)(v_{j+1}-1)}$ is fixed to $U_{[v_j+1,v_{j+1}-1]} - v_{j+1} + v_j - 1$. Therefore, $U_{[v_j+1,v_{j+1}-1]} \geq \sum_{i=1}^{n} A_{i(v_j+1)(v_{j+1}-1)}$. On the other hand, $U_{[v_j+1,v_{j+1}-1]} \leq \sum_{i=1}^{n} A_{i(v_j+1)(v_{j+1}-1)}$ by definition. This make the constraint tight and sets the variable $A_{i(v_j+1)(v_{j+1}-1)}$ to 0 and prunes v from $D(X_i)$.

Case 2. Consider a variable-value pair $X_i = v$, $X_i \in X \setminus P$. Following the proof of Theorem 6.44 we need to construct a matching of size $n - |P|$ using value from $[1, d] \setminus IH$, where $IH = \bigcup_{[b_j,c_j] \in H} [b_j, c_j]$ is union of violated-saturated sets, such that $X_i = v$. If we ignore values in IH then we have the following condition for the variable-value pair $X_i = v$ to be range inconsistent :

- there exists a Hall interval $[l, u]$ that contains v and $D(X_j)$ is not included in the interval $[l, u]$

To adapt this condition to our case we need to take into account that the Hall interval $[l, u]$ is not necessarily an interval of consecutive values as it might contain violated-saturated Hall intervals inside. We show that our decomposition detects this case. Let $H' = \{[b_j, c_j] | [b_j, c_j] \subseteq [l, u]\}$, $j = 1 \ldots, g$, $H' \subseteq H$, be a set of violated-saturated Hall intervals inside $[l, u]$. We denote $\bigcup_{[b_j,c_j] \in H'} [b_j, c_j]$ as IH', the set of variables that are contained inside H' as Q, $Q = \{X_i | D(X_i) \subseteq \bigcup_{[b_j,c_j] \in H'} [b_j, c_j]\}$ and the remaining variables $\{X_i | D(X_i) \cap ([l, u] \setminus IH')) \neq \emptyset\}$ as Q'.

Then we partition $[l, u]$ into a set of intervals $\{[l, b_1 - 1], [b_1, c_1], [c_1 + 1, c_2 - 1], \ldots, [b_g, c_g], [c_g + 1, u]\}$ and the corresponding constraints:

$$E_{lu} = E_{l(b_1-1)} + E_{b_1c_1} + \ldots + E_{b_gc_g} + E_{(c_g+1)u}$$

The lower bounds of the variables E_{a_j,d_j}, $[a_j, d_j] \notin H'$ are set to zero as they are not violated. The interval $[l, u]$ is a Hall interval, if we ignore values from H'. Hence $|Q| = u - l + 1 - |IH'|$ and $|Q'| = |IH'| + \sum_{i=b_j}^{c_j} Hv_i$. Then, $|Q| + |Q'| = u - l + 1 - |IH'| + |IH'| + \sum_{i=b_j}^{c_j} Hv_i = u - l + 1 + \sum_{i=b_j}^{c_j} Hv_i$. This gives

$$\sum_{i=1}^{n} A_{jlu} \geq u - l + 1 + \sum_{i=b_j}^{c_j} Hv_i.$$

On the other hand,

$$E_{lu} = \sum_{i=b_j}^{c_j} Hv_i \geq \sum_{j=1}^{n} A_{jlu} - (u - 1 + 1)$$

This implies that

$$\sum_{i=1}^{n} A_{ilu} = u - l + 1 + \sum_{i=b_j}^{c_j} Hv_i$$

is tight. As $v \in [l, u] \setminus IH'$ and $D(X_i)$ is not completely inside $[l, u]$ the variable A_{ilu} is unset and tightness of the last sum forces A_{ilu} to be 0.

Complexity argument: There are $O(nd^2)$ constraints (6.54) that can be woken $O(d)$ times down the branch of the search tree in $O(1)$, so a total of $O(nd^3)$ down the branch. There are $O(d^2)$ constraints (6.55) which can be propagated in time $O(n)$ down the branch for a total of $O(nd^2)$. There are $O(d^2)$ constraints (6.56) which can be woken $O(n)$ times each down the branch for a total cost in $O(n)$ time down the branch. Thus a total of $O(nd^2)$. The final complexity down the branch of the search tree is therefore $O(nd^3)$. $\diamond$

Similar to the ALL-DIFFERENT, GCC and ATMOSTNVALUE constraints we can easily modify the decomposition to enforce bounds consistency if we replace constraints (6.54) by constraints (6.3) and (6.4).

Theorem 6.49 *Enforcing bounds consistency on constraints* (6.3)–(6.4) *and bounds consistency on* (6.54)–(6.57) *is equivalent to bounds consistency on* ATLEASTNVALUE $([X_1, \ldots, X_n], N)$*, and takes* $O(nd^2)$ *time to enforce down the branch of the search tree.*

Proof: The proof is similar to Theorem 6.45. ⋄

Consider how to achieve range consistency and bounds consistency on our running example.

Example 6.25 *We recall domains of the variables in Example 6.20.*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N					*	*

Bounds consistency on N**.** *Consider the violated saturated Hall intervals* $[2, 2]$*. We partition the interval of values into violated-saturated Hall intervals and non-violated Hall intervals:*

$$[1, 1], [2, 2], [3, 6]$$

and the corresponding implied constraint:

$$E_{16} = E_{11} + E_{22} + E_{36}. \tag{6.66}$$

As there are two variables, X_2 *and* X_3 *whose domains are completely contained inside the intervals* $[2, 2]$*. The constraints* (6.54)–(6.55) *give that* $A_{222} = 1$*,* $A_{322} = 1$*,* $\sum_{i=1}^{6} A_{i22} \leq 2$ *and* $E_{22} \geq 1$*. By the chaining lower bounds procedure on the constraint* (6.66)*, we have that* $lb(E_{16}) \geq 2$*. As* $N \leq 6 - E_{16}$ *we have that* $ub(N) \leq 5$ *and the value* 6 *is pruned from* N*. This fixes* N *to 5.*

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4		*	*	*		
X_5				*	*	
X_6				*	*	
N					*	

Example 6.20 shows that the size of the maximum matching in the graph G_v is 5. By Theorem 6.36 N is bounds consistent .

Bounds consistency on X. *We show that the decomposition detects Hall intervals if we ignore values from violated-saturated Hall intervals. In this example if we ignore the interval $[2,2]$ we can see that the interval $[4,5]$ and $[3,4,5]$ are Hall intervals.*

	1	⊗	3	4	5	6
X_1	*	⊗	*	*	*	*
X_2		⊗				
X_3		⊗				
X_4		⊗	*	*		
X_5		⊗		*	*	
X_6		⊗		*	*	
N					*	

First we show that variables that are not contained inside violated-saturated Hall intervals do not take values from these intervals. Consider again the implied constraint (6.66). *As $E_{16} = 1$ and $lb(E_{22}) \geq 1$ the constraint* (6.66) *forces E_{11} and E_{36} to 0 by the chaining upper bounds procedure. $E_{36} = 0$ forces E_{lu}, $[l,u] \subset [3,6]$.*

Consider the interval $[2,2]$. As $\sum_{i=1}^{6} A_{i22} \leq 2$ and $E_{22} = 1$ we get $\sum_{i=1}^{6} A_{i22} = 2$. This sets A_{i22}, $i = \{1,4,5,6\}$ to 0 and the value 2 is pruned from X_4(constraints (6.3)–(6.4)*). The value 2 is not pruned from X_1 as 2 is not a bound value for this variable.*

Second we show that the decomposition detects Hall intervals. Consider the interval $[4,5]$. There are two variables, X_5 and X_6 whose domains are completely contained inside the intervals $[4,5]$. The constraints (6.3)–(6.4) *give that $A_{545} = 1$, $A_{645} = 1$, $\sum_{i=1}^{6} A_{i45} \leq 2$. The constraints* (6.55) *implies that the sum $\sum_{i=1}^{6} A_{i45}$ is less than or equal to 2. This gives $\sum_{i=1}^{6} A_{i45} = 2$. The channelling constraints* (6.3)–(6.4) *prune 4 from X_4. Again, the value 4 is not pruned from the domain of X_1 as it is not a bound value. At this point the*

constraint is bounds consistent .

	1	2	3	4	5	6
X_1	*	*	*	*	*	*
X_2		*				
X_3		*				
X_4			*			
X_5				*	*	
X_6				*	*	
N					*	

Range consistency on X. *By the same arguments as above but replacing constraints* (6.3)–(6.4) *with* (6.54). *This prunes the values* 2 *and* 4 *from* X_1.

	1	2	3	4	5	6
X_1	*		*		*	*
X_2		*				
X_3		*				
X_4			*			
X_5				*	*	
X_6				*	*	
N					*	

Consider the interval $[3, 5]$ *which is a Hall interval. Similar to the reasoning about the interval* $[4, 5]$, *we derive that the value* $[3, 5]$ *have to be pruned from* $D(X_1)$. *This makes the constraint range consistent .*

	1	2	3	4	5	6
X_1	*					*
X_2		*				
X_3		*				
X_4			*			
X_5				*	*	
X_6				*	*	
N					*	

$\diamond$

6.9.3 Other decompositions of the NVALUE constraint

A natural way to decompose the NVALUE constraint by introducing 0/1 variables to represent which values are used and posting a sum constraint on these introduced variables:

$$X_i = j \rightarrow B_j = 1 \qquad \forall 1 \leq i \leq n, 1 \leq j \leq d \tag{6.67}$$

$$B_j = 1 \rightarrow \bigvee_{i=1}^{n} X_i = j \qquad \forall 1 \leq j \leq d \tag{6.68}$$

$$\sum_{j=1}^{d} B_j = N \tag{6.69}$$

Note that constraint (6.69) is not a fixed arity constraint, but can itself be decomposed to ternary sums without hindering bound propagation. Unfortunately, this simple decomposition hinders propagation. It can be bounds consistent whereas bounds consistency on the corresponding NVALUE constraint detects disentailment.

Theorem 6.50 *Enforcing bounds consistency on* NVALUE *is stronger than bounds consistency on its decomposition into* (6.67) *to* (6.69).

Proof: Clearly, bounds consistency on NVALUE is at least as strong as bounds consistency on the decomposition. To show strictness, consider $X_1 \in \{1, 2\}$, $X_2 \in \{3, 4\}$, $B_j \in \{0, 1\}$ for $1 \leq j \leq 4$, and $N = 1$. Constraints (6.67) to (6.69) are bounds consistent . However, the corresponding NVALUE constraint has no bound support and thus enforcing bounds consistency on it detects disentailment. $\diamond$

We observe that enforcing domain consistency instead of bounds consistency on constraints (6.67)–(6.69) in the example of the above proof still does not prune any value.

6.9.4 Other related work

In [BHH+05] and [BHH+06a], Bessiere *et al.* consider a number of different methods to compute a lower bound on the number of values used by a set of variables. One of their methods is based on a simple linear relaxation of the minimum hitting set problem. This gives a propagation algorithm that achieves a level of consistency strictly stronger than bounds consistency on the NVALUE constraint. Cheaper approximations have also been proposed based on greedy heuristics and an approximation for the independence number of the interval graph which is due to Turán. Recently, an efficient propagation algorithm was proposed for the increasing NVALUE constraint which is a NVALUE over an ordered set of variables [BHLP10].

6.9.5 Experimental results

To evaluate performance of our decompositions, we performed experiments on two sets of problems containing the ATMOSTNVALUE constraint. We used the same problems as in the only previous experimental comparison of propagators for the ATMOSTNVALUE constraint [BHH+06a]. We ran experiments with Ilog 6.2 solver on an Intel Xeon 4 CPU, 2.0 Ghz, 4Gb RAM.

