Efficient Propagators for Global Constraints

by

Claude-Guy Quimper

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2006

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Claude-Guy Quimper

Abstract

We study in this thesis three well known global constraints. The ALL-DIFFERENT constraint restricts a set of variables to be assigned to distinct values. The *global cardinality constraint* (GCC) ensures that a value v is assigned to at least l_v variables and to at most u_v variables among a set of given variables where l_v and u_v are non-negative integers such that $l_v \leq u_v$. The INTER-DISTANCE constraint ensures that all variables, among a set of variables $x_1, \ldots, x_n$, are pairwise distant from p, i.e. $|x_i - x_j| \geq p$ for all $i \neq j$. The ALL-DIFFERENT constraint, the GCC, and the INTER-DISTANCE constraint are largely used in scheduling problems. For instance, in scheduling problems where tasks with unit processing time compete for a single resource, we have an ALL-DIFFERENT constraint on the starting time variables. When there are k resources, we have a GCC with $l_v = 0$ and $u_v = k$ over all starting time variables. Finally, if tasks have processing time t and compete for a single resource, we have an INTER-DISTANCE constraint with $p = t$ over all starting time variables. We present new propagators for the ALL-DIFFERENT constraint, the GCC, and the INTER-DISTANCE constraint i.e., new filtering algorithms that reduce the search space according to these constraints. For a given consistency, our propagators outperform previous propagators both in practice and in theory. The gains in performance are achieved through judicious use of advanced data structures combined with novel results on the structural properties of the constraints.

Acknowledgements

I would like to thank my advisor Alejandro López-Ortiz for his enthusiasm in working with me on a broad selection of problems ranging from video-on-demand to computational geometry, and of course, constraint programming which is the topic of this thesis. I appreciated the time we spent working together. I will remember his precious advice on the importance of constantly acquiring new knowledge and diversifying my research topics as well as his advice on how to address open problems. I would also like to thank Peter van Beek whose excellent course on constraint programming was the starting point for my doctoral studies. I appreciated his judicious comments, suggestions, and ideas. Many thanks to Toby Walsh for inviting me twice to visit NICTA in Sydney, Australia. I really appreciate his invitation to join the COMIC research group on constraint programming. Many thanks to Bill Cunningham, Ian Munro, Peter van Beek, and Pascal Van Hentenryck for being on my thesis committee and their useful comments. I would like to thank all my coauthors. I am looking forward to collaborating again with you. Finally, I would like to thank my mother Lyse, my father Michel, my girlfriend Jiye, and all my friends for their presence and their support during my graduate studies.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Large organizations such as corporations and governments need to optimize their operations or simply find satisfactory solutions to their problems in the course of their normal business activities. With the advent of the electronic computer during the second half of the last century, it became feasible to attempt to solve these problems mechanically. This new technology even allowed to optimize processes that would have never been tackled before. Since many of these problems are NP-Hard, there is no known polynomial time algorithm that can solve all instances. It follows then that combinatorial problems need to be studied using an alternative approach in order to obtain a solution. The scientific community has suggested multiple approaches to solve combinatorial problems. In general, solutions that are more flexible have gained more attention.

The operations research community has extensively used linear programming as a main tool to solve optimization and also combinatorial problems. The technique consists of reducing a problem to a linear program i.e., a system of linear equations $Ax = b$ subject to $x \geq 0$ whose solution must minimize the linear expression $c^T x$. Linear program solvers like the simplex method or the interior point method efficiently find optimal solutions to linear programs. One can add the restriction that the components of x are required to take integer values. This new problem called an integer program captures the class of NP-Hard problems.

NP-Hard problems can also be reduced to the satisfiability problem. This prob-

lem consists of finding a valid assignment to boolean variables that are subject to an expression formed by conjunctions, disjunctions, and negations of boolean variables. The SAT community offers different tools to solve the satisfiability problem. NP-Hard problems can be reduced to the satisfiability problem and then submitted to a SAT solver.

All the techniques described above require reducing a combinatorial or an optimization problem to a linear program, an integer program, or a satisfiability problem, and then submit the transformed problem to a specialized solver. There exist some programming paradigms that allows to directly express the problem as it is and let the computer find the solution.