6.9.5.1 Dominating set of the Queen's graph

The problem is to put the minimum number of queens on a $n \times n$ chessboard, in such a way that each square of the chessboard either contains a queen or is attacked by a queen. This is equivalent to the dominating set problem in the Queen's graph. Each vertex in the Queen's graph corresponds to a square of the chessboard and there exists an edge between two vertices if and only if a queen from one square can attack a queen from the other square. To model the problem, we use a variable X_i for each square, and values from 1 to n^2 and post a single ATMOSTNVALUE$([X_1, \ldots, X_{n^2}], N)$ constraint. The value j belongs to $D(X_i)$ if and only if there exists an edge (i, j) in the Queen's graph or $j = i$. We use the minimum domain variable ordering and the lexicographical value ordering. For $n \leq 120$, all minimum dominating sets for the Queen's problem are either of size $\lceil n/2 \rceil$ or $\lceil n/2 + 1 \rceil$ [OW01]. We therefore only solved instances for these two values of N.

We compare our decomposition with two simple decompositions of the ATMOSTNVALUE constraint. The first decomposition is the one described in Section 6.9.3 except that in constraint (6.69), we replace "$=$" by "$\leq$". We denote this decomposition $Occs$. The second decomposition is similar to the first one, but we use the cardinality variables of a GCC constraint to keep track of the used values. We call this decomposition $Occs_{gcc}$. In the third decomposition, as explained in Section 6.9.2.2, we channel the variables X_i directly to the pyramid variables M_{lu} to avoid introducing many auxiliary variables A_{ilu} and we add the redundant constraint $\sum_{i=1}^{n^2} M_{ii} = M_{1,n^2}$ to the decomposition to speed up the propagation across the pyramid. Finally, we re-implemented the ternary sum constraint $Z = X + Y$ in Ilog. This gave us about 30% speed up. We call this decomposition $Pyramid$.

Results are presented in Table 6.5. Our decomposition performs better than the other two decompositions, both in runtime and in number of backtracks. It should be pointed out that our results are comparable with the results for the ATMOSTNVALUE bounds consistency propagator from [BHH+06a]. Whilst our decomposition is not as efficient as the best

Table 6.5: Backtracks and runtime (in seconds) to solve the dominating set problem for the Queen's graph.

n	N	$Occs$		$Occs_{gcc}$		$Pyramid$	
		backtracks	time	backtracks	time	backtracks	time
5	3	34	0.01	34	0.06	**7**	**0.00**
6	3	540	0.16	540	2.56	**118**	**0.03**
7	4	195,212	84.70	195,212	1,681.21	**83,731**	**21.21**
8	5	390,717	255.64	390,717	8,568.35	**256,582**	**89.30**

results presented in that paper, our decomposition was on the other hand much easier to implement.

6.9.5.2 Random binary *CSP* problems

We also reproduced the set of experiments on random binary *CSP* problems from [BHH$^+$06a]. Problem instances are generated according to model B [Pro94]. When $n \to \infty$ this model can be trivially unsatisfiable if $t/(d^2) > 1/d$ [GMP$^+$01]. In our experiments, the number of variables for hard problems is small, $n \leq 100$, and the presence of flaws is unlikely.

These problems can be described by four parameters. The number of variables n, the domain size d, the number of binary constraints m and the number of forbidden tuples in each binary constraint. The first three classes are harder problems as they are close to phase transition in satisfiability. The last two classes are under-constrained problems. We add a single AtMostNValue constraint over all variables to bound the number of values N that can be used in a solution.

As in [BHH$^+$06a], we generated 500 instances for each of the following 5 classes of random binary *CSP*s:

- class A : $n = 100, d = 10, m = 250, t = 52, N = 8$
- class B : $n = 50, d = 15, m = 120, t = 116, N = 6$
- class C : $n = 40, d = 20, m = 80, t = 240, N = 6$
- class D : $n = 200, d = 15, m = 600, t = 85, N = 8$
- class E : $n = 60, d = 30, m = 150, t = 350, N = 6$

All instances are solved using the minimum domain variable ordering heuristic, a lexicographical value ordering and a timeout of 600 seconds. We use the same decompositions

of the ATMOSTNVALUE constraint as in the experiments with the dominating set of the Queen's graph. Results are given in Table 6.6. We see that our decomposition is faster than the other two decompositions and solves more instances.

Table 6.6: Randomly generated binary *CSP*s with an ATMOSTNVALUE constraint. For each class we give two lines of results. Line 1: number of instances solved in 600 sec (#solved), average backtracks on solved instances (#bt), average time on solved instances (time). Line 2: number of instances solved by all methods, average backtracks on these instances, average time on these instances.

		$Occs$			$Occs_{gcc}$			$Pyramid$		
$Class$		#solved	#bt	time	#solved	#bt	time	#solved	#bt	time
A	total solved	453	139,120	111.2	79	8,960	302.8	**462**	**148,673**	**105.7**
	solved by all	79	8,960	7.1	79	8,960	302.8	79	**8,739**	**6.3**
B	total solved	473	228,757	113.5	125	37,377	292.9	**491**	**235,715**	**94.9**
	solved by all	125	7,377	17.6	125	37,377	292.9	125	**32,110**	**12.2**
C	total solved	479	233,341	110.3	156	37,242	290.3	**490**	**224,802**	**84.2**
	solved by all	156	37242	16.4	156	37,242	290.3	156	**31,715**	**11.1**
D	total solved	482	8,306	6.0	456	207	14.9	**489**	**13,776**	**9.0**
	solved by all	456	**207**	**0.2**	456	207	14.9	456	625	0.4
E	total solved	**500**	331	0.3	**500**	331	5.1	**500**	**174**	**0.1**
	solved by all	500	331	0.3	500	331	5.1	500	**174**	**0.1**
TOTALS										
Total solved/tried		2,387/2,500			1,316/2,500			**2,432**/2,500		
Avg time for solved		67.0			87.5			**58.0**		
Avg backtracks for solved		120,303			8,700			**123,931**		

These experiments demonstrate that this new decomposition is efficient in practice. Of course, if the toolkit contains a specialised bounds consistency propagator for the NVALUE constraint, we will probably do best to use this. However, when the toolkit lacks such a propagator (as is often the case), it is reasonable to try out our decomposition.

6.10 Conclusions

We have studied a number of decompositions of the ALL-DIFFERENT, OVERLAPPINGALLDIFF, GCC and NVALUE constraint. We showed that a simple decomposition can simulate the bounds consistency propagator for these constraints with comparable time complexity to the best known propagators. For ALL-DIFFERENT, GCC and NVALUE constraints we proposed decompositions that enforce range consistency. These decompositions are interesting for a number of reasons. First, we can easily

incorporate them into other solvers. Second, the decompositions provide other constraints with access to the state of the propagator. For example, in the decompositions of the PERMUTATION constraint, propagation is improved by sharing intermediate variables. Third, these decompositions provide a fresh perspective on propagation of global constraints. For instance, our results suggest that it may pay off to focus propagation and nogood learning on relatively small Hall intervals. Finally, these decompositions raise an important question: are there propagation algorithms that cannot be efficiently simulated using decompositions? We consider this question in the next chapter.

Our other theoretical contribution is a reformulation of the consistency checker for the ALL-DIFFERENT constraint into a negative cycle detection problem which is a similar result to the SEQUENCE constraint. In fact, we used an interesting connection between ALL-DIFFERENT and SEQUENCE constraints to build this reformation.

Finally, we implemented our decompositions and showed that they outperform existing decompositions for the corresponding constraints. However, in spite of theoretically similar worse case behaviour, they are slower compared to the best known propagators for each constraint if such a propagator exists. However, as we pointed out above, if a constraint solver does not provide a bounds consistency and a range consistency propagator for one of these constraints, our decompositions can be used to replace a propagator.

Chapter 7

Limitations of decompositions of global constraint

7.1 Introduction

In Chapters 4–6 we presented decompositions of global constraints propagators. One important question that arises is 'What are the limitations of this approach?'. Which global constraint propagators can be effectively decomposed using simple encodings, like the one used in the previous Chapters 4–6? We show that results from circuit complexity can be used to resolve this question. Our main result is that there is a polynomial sized decomposition of a constraint propagator into CNF if and only if the propagator can be computed by a polynomial size monotone Boolean circuit. It follows that bounds on the size of monotone Boolean circuits give bounds on the size of decompositions of global constraints into CNF. For instance, a super-polynomial lower bound on the size of a Boolean circuit for perfect matching in a bipartite graph gives a super-polynomial lower bound on the size of a CNF decomposition of the domain consistency propagator for the ALL-DIFFERENT constraint. The limitation of our result is that it holds for constraint propagator decompositions where variables domains are represented using the direct encoding. This is a matter of future research to investigate other encodings, like the logarithmic encoding. However, our results directly extend to decompositions into *CSP* constraints of bounded arity with a relation given in extension since such decompositions can be translated into clauses of polynomial size [BH03]. Each constraint that we used in our decomposition belongs to this class or can be decomposed into bounded arity constraints without hindering propagation. The monotone circuit complexity results are thus useful in understanding the limits of what we can achieve with decompositions.

7.2 Background

In this section we provide some background on Boolean circuits and monotone Boolean functions. A Boolean circuit S is a directed acyclic graph (DAG). Each source vertex of the DAG is an *input gate* and the unique sink of the DAG is the output gate. Each non-input vertex is labelled with a logical connective, such as and ($\wedge$), or ($\vee$) and not ($\neg$). An *input* $\mathbf{b}$ to the circuit is an assignment of a *value* 0 or 1 to each input gate.[1] The value of a non-input gate is computed by applying the connective that it is labelled with the values of its ancestor gates. The value of the circuit $S(\mathbf{b})$ is the value of its output gate. Any polynomial time decision algorithm can be encoded as a Boolean circuit of polynomial size for a fixed length input [PS82].

In this work, we will use a restriction of Boolean circuits to $\wedge$-gates and $\vee$-gates, called *monotone circuits* . The family of functions that are computable by monotone circuits is exactly all the monotone Boolean functions. Note that there exist families of polynomial time computable monotone Boolean functions such that the smallest monotone circuit that computes them is super-polynomial in size [Raz85].

Definition 7.1 (Monotone Boolean function) *A Boolean function f is monotone if and only if $f(\mathbf{b}) = 0$ implies $f(\mathbf{b}') = 0$ for all $\mathbf{b}' \leq \mathbf{b}$, where $\leq$ is the pairwise vector comparison, i.e., $b'_i \leq b_i$ for all i.*

It is convenient to check monotonicity of a function on the lattice structure of all assignments. We illustrate this on Example 7.1.

Definition 7.2 (Latices) *A* lattice *is a partially ordered set in which any two elements have a unique supremum and infimum.*

Example 7.1 *Consider a monotone function over Boolean variables X_1, X_2, X_3, X_4 with satisfying assignments presented in Table 7.1. We use pairwise vector comparison to order all possible assignments of input variables. With this order relation, assignments form a lattice that is presented in Figure 7.1.*

Satisfying assignments are in gray. As can be seen from the lattice, the function f is monotone because $f(\mathbf{X}) = 0$ implies $f(\mathbf{X}') = 0$ for all $\mathbf{X}' \leq \mathbf{X}$. $\diamond$

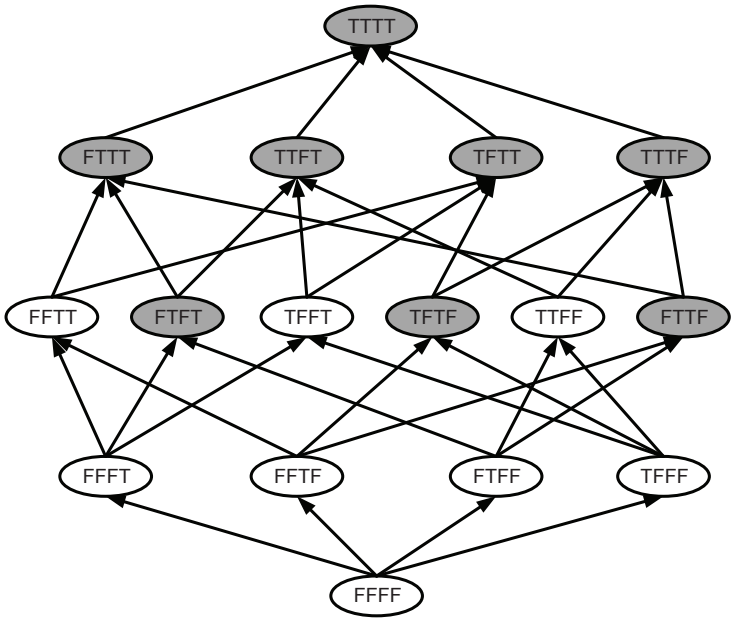
Next we introduce the notion of a partial instantiation of input variables that we use in the following sections.

[1] We use 0 and 1 to distinguish from TRUE and FALSE for *SAT* variables.

Table 7.1: Monotone Boolean function f.

y_i	X_1	X_2	X_3	X_4	f(**X**)
1	T	F	F	T	**1**
2	F	T	T	F	**1**
3	T	F	T	F	**1**
4	F	T	T	T	**1**
5	T	F	T	T	**1**
6	T	T	F	T	**1**
7	T	T	T	F	**1**
8	T	T	T	T	**1**

Figure 7.1 Lattice of a monotone function assignments from Example 7.1. Satisfying assignments are in gray.



Definition 7.3 (Partial instantiation) *A* partial instantiation *of input Boolean variables* **X**, *I, is a set of literals such that $\overline{X}_i \in I$ ($X_i \in I$) if and only if X_i is set to* FALSE*(*TRUE*).*

By $f(X_I) = 0$ we mean that given a partial instantiation I, f evaluates 0 for any instantiation of the unset variables. Similarly, by $f(X_I) = 1$ we mean that f evaluates 1 for any instantiation of the unset variables. To simplify notation we write $f(I)$ instead of $f(X_I)$ when the set of variables is clear from the context.

We also consider a relaxation of partial instantiation that ignores information about variables that are set to TRUE.

Definition 7.4 (FALSE**-only partial instantiation)** *Let I be a partial instantiation of the*

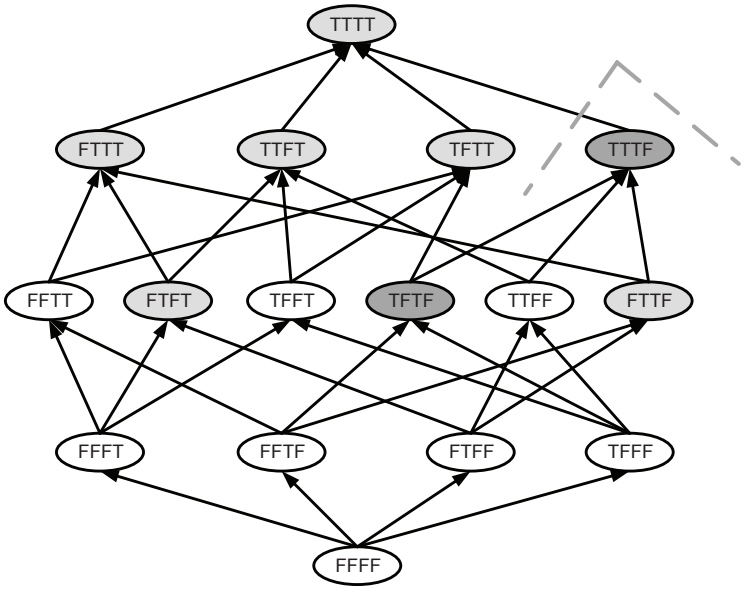
input variables **X**. *The* FALSE*-only partial instantiation* I^F *is the partial instantiation such that* $I^F = \{\overline{X}_i | \overline{X}_i \in I\}$.

Finally, we introduce a mapping between partial instantiations and their extensions to the full instantiation that corresponds to the highest assignment in the lattice.

Definition 7.5 *Let* I *be a partial instantiation of the input variables* **X**. *The top assignment* X^t *w.r.t. a partial instantiation* I *is defined as follows:* $X_i^t(I) =$ FALSE *if* $\overline{X}_i \in I$ *and* $X_i^t(I) =$ TRUE *otherwise.*

We write X^t instead of $X^t(I)$ when the corresponding partial assignment is clear from the context. Clearly, I and I^F have the same top assignment X^t.

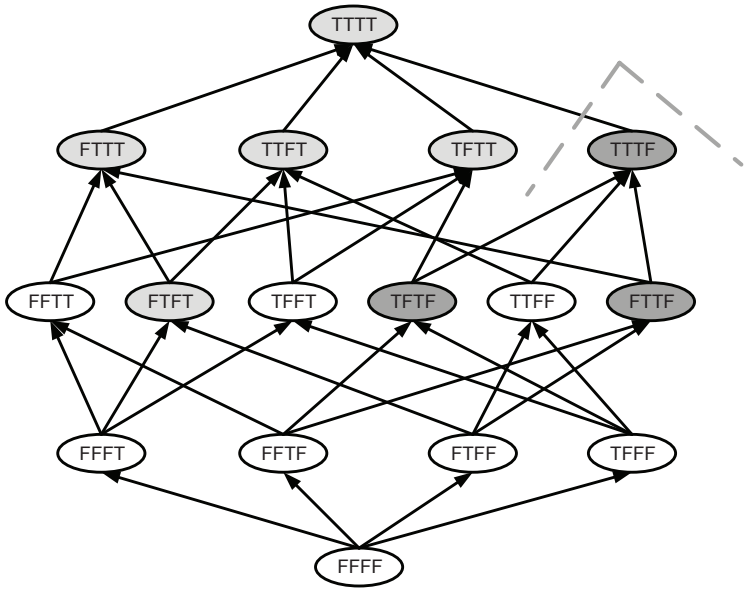
Figure 7.2 Lattice of a monotone function assignments from Example 7.1. A partial instantiation is $I = \{X_1, X_3, \overline{X}_4\}$. All assignments that became incompatible by the partial assignment are in light gray.



Example 7.2 *Consider the monotone function from Example 7.1. Suppose a partial instantiation is* $I = \{X_1, X_3, \overline{X}_4\}$. *Figure 7.2 shows the lattice where we mark light gray all assignments that are incompatible by the partial assignment* I. *The* FALSE*-only partial instantiation* $I^F = \{\overline{X}_4\}$. *The top assignment* $X^t(I)$ *is the assignment* (TRUE, TRUE, TRUE, FALSE) *which is the highest among all possible extensions of* I *and* I^F *in the lattice.* $\diamond$

Lemma 7.1 *Let* I *be a partial instantiation of the input variables and* $X^t(I)$ *is the corresponding top assignment. Then* $f(X^t) = 0$ *if and only if* $f(I) = 0$.

Figure 7.3 Lattice of a monotone function assignments from Example 7.1. A partial instantiation is $I = \{\overline{X}_4\}$. All assignments that are compatible with the partial assignment are in gray.



Proof: If X^t is not a solution then all other extensions X' of I are such that $X' \leq X^t$ are not solutions as well due to monotonicity of the function. Hence, $f(X') = 0$.

Conversely, if $f(I) = 0$, then the function is 0 for all possible extensions, including X^t, by definition. $\diamond$

Finally, we show that I and I^F contain equivalent information for the monotone Boolean function to decide whether there exists a solution that extends I or the corresponding I^F.

Lemma 7.2 *Let I be a partial instantiation of the input variables* **X** *and I^F be the corresponding* FALSE*-only partial instantiation. Then $f(I) = 0$ if and only if $f(I^F) = 0$.*

Proof: Follows from the fact that I and I^F have the same top assignment in the lattice. By Lemma 7.1, $f(I) = f(I^F) = 0$. $\diamond$

We summarise results of Lemmas 7.1–7.2 in Table 7.2 (first line, page 228).

In the next section, we show that a monotone Boolean function can be decomposed into a CNF of polynomial size if and only if it can be computed by a polynomial monotone circuit. Based on this result we show a similar result for CNF decompositions of constraint propagators. This allows determining lower bounds on the size of CNF decompositions of many cardinality constraints, including ALL-DIFFERENT and GCC constraints.

7.3 Properties of CNF decompositions

In this section, we formally define a *CNF decomposition* of a monotone Boolean function. We start with a definition of a CNF decomposition that mimics the behaviour of a monotone Boolean function.

Definition 7.6 (CNF_f decomposition of a monotone Boolean function) *A CNF_f decomposition of a monotone Boolean function f is a formula in CNF B_f over variables $\mathbf{X} \cup \mathbf{y}$ such that*

- $\mathbf{X}$ *is the set of* input variables *and* $\mathbf{y}$ *is a set of* auxiliary variables *whose size is polynomial in* $|\mathbf{X}|$.
- *Given a partial instantiation I, unit propagation on B_f produces the empty clause when $f(I) = 0$.*

We also use the notation $B(I)$ for a CNF formula where X_i is replaced with TRUE (FALSE) when $X_i \in I$ ($\overline{X}_i \in I$).

Example 7.3 *To illustrate Definition 7.6, consider the monotone function from Example 7.1. This monotone function can be seen as a* TABLE *constraint. Bacchus [Bac07] decomposes such a* TABLE *constraint into CNF B_f using the following set of clauses:*

$$X_1 \Rightarrow y_3 \vee y_5 \vee y_6 \vee y_7 \vee y_8 \qquad \overline{X}_1 \Rightarrow y_1 \vee y_2 \vee y_4$$
$$X_2 \Rightarrow y_1 \vee y_2 \vee y_4 \vee y_5 \vee y_7 \vee y_8 \qquad \overline{X}_2 \Rightarrow y_3 \vee y_5$$
$$X_3 \Rightarrow y_3 \vee y_4 \vee y_5 \vee y_7 \vee y_8 \qquad \overline{X}_3 \Rightarrow y_1 \vee y_6$$
$$X_4 \Rightarrow y_1 \vee y_4 \vee y_5 \vee y_6 \vee y_8 \qquad \overline{X}_4 \Rightarrow y_2 \vee y_3 \vee y_7$$

$$y_1 \Rightarrow \overline{X}_1 \;\; y_1 \Rightarrow X_2 \;\; y_1 \Rightarrow \overline{X}_3 \;\; y_1 \Rightarrow X_4$$
$$y_2 \Rightarrow \overline{X}_1 \;\; y_2 \Rightarrow X_2 \;\; y_2 \Rightarrow X_3 \;\; y_2 \Rightarrow \overline{X}_4$$
$$y_3 \Rightarrow X_1 \;\; y_3 \Rightarrow \overline{X}_2 \;\; y_3 \Rightarrow X_3 \;\; y_3 \Rightarrow \overline{X}_4$$
$$y_4 \Rightarrow \overline{X}_1 \;\; y_4 \Rightarrow X_2 \;\; y_4 \Rightarrow X_3 \;\; y_4 \Rightarrow X_4$$
$$y_5 \Rightarrow X_1 \;\; y_5 \Rightarrow \overline{X}_2 \;\; y_5 \Rightarrow X_3 \;\; y_5 \Rightarrow X_4$$
$$y_6 \Rightarrow X_1 \;\; y_6 \Rightarrow X_2 \;\; y_6 \Rightarrow \overline{X}_3 \;\; y_6 \Rightarrow X_4$$
$$y_7 \Rightarrow X_1 \;\; y_7 \Rightarrow X_2 \;\; y_7 \Rightarrow X_3 \;\; y_7 \Rightarrow \overline{X}_4$$
$$y_8 \Rightarrow X_1 \;\; y_8 \Rightarrow X_2 \;\; y_8 \Rightarrow X_3 \;\; y_8 \Rightarrow X_4$$

where $\mathbf{y} = \{y_i\}$, $i \in \{1, \ldots, 8\}$ are auxiliary variables that correspond to satisfying tuples. Suppose the X_1, X_2 and X_4 are set to FALSE*, so that the partial instantiation is $I = \{\overline{X}_1, \overline{X}_2, \overline{X}_4\}$. This instantiation forces the variables y_i, $i = 1, \ldots, 8$ to* FALSE*,*

which generates an empty clause. This falsifies the first clause $X_1 \Rightarrow y_3 \vee y_5 \vee y_6 \vee y_7 \vee y_8$.

$\diamond$

Next we introduce an alternative to CNF_f definition of a CNF decomposition of a monotone Boolean function that does not generate an empty clause when a partial assignment cannot be extended to a solution. Instead, it records this result in an auxiliary Boolean variable. Definition 7.7 is equivalent to Definition 7.6 as Theorem 7.1 will show.

Definition 7.7 (CNF_C **decomposition a monotone Boolean function)** *A* CNF_C *decomposition of a monotone Boolean function* $f(\mathbf{X})$ *is a CNF* B_C *over variables* $\mathbf{X} \cup \mathbf{y} \cup \{z\}$ *such that*

- $\mathbf{X}$ *is a set of input variables and* $\mathbf{y}$ *is a set of auxiliary variables whose size is polynomial in* $|\mathbf{X}|$ *and* z *is the* output variable.
- *Unit propagation on* B_C *never forces any variable from* $\mathbf{X}$ *or generates the empty clause if no variable in* $\mathbf{y}$ *is set externally to* B_C*, i.e., every variable* $y \in \mathbf{y}$ *is either unset or forced by a clause in* B_C.
- *Given a partial instantiation* I, z *is set to* FALSE *by unit propagation if and only if* $f_C(I) = \emptyset$.

Example 7.4 *Consider the monotone Boolean function from Example 7.3. We construct a* CNF_C *decomposition of the monotone function* B_C*, using the* CNF_f *decomposition of the monotone function,* B_f*. The clauses that cause pruning of input variable domains are removed and the last clause is augmented with the output variable* z *to avoid generation of the empty clause in case of failure:*

$$y_1 \Rightarrow \overline{X}_1 \;\; y_1 \Rightarrow X_2 \;\; y_1 \Rightarrow \overline{X}_3 \;\; y_1 \Rightarrow X_4$$
$$y_2 \Rightarrow \overline{X}_1 \;\; y_2 \Rightarrow X_2 \;\; y_2 \Rightarrow X_3 \;\; y_2 \Rightarrow \overline{X}_4$$
$$y_3 \Rightarrow X_1 \;\; y_3 \Rightarrow \overline{X}_2 \;\; y_3 \Rightarrow X_3 \;\; y_3 \Rightarrow \overline{X}_4$$
$$y_4 \Rightarrow \overline{X}_1 \;\; y_4 \Rightarrow X_2 \;\; y_4 \Rightarrow X_3 \;\; y_4 \Rightarrow X_4$$

$$y_5 \Rightarrow X_1 \;\; y_5 \Rightarrow \overline{X}_2 \;\; y_5 \Rightarrow X_3 \;\; y_5 \Rightarrow X_4$$
$$y_6 \Rightarrow X_1 \;\; y_6 \Rightarrow X_2 \;\; y_6 \Rightarrow \overline{X}_3 \;\; y_6 \Rightarrow X_4$$
$$y_7 \Rightarrow X_1 \;\; y_7 \Rightarrow X_2 \;\; y_7 \Rightarrow X_3 \;\; y_7 \Rightarrow \overline{X}_4$$
$$\overline{y}_1 \wedge \overline{y}_2 \wedge \overline{y}_3 \wedge \overline{y}_4 \wedge \overline{y}_5 \wedge \overline{y}_6 \wedge \overline{y}_7 \wedge \overline{y}_8 \Rightarrow \overline{z}$$

Consider the partial instantiation where X_1, X_2 *and* X_4 *are set to* FALSE*, so* $I = \{\overline{X}_1, \overline{X}_2, \overline{X}_4\}$*. This forces all auxiliary variables* y_i*,* $i = 1, \ldots, 8$ *to be* FALSE*. Therefore,*

the output variable z is forced to FALSE*, signalling that the monotone function f does not have a satisfying assignment under current partial instantiation I.* $\diamond$

In example 7.4, we transformed a CNF_f decomposition of monotone function from example 7.3 into a CNF_C decomposition of monotone function in an ad-hoc manner. The next theorem shows that this can be done in a generic way.

Theorem 7.1 *There exists a polynomial time and space conversion between a CNF_f and CNF_C decompositions of a monotone Boolean function f.*

Proof: We recall that we denote B_f and B_C CNF_f and CNF_C decompositions of a monotone Boolean function f, respectively. We assume that formulae are given in 3-CNF form. We can convert any CNF formula to 3-CNF, increasing its size by at most a linear factor and without hindering unit propagation [GJ90, section 3.1.1].

($\rightarrow$) We construct B_C as a transformation of f such that the output variable z of B_C is FALSE if and only if unit propagation on B_f produces the empty clause.

Let the set of clauses of B_C be $c_1 \dots c_m$. For each variable $p \in \mathbf{X} \cup \mathbf{y}$, we introduce 2 variables p_t and p_f in C_C so that p_t and p_f are true if p is forced to TRUE or FALSE, respectively:

$$p \implies p_t \qquad \overline{p} \implies p_f \tag{7.1}$$

Then, we simulate unit propagation for each clause c_k by replacing it with 3 implicationsthat contain the variables p_t and p_f rather than p. For example, to simulate unit propagation for the clause $c_1 = (p, q, \overline{r})$, we replace it with

$$p_f \wedge q_f \implies r_f \quad p_f \wedge r_t \implies q_t \quad q_f \wedge r_t \implies p_t \tag{7.2}$$

Unit propagation on Equation (7.2) can never derive the empty clause, because the true and false values of p are encoded in different variables p_t and p_f, which may be true simultaneously. When this happens, unit propagation on C_P would generate the empty clause, therefore we must set the output variable z to FALSE, using the following clauses:

$$p_t \wedge p_f \implies \overline{z} \tag{7.3}$$

The union of the clauses (7.1), (7.2) and (7.3) is a CNF_C decomposition of f_C with size $O(|\mathbf{X} \cup \mathbf{y}| + |C_P|) = O(|C_P|)$, therefore the transformation is polynomial.

($\leftarrow$) This direction is trivial, as we can extend a B_C with an extra clause (z). $\diamond$

Since all sizes of CNF decompositions, B_f and B_C, that we introduced in this section are polynomially equivalent, in the remainder of this paper we only prove results for the

CNF_C decomposition. We also omit the subscript C from CNF_C to simplify notation as we always assume that CNF decomposition of a monotone function is defined by Definition 7.6.

We show a similar result to Lemma 7.2 for a CNF decomposition of monotone function f. As in the case of a monotone function, we show that the information about variables that are set to TRUE is not important for a CNF decomposition of monotone function f.

Lemma 7.3 *Let I be a partial instantiation of the input variables* **X** *of B_C and I^F be the corresponding* FALSE*-only partial instantiation. Then $B_C(I)$ sets the output variable z to* FALSE *if and only if $B_C(I^F)$ sets z to* FALSE.

Proof: Follows from Lemma 7.2 and the correctness of the CNF decompositions. $\diamond$

7.4 Equivalence to monotone circuits

In this section, we show our main result, which establishes a connection between CNF decompositions of a monotone function and circuit complexity.

Theorem 7.2 *A monotone function f can be decomposed to a CNF of polynomial size if and only if it can be computed by a monotone circuit of polynomial size.*

The proof of theorem 7.2 is constructive. We will first show the reverse direction, using the Tseitin encoding [TSW83] of a monotone circuit.

Definition 7.8 (Tseitin encoding of a Boolean circuit) *The Tseitin encoding of a circuit S into clausal form has one propositional variable for each input of S and a gate variable y_g for each gate of S. W.l.o.g, we assume all gates have fan-in 2. For each $\wedge$-gate g with inputs X_1, X_2, the Tseitin encoding contains the clauses $(X_1, \overline{g})$, $(X_2, \overline{g})$, $(\overline{X}_1, \overline{X}_2, g)$ and for each $\vee$-gate it contains the clauses $(\overline{X}_1, g)$, $(\overline{X}_2, g)$, $(X_1, X_2, \overline{g})$.*

Proposition 7.1 (Correctness of Tseitin encoding of a Boolean circuit) *Given any complete instantiation of the input variables, unit propagation on the Tseitin encoding sets the variable corresponding to the output gate of S to* TRUE *if the circuit computes 1 and to* FALSE *otherwise.*

We define the mapping between a partial instantiation I and inputs of Boolean circuit. The inputs of a Boolean circuit correspond to a complete instantiation of the inputs. Hence, we have to select one full instantiation among all possible extensions of I.

Table 7.2: Equivalence between a monotone function, its CNF decomposition and the corresponding monotone circuit.

$f(I^F)$	$\Leftrightarrow_{Lemma\ 7.1}$	$f(I)$	$\Leftrightarrow_{Lemma\ 7.2}$	$f(X^t)$
		$\Uparrow$		
		correctness of CNF decomposition		
		$\Downarrow$		
$B_C(I^F)$	$\Leftrightarrow_{Lemma\ 7.3}$	B_C (I)	$\Leftrightarrow_{monotonicity\ of\ f}$	$B_C(X^t)$
		$\Uparrow$		
		Corollary 7.1		
		$\Downarrow$		
$S_C(\mathbf{b}(I^F))$	$\Leftrightarrow_{\mathbf{b}(I^F)=X^t}$	$S_C(\mathbf{b}(I))$	$\Leftrightarrow_{\mathbf{b}(I)=X^T}$	$S_C(\mathbf{b}(X^t))$

Definition 7.9 (Mapping between a partial instantiation and a Boolean circuit input) *Let f be a monotone Boolean function computed by a Boolean circuit S_C. Let I be a partial instantiation of the input variables $\mathbf{X}$ of B_C and $\mathbf{b}(I)$ be the corresponding input to S_C. We set inputs $\mathbf{b}(I)$ to be equal to the top assignment of I, $X^t(I)$, $b_i(I) = X^t_i(I)$, $i = 1, \ldots, |b|$.*

We show that the function is unsatisfiable under the partial assignment if and only if the circuits output 0 on the corresponding input $\mathbf{b}(I)$ as the following lemma shows. This lemma mirrors Lemma 7.1 for inputs of monotone Boolean circuit.

Lemma 7.4 *Let I be a partial instantiation of the input variables $\mathbf{X}$ of f and $\mathbf{b}(I)$ is the corresponding input of a circuit. Then $f(I) = 0$ if and only if $S_C(\mathbf{b}(I)) = 0$.*

Proof: If $f(I)$ evaluates to 0 then it will evaluate to 0 for any possible extension of I' of I, including $X^t(I)$. As $\mathbf{b}(I) = X^t(I)$, we get that $S_C(\mathbf{b}(I)) = 0$.

Consider the reverse direction. Suppose $S_C(\mathbf{b}(I)) = 0$. Hence, the assignment $X^t(I) = \mathbf{b}(I)$ is not a solution of f. As $X^t(I)$ is the top assignment in the lattice among all possible extensions of I and f is a monotone function, we get that $f(I) = 0$. $\diamond$

Next we consider how to construct a CNF decomposition of a monotone function. Suppose that a monotone function f can be encoded into a monotone circuit S_C of polynomial size. The Tseitin encoding of S_C turns out to be a CNF decomposition of f_C. This is a direct consequence of the following lemma.

Lemma 7.5 *Let S_C be a monotone circuit, T_C be its Tseitin encoding and I be a partial instantiation of the input variables $\mathbf{X}$. Then, T_C is a CNF decomposition of the function computed by S_C, i.e., unit propagation on T_C with I forces the output variable z to* FALSE *if and only if $S_C(\mathbf{b}(I)) = 0$.*

Proof: ($\rightarrow$) This follows from the correctness of the Tseitin encoding (Proposition 7.1).

($\leftarrow$) Suppose that $S_C(\mathbf{b}(I)) = 0$, but the output variable z is not forced to FALSE by unit propagation under I. Consider a instantiation $X^t(I)$ that corresponds to I. Let $y_g \in \mathbf{y} \cup \{z\}$ be an auxiliary gate variable that is unset under $X^t(I)$. Since T_C is an encoding of the monotone circuit S_C, y will be set to TRUE under $X^t(I)$. To see this, observe that an unset gate variable will be made TRUE if its unset input variables are set to TRUE. Consider now a gate g' at depth 1 that has only inputs of the circuit as inputs. When we set all unset input variables to true in $X^t(I)$, the variable $y_{g'}$ will be set to TRUE. Applying this argument recursively, we get that the unset variable z that corresponds to the output gate will also be set to true.

By the correctness of the Tseitin encoding, $S_C(\mathbf{b}) = 1$, a contradiction. $\diamond$

As the Tseitin encoding of a circuit S_C is a CNF decomposition of the corresponding Boolean function, we use the same notation, B_C, to denote the Tseitin encoding.

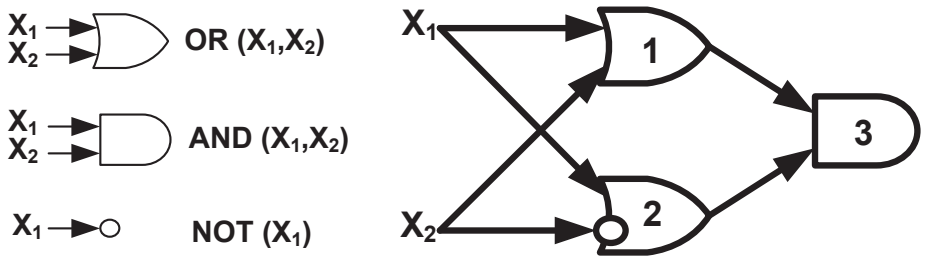
Corollary 7.1 *Let S_C be a monotone circuit and B_C be its Tseitin encoding. Let I be a partial instantiation of the input variables $\mathbf{X}$ of B_C. Then, unit propagation on B_C with I forces the output variable z to* FALSE *if and only if $S_C(\mathbf{b}) = 0$, for all $\mathbf{b}$ where $\mathbf{b}$ is the input to S_C that corresponds to any extension of I to a complete instantiation $X^t(I)$.*

Proof: This follows from Lemma 7.5 and the fact that S_C is a monotone circuit. $\diamond$

We summarise the results of Lemmas 7.3 and Corollary 7.1 in Table 7.2.

Interestingly, Lemma 7.5 cannot be generalised to non-monotone Boolean circuits. The next example shows that there exists a non-monotone Boolean circuit S that computes a monotone function, and a partial instantiation I with $\mathbf{b}$ the corresponding input to S, such that $S(\mathbf{b}) = 0$ but unit propagation on the Tseitin encoding of S under the instantiation I does not set the output variable to FALSE.

Figure 7.4 A circuit whose Tseitin encoding is incomplete.



Example 7.5 *Consider the non-monotone circuit S shown in Figure 7.4. Note that S computes a monotone function.*

The Tseitin encoding of S introduces three Boolean variables g_1, g_2 and g_3 for the gates OR_1, OR_2 and AND_3, respectively, and the clauses $(\overline{X}_1, g_1)$, $(\overline{X}_2, g_1)$, $(\overline{g}_1, X_1, X_2)$, $(\overline{X}_1, g_2)$, (X_2, g_2), $(\overline{g}_2, X_1, \overline{X}_2)$, $(\overline{g}_3, g_1)$, $(\overline{g}_3, g_2)$, $(\overline{g}_1, \overline{g}_2, g_3)$.

Now suppose that $I = \{\overline{X}_1\}$. Then, by Definition 7.9, $\mathbf{b} = \{X_1 = 0, X_2 = 1\}$ and $S(\mathbf{b}) = 0$. Since S computes a monotone function, all possible extensions of $\mathbf{X}$ evaluate to 0. But in the Tseitin encoding, setting X_1 to FALSE *does not make any clauses unit, therefore unit propagation does not set g_3 to* FALSE. $\diamond$

We now show the forward direction of Theorem 7.2: every CNF decomposition B_C of a monotone function f can be converted to a monotone circuit that computes f with at most a polynomial increase in size.

The transformation from a CNF decomposition B_C to monotone circuit exploits two properties of CNF decompositions, namely, that only positive literals of input variables appear in B_C, and that unit propagation only makes auxiliary variables FALSE. We show the former property in Lemma 7.6 and the latter in Lemma 7.7.

Lemma 7.6 *Let B_C be a CNF decomposition of a monotone Boolean function f. There exists a polynomial size CNF decomposition $B_C{}'$ of f such that negative literals of the input variables do not appear in any clause in $B_C{}'$.*

Proof: Let I be a partial instantiation of the input variables such that unit propagation on $B_C(I)$ sets z to FALSE. By Lemma 7.3, $B_C(I^F)$ sets z to FALSE. By definition, B_C never forces any literal of an input variable. As I^F does not contain literals $X_i =$ TRUE, then a clause that contain negative literals of the input variables does not become unit during unit propagation on $B_C(I^F)$. Hence, these clause are not necessary to derive that z to FALSE. Therefore, we just remove these clause from $B_C(I)$ to obtain $B_C{}'$. $\diamond$

The next step is to show that we can transform a CNF decomposition so that each auxiliary variable is unset or FALSE for *all* inputs that make the output variable FALSE. The transformation is a renaming of the auxiliary variables. Lemma 7.7 describes the property that allows this transformation.

Lemma 7.7 *Let B_C be a CNF decomposition of a monotone function f over the variables $\mathbf{X} \cup \mathbf{y} \cup \{z\}$, I_1, I_2 be partial instantiations such that unit propagation on B_C forces z to* FALSE *under both I_1 and I_2. For any variable $y \in \mathbf{y}$, if y is forced to* FALSE *(*TRUE*) by unit propagation under I_1 then it is not forced to* TRUE *(*FALSE*) by unit propagation under I_2.*

Proof: Let a variable y be forced to TRUE by unit propagation under I_1 and to FALSE under I_2, but z is FALSE under both I_1 and I_2. W.l.o.g we can assume that I_1 and I_2 do not

contain information about variables assigned to TRUE (Lemma 7.3). Consider the partial instantiation I such that if a variable $X_i \in \mathbf{X}$ is FALSE in either I_1 or I_2, it is also FALSE in I, otherwise it is unset. Since I fixes a superset of the literals that are fixed in either I_1 or I_2, all clauses that became unit by either I_1 or I_2 will also be unit in I. Therefore, unit propagation under I will force at least the union of the sets of literals forced by I_1 and I_2. This means that unit propagation under I will make both y and $\overline{y}$ TRUE, which generates the empty clause. This is a contradiction, as B_C can never produce the empty clause, by Definition 7.7. $\diamond$

Corollary 7.2 *A CNF decomposition B_C of a monotone function f over variables $\mathbf{X} \cup \mathbf{y} \cup \{z\}$, can be polynomially converted into a decomposition $B_C{}'$ of f such that every variable in $\mathbf{y}$ is either unset or* FALSE *when z is* FALSE.

Proof: We construct $B_C{}'$ from B_C by flipping in every clause the polarity of each auxiliary variable y such that there exists a partial instantiation I such that unit propagation on $B_C(I)$ sets y to TRUE and z is FALSE. $\diamond$

Lemma 7.6 and Corollary 7.2 allow us to precisely characterise the form of the clauses in a CNF decomposition.

Corollary 7.3 *Let B_C be a CNF decomposition of a monotone function f. The variables of B_C can be renamed so that each clause has exactly one negative literal.*

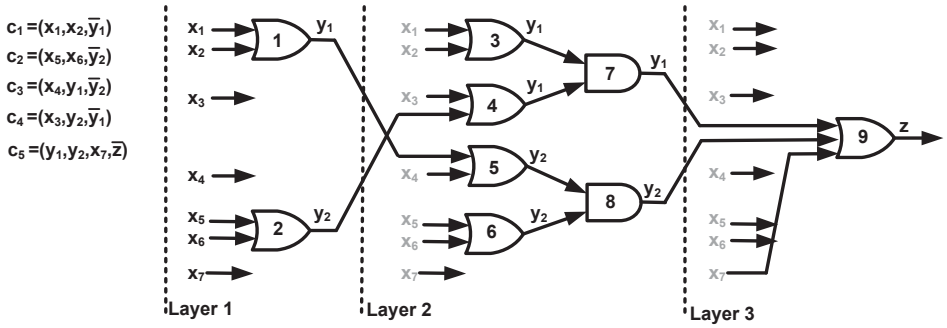
Proof: By Lemma 7.6, all input variables are positive literals in the decomposition and by Definition 7.7 they are never forced by unit propagation on B_C. In addition, by Corollary 7.2, we can rename the auxiliary variables so that unit propagation on B_C may only ever set them to FALSE. Then, in any clause that consists of input variables and one auxiliary variable y, y must be negative, otherwise it may be set to TRUE, a contradiction.

Suppose there exists a clause c with two auxiliary variables y_1 and y_2 and both are negative in c. Since neither y_1 nor y_2 can ever be made TRUE, this clause can never become unit and can be ignored. Suppose the literals of both y_1 and y_2 are positive in c. Then, if c becomes unit, it makes one of the auxiliary variables TRUE, a contradiction. Thus, exactly one of the literals of y_1 and y_2 is negative in c. The same reasoning can be extended to clauses with more than two auxiliary variables. $\diamond$

The condition described by Corollary 7.3 is similar to B_C being re-nameable anti-Horn, but is stronger as it requires *exactly* one negative literal in each clause, rather than at most one. This condition allows us to build a monotone circuit from a decomposition, using the construction of the next lemma.

Lemma 7.8 *Let B_C be a CNF decomposition of a monotone function f. Then, there exists a monotone circuit S_C of size $O(n|B_C|)$ that computes f.*

Figure 7.5 Conversion of a CNF decomposition of a consistency checker into a monotone Boolean circuit.



Proof: We assume that B_C is in the form described in Corollary 7.3 because we can always polynomially transform B_C to this form.

The inputs of the circuit correspond to the input variables of B_C. For each input variable X_i of B_C, there exists an input b_i of S_C which is 0 if X_i is FALSE and 1 otherwise. Internal gates of the circuit correspond to auxiliary variables after a certain number of unit propagation steps, using the same mapping.

Circuit construction. We create a circuit with $|\mathbf{y}|$ *layers* $1 \ldots |\mathbf{y}|$. Let $c_1, \ldots, c_m$ be the clauses of B_C. The i^{th} layer of the circuit contains an $\vee$-gate c_j^i for each clause c_j, called *clause gates* and an $\wedge$-gate y_k^i for each auxiliary variable y_k, called *variable gates*. Consider a clause c_j which contains $\overline{y}$ as the sole negative literal (recall that Corollary 7.3 ensures that this is the case), the positive literals of input variables $X_{j_1}, \ldots, X_{j_q}$ and the positive literals of auxiliary variables $y_{j_{q+1}}, \ldots, y_{j_{q+r}}$. The inputs of each gate c_j^i are $b_{j_1}, \ldots, b_{j_q}$ and $y_{j_{q+1}}^{i-1}, \ldots, y_{j_{q+r}}^{i-1}$. Let the clauses with $\overline{y}_k$ as the sole negative literal be $c_{k_1}, \ldots, c_{k_s}$. Then, the inputs of each gate y_k^i are $c_{k_1}^i, \ldots, c_{k_s}^i$. The output of the circuit is $z^{|\mathbf{y}|}$. Note that in this construction the inputs of some the gates may not be defined. This is the case, for example, for the gate c_i^1, where the clause c_i contains the positive literals of some auxiliary variables. If this happens for a clause gate, we omit it, while if it happens for a variable gate, we omit the undefined input. If all the inputs of a variable gate are undefined, we omit the gate.

This construction computes one breadth first application of unit propagation at each layer that works as follows. Let C be a CNF formula and $U = \{(w_i)\}$, $w_i \in \mathbf{X} \cup \mathbf{y} \cup \{z\}$ be a set of unit clauses. By breadth first application of unit propagation we mean a set of resolution steps with unit clauses from U. For example, consider a clause $c =$

$(w_1, \ldots, w_n, w')$ such that $\exists(\overline{w}_i) \in U$, $i = 1, \ldots, n$. Unit propagation derives the unit clause (w') from c using only clauses in U at any resolution step. But if there exists a clause $(-w', w'')$, one breadth first step of the unit propagation will not derive the unit clause (w'').

Induction step. We show by induction that the gate y_r^k outputs 0 if and only if an auxiliary variable y_r is forced to be FALSE after k breadth first steps of unit propagation.

For the first layer, there exist gates only for clauses with no positive literals of auxiliary variables. Suppose unit propagation sets $\overline{y}_r$ to FALSE. Consider any gate c_j which contains the negative literal $\overline{y}_r$ and is used to derive $\overline{y}_r$. All the propositional variables in c_j except y_r have to be FALSE to imply $\overline{y}_r$. Hence, the inputs corresponding to these propositional variables are 0. Thus c_j^1 is 0 if and only if y_r is FALSE after unit propagation of c_j. If many clauses contain the negative literal $\overline{y}_r$, then at least one of them sets y_r to FALSE in one breadth first step if and only if there exists a clause gate that is 0 and is an input to the variable gate y_r^1, which is an $\wedge$-gate and is thus 0.

For the inductive step, assume that the layers $1 \ldots k-1$ simulate $k-1$ breadth first steps of unit propagation and a gate y_r^{k-1} outputs 0 if and only if an auxiliary variable y_r is forced to be FALSE during $k-1$ steps.

Consider the result of the kth step. First, we note that if y_r^{k-1} outputs 0 then y_r^k also outputs 0 by construction. Suppose unit propagation derives $\overline{y}_r$ using a clause $c_{j'}$. In this case, all positive literals, input and auxiliary variables, in $c_{j'}$ are fixed to FALSE. By construction, all inputs that correspond to input variables X_i, $X_i \in scope(c_{j'})$ are set to 0. By the inductive hypothesis, all gates that correspond to auxiliary variables in the scope of $c_{j'}$ also output 0. These outputs are inputs of $c_{j'}^k$. Hence, $c_{j'}^k$ outputs 0 and y_r is forced to FALSE.

To conclude the proof, observe that in the extreme case, unit propagation will set exactly one more literal at every breadth first step, thus after $|\mathbf{y}|$ steps it must either arrive at a fixpoint or set all literals. Since the circuit has $|\mathbf{y}|$ layers, it will correctly compute the result of unit propagation on B_C. Since B_C sets z to FALSE iff f is 0 and the circuit will compute 0 in its output iff B_C sets z to FALSE, we get that the circuit computes f $\diamond$

We illustrate the construction of Lemma 7.8 with an example.

Example 7.6 *Consider the CNF decomposition* $B_C = \{c_1, c_2, c_3, c_4, c_5\}$, *where* $c_1 = (X_1, X_2, \overline{y}_1)$, $c_2 = (X_5, X_6, \overline{y}_2)$, $c_3 = (X_4, y_1, \overline{y}_2)$, $c_4 = (X_3, y_2, \overline{y}_1)$, $c_5 = (y_1, y_2, X_7, \overline{z})$.

We construct a monotone circuit S_C *from* B_C, *(Figure 7.5). For a given instantiation of the input variables, this circuit computes 0 for the corresponding Boolean inputs if and*

only if unit propagation on B_C forces the output variable to FALSE.

The circuit consists of 3 layers, with gates 1 and 2 in the first layer, 3–8 in the second and gate 9 in the third. The gates 1–6 and 9 are clause gates, while gates 7 and 8 are variable gates. A strict application of the construction of Lemma 7.8 would also have variable gates in layers 1 and 3, but we omit them here as they would be single-input gates. Note that in Figure 7.5, inputs are replicated at each layer to reduce clutter.

We note also that the layered construction of Lemma 7.8 is necessary. A circuit that attempts to capture unit propagation on all clauses without using layers would have to contain a cycle between the gates that compute y_1 and y_2, because y_1 would need to be an input of the clause gate c_3 that computes y_2 and y_2 would need to an input of the clause gate c_4 that computes y_1. Constructing a layered circuit allows us to remove such cycles. $\diamond$

The proof of theorem 7.2 is now immediate from Lemmas 7.5 and 7.8.

7.5 Decompositions of global constraint propagator

In this section we consider CNF decompositions of global constraint propagators. We recall the definition of a CNF decomposition propagator from Section 3.4.

Definition 7.10 (CNF Decomposition of a propagator) *A CNF decomposition of a propagation algorithm P_Φ for a global constraint $C(\mathbf{X})$ is a formula in CNF C_P over variables $\mathbf{x} \cup \mathbf{y}$ such that*

- input variables $\mathbf{x}$ *are the propositional representation of $D(\mathbf{X})$ using the direct encoding and $\mathbf{y}$ is a set of auxiliary variables whose size is polynomial in $|\mathbf{x}|$.*
- *$x_{i,j}$ is set to* FALSE *by unit propagation if and only if $X_i = j \notin P_\Phi(\mathbf{D}(\mathbf{X}))$.*
- *Unit propagation on C_P produces the empty clause when $P_\Phi(D(\mathbf{X})) = \emptyset$.*

A constraint propagator $P_\Phi(\mathbf{D}(\mathbf{X}))$ is a monotone function over $\mathbf{D}(\mathbf{X})$. Hence, its CNF decomposition C_P is a monotone Boolean function over Boolean variables $\mathbf{x}$, because Boolean variables in the direct encoding represent the characteristic function of $\mathbf{D}(\mathbf{X})$.

Due to monotonicity of a CNF decomposition over Boolean variables we can restate Theorem 7.2 for a constraint propagator.

Theorem 7.3 *A constraint propagator P_Φ can be decomposed to a CNF of polynomial size if and only if it can be computed by a polynomial monotone circuit of polynomial size.*

Now we use an existing circuit complexity result to show that there is no polynomial size CNF decomposition of the domain consistency propagator for the ALL-DIFFERENT constraint. This also applies to generalisations of ALL-DIFFERENT , such as GCC .

Corollary 7.4 *There is no polynomial sized CNF decomposition of the* ALL-DIFFERENT$([X_1, \ldots, X_n])$ *domain consistency propagator.*

Proof: Regin [Reg94] showed that an ALL-DIFFERENT$([X_1, \ldots, X_n])$ constraint has a solution if and only if the corresponding bipartite variable-value graph has a matching of size n. If the total number of values in the variable domains equals n then we need to find a perfect matching. In addition, every bipartite graph corresponds to the value graph of an ALL-DIFFERENT constraint and domain consistency propagators detect disentailment. Thus, if there exists a polynomial size CNF decomposition of the ALL-DIFFERENT domain consistency propagator, we can construct a monotone circuit that computes whether a bipartite graph has a perfect matching. But Razborov [Raz85] showed that the smallest monotone circuit that computes whether there exists a perfect matching for a bipartite graph is super-polynomial in the number of vertices in the graph. Therefore, the smallest CNF decomposition of the ALL-DIFFERENT domain consistency propagator is super-polynomial in size. $\diamond$

7.6 Conclusions

In this chapter we have shown how the tools of circuit complexity can be used to study decompositions of global propagators into CNF. We show that there is no polynomial size CNF decomposition of the domain consistency propagator for the ALL-DIFFERENT constraint if variables domains are represented using direct encoding. Our results directly extend to decompositions into *CSP* constraints of bounded arity with domains given in extension since such decompositions can be translated into clauses of polynomial size. An interesting next step is to consider the decomposability of constraint propagators into more expressive primitive constraints where domains are represented in logarithmic space via their bounds. *CSP* solvers provide this feature, which is missing in CNF. We conjecture that there exists an equivalence between such *CSP* decompositions of constraint propagators and monotone arithmetic circuits that are generalisations of Boolean monotone circuits to real numbers and gates for addition and multiplication. Since lower bound results on monotone circuits usually transfer to monotone arithmetic circuits, this would imply that the domain consistency propagator for ALL-DIFFERENT cannot be decomposed to constraints that exploit

(exponentially) large domains.

Chapter 8

Conclusion

The thesis defended in this dissertation is that:

> *Efficient propagators for many important global constraints can be developed by reformulating them using decompositions based on linear inequalities and network flows. However, we show that there are also theoretical limits on the efficiency of propagators and decompositions that can simulate them.*

We proposed reformulations of a number of useful global constraints, including SEQUENCE, ALL-DIFFERENT, GCC, GRAMMAR, NVALUE and their generalisations into a set of bounded arity primitive constraints or network flow problems. We analysed the proposed decompositions from theoretical and practical points of view. We proved that our decomposition into network flow or shortest path problems allows constructing the most up-to-date efficient propagator for the time complexity to the best known propagators. We investigated limitations of our approach and showed that the domain consistency filtering algorithm for ALL-DIFFERENT cannot be decomposed in a set of bounded arity constraints of polynomial size without hindering propagation.

Below we summarise the main contributions of this work and significance of the results. Our results support the thesis message above.

Constraint decompositions into primitive constraints. We propose reformulations of a number of global constraints, including SEQUENCE, SLIDINGSUM, weighted GRAMMAR, ALL-DIFFERENT, GCC, ATMOSTNVALUE, and ATLEASTNVALUE into a set of primitive constraints, such as constraints of bounded arity. The results are significant for the following of reasons:

- *Preserving inference.* We show that in many cases complex filtering algorithms can be simulated by simple decompositions, meaning that reasoning with the decomposition does the same pruning as existing monolithic propagators.

- *Maintaining complexity.* The time complexity of our propagators based on decomposition are similar to the time complexity of the best known propagators for the corresponding global constraints.

- *Easy to implement.* Constraint decompositions can be easily added to a new solver avoiding the need to re-implement filtering algorithms for all global constraint. While there are about 300 constraints in the global constraints catalogue, most modern constraint solvers typically support fewer than 20 of them. Using our decompositions, constraint solver developers can easily extend their modelling layer and avoid spending time on implementing inference algorithm for many decomposable constraints.

- *Access to the internal state of the inference algorithm.* Reformulations are based on encoding the theoretical concepts that underlie constraint filtering algorithms, like detection of Hall intervals in the case of the ALL-DIFFERENT constraint. To construct a decomposition, we introduce variables to keep track of these Hall intervals. In contrast, an inference algorithm will infer this information using variable domain processing. The variables that we introduce expose the internal state of the filtering algorithm to the constraint solver. The solver can, potentially, use this information in its branching heuristics or nogood learning. This direction was not investigated in this thesis. Recently, Moore [Moo11] investigated a related research direction. Moore proposed a new c-learning scheme that introduces new variables during learning. He showed an exponential separation between nogood learning [KB05] and c-learning scheme. As decompositions of global constraints introduce new variables these variables can be, potentially, used such a in c-learning scheme.

- *Beyond standard consistency levels.* Our decompositions allow achieving the same inference as the original algorithm. However, they also allow relaxing the level of inference in order to investigate trade-offs between inference and search. For example, we found problems where Hall intervals of large size occur rarely. Hence, we can ignore these intervals, which hinders inference, but makes the reformulation smaller and more efficient.

- *Integration into other search paradigms.* Our decompositions, which use only fixed arity constraints or constraints that can be decomposed into fixed arity constraints,

can be easily integrated into other search paradigms, like *SAT* or *ILP* solvers.

We would like to point out two tradeoffs of using decompositions into primitive constraints versus monolithic propagators. First, the space complexity is equal to the time complexity for the decomposition as we use auxiliary variables to store internal states of propagators. This can be avoided by using intricate monolithic propagators that support smart data-structures and save the space significantly in come case. The second point is that the average time complexity depends on the invocation order of constraints. If a constraint solver is not aware that a set of primitive constraints constitutes a decomposition, the order of their invocation can be different to the best invocation order and hit the worst case complexity.

Constraint decompositions into network flows. We propose new filtering algorithms that are based on network flow or shortest path algorithms for a number of global constraints, including SEQUENCE, generalised SEQUENCE, soft SEQUENCE, SLIDINGSUM and overlapping ALL-DIFFERENT constraints. These results are significant for the following reasons:

- *Efficient algorithms.* To the best of our knowledge, the proposed reformulations allow constructing the most efficient algorithms for a number of global constraints. A reformulation to a network flow or shortest path algorithm enables the use of the whole body of research on graph algorithms to construct filtering algorithms for global constraints.

- *Better understanding of the nature of these constraints.* Reformulation to a problem on a graph gives a better understanding of the nature of the constraint and allows drawing a connection between seemingly unrelated constraints like counting constraint, the ALL-DIFFERENT constraint, the sliding constraint, and the SEQUENCE constraint. This understanding allowed us to build new filtering algorithms for even more expressive constraints, like the overlapping ALL-DIFFERENT constraint.

Limitations of decompositions. We propose a technique for identifying whether a constraint can be polynomially decomposed into CNF so that unit propagation achieves the same amount of inference as a filtering algorithm on the original constraint would do. This result is significant for the following reasons:

- *Connection to circuit complexity.* The non-decomposability result presented in this thesis reveals a connection between the constraint reformulation problem and circuit

complexity results. In particular, we use the lower bounds results on the size of monotone functions computed on monotone circuits. Hence, any new results on the lower bounds of monotone functions can be used to prove results on non-decomposability of global constraints.

- *Resolution of an open question.* Our result resolves the open questions of which global constraints can be effectively propagated using simple encodings [BH03]. In particular, we show that the polynomial time domain consistency filtering algorithm for the ALL-DIFFERENT constraint cannot be encoded into a polynomial size *SAT* formula so that unit propagation in a SAT solver achieves the same amount of inference. The limitation of our result is that we assume direct encoding of variables domains. This shows that there are global constraints on which a constraint solver can be theoretically exponentially more efficient compared to a *SAT* solver.

Limitations on the efficiency of a propagator. We investigate a number of restricted classes of the GRAMMAR constraint. We identify a subclass of context free grammars that permit a faster filtering algorithm compared to the filtering algorithm for an arbitrary context free GRAMMAR constraint and several classes of widely used restricted context free grammars that require cubic time to perform inference in the worst case. These results are significant for the following reasons:

- *Exploration of the space between the first and the second levels of Chomsky hierarchy.* We investigate a wide class of restricted context free grammars and show that even for a very restricted class of simple context-free grammars there exist no filtering algorithm faster than $O(n^3)$.

- *Resolution of an open question.* Our result resolves the open question whether there exists a filtering algorithm for the unambiguous context free GRAMMAR constraint that exploits the unambiguity to achieve better time efficiency compared to an arbitrary context free GRAMMAR constraint [Sel06]. We answer this question negatively.

8.1 Future work

The reformulations presented in this work are already being used in the constraint programming community. In particular, many of our reformulations have been implemented by other researchers in the field. The flow-based propagator for the SEQUENCE constraint (Section 4.2.3) was implemented in the ECLiPSe Constraint Logic Programming system

[Ecl]. This propagator was adapted to improve pruning in solving the problem of determining the minimum number of points needed to guarantee a playoff spot in the National Hockey League [RvB09]. It was also extended to the case of open/dynamic SEQUENCE constraints [Mah09]. Several decompositions of SEQUENCE constraints (Sections 4.3.2 and 4.3.4) were implemented in the MiniZinc library [G12b], the decomposition of the ALL-DIFFERENT constraint (Section 6.3) was implemented in the MiniZinc library [G12a]. Decompositions of ALL-DIFFERENT and PERMUTATION constraints were smoothly translated into an ASP solver [DW10, Dre10]. This allowed enhancing an ASP solver with range and bounds consistency algorithms for the ALL-DIFFERENT and PERMUTATION constraints. Our work on the transformation of the GRAMMAR constraint and minimisation of unfolded automata helped to improve the REGULAR constraint implementation in GeCode [GeC11b]. The new propagator, starting from version 3.2.0, stores and copies unfolded automata 'That can improve performance considerably (twice as fast) at a slight increase in memory' [GeC11a].

Our work on constraint decompositions has also motivated improvements to constraint solvers that enable them to handle large numbers of primitive constraints more efficiently. As pointed out above, reformulations offer a number of advantages, including incrementality and efficiency. However, in practice, we could not achieve the same performance as monolithic propagators in some cases, like the decomposition of the NVALUE constraint. The reason for this is that the order of constraints in the decomposition, which does not matter for the worst-case theoretical complexity, does matter in practice. The average case complexity of our decompositions depends on the order in which constraints are invoked. To achieve good performance, we need to call constraints in the decomposition in an optimum order to achieve the fixpoint with a minimum number of constraints invocations. So far we do not have control over propagation order for constraints other that the global priority mechanism available in some solvers [GeC11b, Ilo03]. A possible remedy for this problem is to enhance the solver with the ability to group several constraints to propagate them together, in a specific order, during the search. This enhancement was recently prototyped in the GeCode constraint solver. Lagerkvist and Schulte introduced a notion of propagator groups, which allow keeping together a set of constraints and specifying their propagation ordering [LS09].

This work raises a number of open questions.

Constraint decompositions into primitive constraints. What other global constraints can be decomposed into a set of primitive constraints without hindering propagation? What

are tractable cases for conjunctions of global constraints to enforce bounds consistency or domain consistency? We made a first step forward in this direction and considered the OVERLAPPINGALLDIFF constraint. The next step could be to consider conjunctions of two GCC constraints. This conjunction is significantly more complex due to lower bound on values bounds that have to be met. Another direction here is to combine constraint decompositions with symmetry breaking constraints, similar to the recent work [BNQW11]. Can we improve average case behaviour of decomposition in modern constraint solvers? How can we use additional variables that we introduce in these decompositions to improve branching heuristics or nogood learning [KB05, Kat08]?

Constraint decompositions into network flows. What are other global constraints that can be encoded as network flow or shortest path problems? These types of reformulations are very appealing as they allow using a large body of research in the graph theory to improve propagators.

Limitations on the efficiency of a propagator. What are other global constraints that cannot be decomposed into a set of bounded arity constraints of polynomial size? Can we exploit variables with exponentially large domains together with bounded arity constraints to overcame the circuit complexity results?

Whatever the answer to these questions, it is clear from this thesis that decompositions have an important theoretical and practical role to play in the development of constraint solvers.

Index

Bibliography

[AFF+05] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. SAT-based induction for temporal safety properties. *Electron. Notes Theor. Comput. Sci.*, 119:3–16, March 2005.

[AHHT07] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 118–132, Berlin, Heidelberg, 2007. Springer-Verlag.

[AJV01] Jérôme Amilhastre, Philippe Janssen, and Marie-Catherine Vilarem. FA minimisation heuristics for a class of finite languages. In *Revised Papers from the 4th International Workshop on Automata Implementation*, WIA '99, pages 1–12, London, UK, 2001. Springer-Verlag.

[AM04] Carlos Ansotegui and Felip Manya. Mapping problems with finite-domain variables into problems with boolean variables. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 1–15. Springer LNCS, 2004.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[Bac07] Fahiem Bacchus. GAC via Unit Propagation. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741, pages 133–147. Springer, 2007.

[BC80] Robert E. Bixby and William H. Cunningham. Converting linear programs to network problems. *Mathematics of Operation Research*, 5(3):321–357, 1980.

[BC94] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.

[BC01] N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing cardinality-path constraint family. In Philippe Codognet, editor, *Proceedings of the International Conference on Logic Programming (ICLP'01)*, volume 2237, pages 59–73. Springer Verlag, 2001.

[BCP04] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258, pages 107–122. Springer, 2004.

[BCR05] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical report 2005-08, Swedish Institute of Computer Science, 2005.

[Bel01] Nicolas Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239, pages 211–224. Springer Berlin / Heidelberg, 2001.

[Bes06] Christian Bessiere. *Constraint Propagation, Handbook of constraint programming*, page 978. Foundations of Artificial Intelligence, 2006.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30:479–513, July 1983.

[BH03] Christian Bessiere and Pascal Van Hentenryck. To be or not to be ... a global constraint. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833, pages 789–794. Springer, 2003.

[BHH+05] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering Algorithms for the NVALUE Constraint. In Michela Milano Roman Bartak, editor, *Proceedings of the 2th International*

Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'05), volume 3524, pages 79–93. Springer-Verlag, 2005.

[BHH+06a] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering Algorithms for the NVALUE Constraint. *Constraints*, 11(4):271–293, 2006.

[BHH+06b] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The SLIDE-meta constraint. Technical report, 2006.

[BHHW07] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. The complexity of global constraints. *Constraints*, 12(2):239–259, 2007.

[BHLP10] Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca, and Thierry Petit. The increasing NVALUE constraint. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constaint Programming for Combinatorial Optimization Problems (CPAIOR'10)*, volume 6140, pages 25–39. Springer, 2010.

[BHW03] Christian Bessière, Emmanuel Hebrard, and Toby Walsh. Local consistencies in SAT. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 299–314, 2003.

[Bix11] Robert E. Bixby. 'Robert E. Bixby returns to Cornell to deliver the Fulkerson lectures on optimization', 2011. Available from `www.orie.cornell.edu/news/news/profile3.cfm?\customel_dataPageID_3742=32163`.

[BKN+10] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Propagating conjunctions of alldifferent constraints. In *Proceedings of the 24th National Conference on Artificial intelligence (AAAI'10)*. AAAI Press, 2010.

[BKNV11] Lucas Bordeaux, George Katsirelos, Nina Narodytska, and Moshe Y. Vardi. The complexity of integer bound propagation. In *Journal of Artificial Intelligence Research*, volume 40, pages 657–676, 2011.

[BL76] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *Journal of Computer and Systems Sciences*, 13:335–379, 1976.

[BNQW11] Christian Bessiere, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. The alldifferent constraint with precedences(to appear). In *Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constaint Programming for Combinatorial Optimization Problems (CPAIOR'11)*, 2011.

[Bue10] David Buezas. Constraint-based modeling of minimum set covering: Application to species differentiation. Master's thesis, Universidade Nova de Lisboa, 2010.

[CBCGLM07] Marie-Claude Cote, Gendron Bernard, Quimper Claude-Guy, and Rousseau Louis-Martin. Formal languages for integer programming modeling of shift scheduling problems. Technical Report, 2007.

[Cho11] Choco Team. Choco is a java library for constraint satisfaction problems (csp) and constraint programming (cp)., 2011. Available from `http://www.emn.fr/z-info/choco-solver/`.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[CM06] Scott Cotton and Oded Maler. Fast and flexible difference constraint propagation for dpll(t). In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 170–183, 2006.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280, 1990.

[Dar01] Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.

[DB97] R. Debruyne and C. Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the 19th Interna-*

tional Joint Conference on Artificial Intelligence (IJCAI'05), pages 412–417, 1997.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DPR05] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. Constraint programming based column generation for employee timetabling. In *Proceedings of the 2th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'05)*, volume 3524, pages 140–154, 2005.

[DPR06] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006.

[Dre10] Christian Drescher. Constraint answer set programming systems. In *Proceedings of the International Conference on Logic Programming (ICLP'2010) (Technical Communications)*, pages 255–264, 2010.

[DW10] Christian Drescher and Toby Walsh. A translational approach to constraint answer set solving. *TPLP*, 10(4-6):465–480, 2010.

[Ecl] Eclipse Team. Eclipse 6.0 reference manual, the SEQUENCE constraint.

[EFK00] Markus Eiglsperger, Ulrich Fö, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 3–11. Society for Industrial and Applied Mathematics, 2000.

[EKKM05] Khaled Elbassioni, Irit Katriel, Martin Kutz, and Meena Mahajan. Simultaneous matchings. In *Proceedings of 16th International the Symposium Algorithms and Computation (ISAAC'05)*, pages 106–115, 2005.

[Fre95] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[Fre97] Eugene C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997.

[FS09] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, pages 352–366, 2009.

[G12a] G12 Team. Minizinc library, the ALLDIFFERENT constraint. Available from `http://ww2.cs.mu.oz.au/~pjs/MiniZinc_Library/Globals/all_different/index.html`.

[G12b] G12 Team. Minizinc library, the SEQUENCE constraint. Available from `http://ww2.cs.mu.oz.au/~pjs/MiniZinc_Library/Globals/sequence/index.html`.

[G1211] G12 Team. Minizinc, 2011. Available from `http://www.g12.csse.unimelb.edu.au/minizinc/`.

[Gav07] Marco Gavanelli. The log-support encoding of CSP into SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 815–822, 2007.

[GeC11a] GeCode Team. Change log, 2011. Available from `http://www.gecode.org/doc-latest/reference/PageChange.html`.

[GeC11b] GeCode Team. GeCode: Generic constraint development environment, 2011. Available from `http://www.gecode.org`.

[Gen02] Ian P. Gent. Arc consistency in *SAT*. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI'02)*, pages 121–125, 2002.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GMN08] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.

[GMP+01] Ian P. Gent, Ewan Macintyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6:345–372, 2001.

[GN04] Ian P. Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *The 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110, 2004.

[Goo11] Google constraint solver, 2011. Available from `http://code.google.com/p/or-tools/`.

[GR98] Andrew Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[Gut08] Boris Gutkovich. Integration of CP and compilation techniques for instruction sequence test generation. In Laurent Perron and Michael Trick, editors, *Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constaint Programming for Combinatorial Optimization Problems (CPAIOR'08)*, volume 5015, pages 313–317. Springer Berlin / Heidelberg, 2008.

[Hal35] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[HCG09] Pascal Van Hentenryck, Carleton Coffrin, and Boris Gutkovich. Constraint-based local search for the automatic generation of architectural tests. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, pages 787–801, 2009.

[Heb11] Emmanuel Hebrard. Mistral, 2011. Available from `http://4c.ucc.ie/~ehebrard/Software.html`.

[HK06] Willem-Jan van Hoeve and Irit Katriel. *Global Constraints, Handbook of constraint programing*, volume 1, pages 169–209. 2006.

[Hoe05] Willem-Jan van Hoeve. *Operations Research Techniques in Constraint Programming.* PhD thesis, University of Amsterdam, 2005.

[HPR06] Willem-Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12(4-5):347–373, 2006.

[HPRS] Willem-Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. Revisiting the Sequence Constraint. In Frédéric Benhamou,

editor, *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP '06)*, volume 4204, pages 620–634. Springer.

[HPRS09] Willem-Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. New filtering algorithms for combinations of among constraints. *Constraints Journal*, 14(2):273–292, 2009.

[HU79] J. W. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[IBM11] IBM research's Constraint Solver, 2011.

[Ilo03] Ilog Team. ILOG Optimization Suite, 2003. Available from `http://www-01.ibm.com/software/websphere/ilog/`.

[ILO05] ILOG6.2. ILOG Solver 6.1 User's manual. ILOG S.A., 2005.

[JM96] Michael Junger and Petra Mutzel. Exact and heuristic algorithms for 2-layer straightline crossing minimization. In *Proceedings of the Symposium on Graph Drawing*, GD '95, pages 337–348, London, UK, 1996. Springer-Verlag.

[Kat08] George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, 2008.

[KB85] R. V. Kulkarni and P. R. Bhave. Integer programming formulations of vehicle routing problems. *European Journal of Operational Research*, 20(1):58–67, April 1985.

[KB05] George Katsirelos and Fahiem Bacchus. Generalized nogoods in CSPs. In *Proceedings of the 20th National Conference on Artificial intelligence (AAAI'05)*, pages 390–396, 2005.

[KEKM08] Martin Kutz, Khaled M. Elbassioni, Irit Katriel, and Meena Mahajan. Simultaneous matchings: Hardness and approximation. *Journal of Computer and System Sciences*, 74(5):884–897, 2008.

[KNW09] George Katsirelos, Nina Narodytska, and Toby Walsh. Reformulating global grammar constraints. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constaint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, pages 132–147, 2009.

[KS03] Henry A. Kautz and Bart Selman. Ten challenges redux: Recent progress in propositional reasoning and search. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 1–18, 2003.

[KS08] Serdar Kadioglu and Meinolf Sellmann. Efficient context-free grammar constraints. In *Proceedings of the 23th National Conference on Artificial intelligence (AAAI'08)*, pages 310–316. AAAI Press, 2008.

[KT03] Irit Katriel and Sven Thiel. Fast bound consistency for the global cardinality constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833, pages 437–451. Springer, 2003.

[Lau78] J. L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[Lec96] M. Leconte. A bounds-based reduction scheme for difference constraints. In *The 2nd International Workshop on Constraint-based Reasoning, Constraints'96*, 1996.

[Lee02] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, 49:1–15, 2002.

[LMS+09] Frédéric Lardeux, Eric Monfroy, Frédéric Saubion, Broderick Crawford, and Carlos Castro. SAT encoding and CSP reduction for interconnected ALLDIFF Constraints. In *Proceedings of the 8th Mexican International Conference on Artificial Intelligence (MICAI'09)*, pages 360–371. Springer, 2009.

[LOQTvB03] Alejandro Lopez-Ortiz, Claude Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 245–250, 2003.

[LS09] Mikael Z. Lagerkvist and Christian Schulte. Propagator groups. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, pages 524–538, 2009.

[Mah09] Michael J. Maher. Open contractible global constraints. In *Proceedings of the 21th International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 578–583, 2009.

[Mic11] Microsoft. Microsoft Solver Foundation, 2011.

[MMA10] D. Magos, I. Mourtos, and G. Appa. A polyhedral approach to the AllDifferent system. *Mathematical Programming*, pages 1–52, 2010.

[MNQW08] Michael Maher, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Flow-based propagators for the sequence and related global constraints. In Peter J. Stuckey, editor, *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08)*, pages 159–174. Springer, 2008.

[Moo11] Neil Moore. *Improving the efficiency of learning CSP solver*. PhD thesis, University of St Andrews, School of Computer Science, 2011.

[MS72] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT '72: Proceedings of the 13th Annual Symposium on Switching and Automata Theory*, pages 125–129. IEEE Computer Society, 1972.

[MSW96] Jiri Matousek, Micha Sharir, and Emo Welzl. A subexponential bound for linear programming. *Algorithmica*, 16(4/5):498–516, 1996.

[MT00] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 306–319, London, UK, 2000. Springer-Verlag.

[Ney91] Hermann Ney. Dynamic programming parsing for context-free grammars in continuous speech recognition. *IEEE Transactions on Signal Processing*, 39(2):336–340, 1991.

[OMT89] A. Oplobedu, J. Marcovitch, and Y. Tourbier. Charme: un language industriel de programmation par contraintes, illustre par une application chez

renault. In *Proceedings of the Ninth International Workshop on Expert Systems and their Applications*, pages 55–70, 1989.

[OSC07] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741, pages 544–558. Springer, 2007.

[OW01] Patric R. J. Ostergaard and William D. Weakley. Values of domination numbers of the queen's graph. *The Electronic Journal of Combinatorics*, 8(1), 2001.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258, pages 482–495. Springer, 2004.

[PR99] Franccois Pachet and Pierre Roy. Automatic generation of music programs. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, pages 331–345. Springer-Verlag, 1999.

[Pro94] Patrick Prosser. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings of the 11th Eureopean Conference on Artificial Intelligence (ECAI'94)*, pages 95–99, 1994.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[PS03] Karen E. Petrie and Barbara M. Smith. Symmetry breaking in graceful graphs. In Francesca Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833, pages 930–934, 2003.

[Pug98] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 16th National Conference on Artificial intelligence (AAAI'98)*, pages 359–366. American Association for Artificial Intelligence, 1998.

[QLOvBG04] Claude Quimper, Alejandro Lopez-Ortiz, Peter van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, pages 542–556, 2004.

[QR10] Claude-Guy Quimper and Louis-Martin Rousseau. A large neighbourhood search approach to the multi-activity shift scheduling problem. *Journal of Heuristics*, 16:373–392, June 2010.

[Qui06] Claude-Guy Quimper. *Efficient Propagators for Global Constraints*. PhD thesis, University of Waterloo, 2006.

[QW06] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In Frédéric Benhamou, editor, *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204, pages 751–755. Springer, 2006.

[QW07] Claude-Guy Quimper and Toby Walsh. Decomposing global grammar constraints. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741, pages 590–604. Springer, 2007.

[Raz85] A. A. Razborov. Lower bounds on the monotone complexity of some Boolean functions. *Doklady Akademii Nauk SSSR*, 285:798–801, 1985.

[Reg94] Jean-Charles Regin. A filtering algorithm for constraints of difference in *CSP*s. In *Proceedings of the 12th National Conference on Artificial intelligence (AAAI'94)*, volume 1, pages 362–367. American Association for Artificial Intelligence, 1994.

[Reg96] Jean-Charles Regin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 14th National Conference on Artificial intelligence (AAAI'98)*, pages 209–215, 1996.

[Reg99] Jean-Charles Regin. Arc consistency for global cardinality constraints with costs. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, pages 390–404. Springer-Verlag, 1999.

[Reg05] Jean-Charles Regin. Combination of Among and Cardinality constraints. In Roman Barták and Michela Milano, editors, *Proceedings of the 2th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'05)*, volume 3524, pages 288–303. Springer, 2005.

[Rev92] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theor. Comput. Sci.*, 92:181–189, January 1992.

[Roz97] *Handbook of formal languages, vol. 1: word, language, grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[RP97] Jean-Charles Regin and Jean-Francois Puget. A filtering algorithm for global sequencing constraints. In Gert Smolka, editor, *Proceedings of the 3th International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330, pages 32–46. Springer, 1997.

[RT08] Rasmus V. Rasmussen and Michael A. Trick. Round robin scheduling - a survey. *European Journal of Operational Research*, 188(3):617–636, 2008.

[RvB09] Tyrel Russell and Peter van Beek. Determining the number of games needed to guarantee an nhl playoff spot. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constaint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, pages 233–247, 2009.

[Sch86] Alexander Schrijver. *Theory of linear and interger programming*. John Wiley & Sons, Inc., 1986.

[SCNA08] Christine Solnon, Van Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *European Journal of Operational Research*, 191(3):912–927, 2008.

[Sel06] Meinolf Sellmann. The theory of grammar constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204, pages 530–544. Springer, 2006.

[SS04] Christian Schulte and Peter Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Proceedings of the 10th International Conference*

on Principles and Practice of Constraint Programming (CP'04), volume 3258, pages 619–633. Springer-Verlag, 2004.

[Sug11] Sugar Team. Sugar: A sat-based constraint solver, 2011. Available from `http://bach.istc.kobe-u.ac.jp/sugar/`.

[Tru90] Klaus Truemper. A decomposition theory for matroids. v. testing of matrix total unimodularity. *Journal of Combinatorial Theory, Series B*, 49(2):241 – 281, 1990.

[TSW83] G. Tseitin, J. Siekmann, and G. Wrightson. *On the complexity of proofs in propositional logics*, volume 2. Springer-Verlag, 1983. Originally published 1970.

[TTKB06] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP to SAT. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, pages 590–603, 2006.

[WM62] Arthur F. Veinott Jr. Wagner and Harvey M. Optimal capacity scheduling I. *Operations Research*, 10(4):518–532, 1962.

[WW86] K. Wagner and G. Wechsung. *Computational Complexity*. Springer, 1986.

[Zar04] Emmanuel Zarpas. Simple yet efficient improvements of *SAT* based bounded model checking. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design(FMCAD'04)*, pages 174–185, 2004.

[Zar05] Emmanuel Zarpas. Benchmarking *SAT* solvers for bounded model checking. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, volume 3569, pages 23–33. Springer Berlin / Heidelberg, 2005.

[ZMP06] Alessandro Zanarini, Michela Milano, and Gilles Pesant. Improved algorithm for the soft global cardinality constraint. In *Proceedings of the 3th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'06)*, pages 288–299, 2006.