Logic programming is a programming paradigm that aims at solving problems only by stating the facts about the instance of the problem and by giving some inference rules. Prolog [15] is a well known programming language that uses this paradigm. The paradigm evolved to become *constraint logic programming* which directly uses constraints such as $A > B$ as predicates. *Constraint programming* follows the constraint logic programming paradigm. With constraint programming, one lists the variables of the problem and the constraints representing the relations between the variables. Each variable is associated to a set of values called its *domain* that specifies which values can be assigned to the variable. The variables, the domains, and the constraints form a *constraint satisfaction problem*. Different search techniques exist to solve constraint satisfaction problems such as the exploration of a search tree and local searches.

A common technique to reduce the search space is to filter the domains of the variables. Let $\mathrm{dom}(x_1) = [1, 4]$ and $\mathrm{dom}(x_2) = [2, 6]$ be two variable domains. Let $x_1 > x_2$ be a constraint in the problem. Filtering this constraint removes from $\mathrm{dom}(x_1)$ and $\mathrm{dom}(x_2)$ the values that cannot be part of any solution. In our example, filtering the domains of x_1 and x_2 results in the new domains: $\mathrm{dom}(x_1) = [3, 4]$ and $\mathrm{dom}(x_2) = [2, 3]$. The algorithm that filters the domain is called a *filtering algorithm* or *propagator*. The *constraint propagation* phase consists of iteratively calling the constraint propagators until no more values can be removed. When the constraint propagation phase reaches this fix point, the problem is said to be *locally consistent*.

A propagator can enforce different level of consistencies. A propagator enforcing *domain consistency* removes the maximum number of values from the domains. A propagator enforcing *bounds consistency* approximates the variable domains with intervals and then increases the lower bound and decreases the upper bound of each domain. Finally, a propagator enforcing *range consistency* removes all possible values from one domain while approximating all other domains with intervals.

Some constraints occur more often than others in combinatorial problems. This is the case for the ALL-DIFFERENT constraint which ensures that a set of variables are assigned to pairwise distinct values. This constraint is largely used in scheduling problems where tasks with unit processing time compete for a resource. We therefore want all execution times to be different. Permutations can also be modeled using an ALL-DIFFERENT constraint. We simply post an ALL-DIFFERENT constraint on n variables whose domains consists of the integers from 1 to n. The ALL-DIFFERENT constraint is one of the most studied constraints (see for instance [46, 51, 54, 60, 63, 65, 67, 83, 84]). We present in this thesis (see Chapter 4) new algorithms that filter the variable domains for this constraint.

A generalization of the ALL-DIFFERENT constraint is the *global cardinality constraint* (GCC). This constraint ensures that a value v is assigned to at least l_v variables and to at most u_v variables among a set of given variables. When $l_v = 0$ and $u_v = 1$ for all values v, we obtain the ALL-DIFFERENT constraint. This constraint naturally appears, for instance, in scheduling problems where tasks with unit processing times compete for a set of k resources. In this case, we have a GCC over all starting time variables where $l_v = 0$ and $u_v = k$. The GCC also occurs in sport scheduling problems where teams must play at least once a week but no more than say, thrice a week.

We study another generalization of the ALL-DIFFERENT constraint called the INTER-DISTANCE constraint. Given a set of variables $x_1, \ldots, x_n$, the INTER-DISTANCE constraint ensures that all variables are pairwise distant from p, i.e. $|x_i - x_j| \geq p$ for all $i \neq j$. When $p = 1$, the INTER-DISTANCE constraint specializes into an ALL-DIFFERENT constraint. The INTER-DISTANCE constraint is particularly useful for modeling scheduling problems where tasks with processing time p compete for a single resource. In this case, we want each starting time to be at distance at

least p from all other starting times.

The main contribution of this thesis is a collection of new propagators enforcing bounds and range consistency for the ALL-DIFFERENT constraint and the GCC. We also present a new propagator for the bounds consistency of the INTER-DISTANCE constraint. The development of propagators for global constraints is an important research area in constraint programming. Designing specialized propagators for these constraints can lead to a stronger pruning of the search tree that can result in an exponential gain in computational time [78].

Even though we approach problems from a theoretical point of view, the solutions we propose are of practical use. We implemented our propagators and tested them against already existing techniques. All our propagators were implemented using constraint programming libraries such as ILog. When comparing to existing propagators achieving the same consistencies, our propagators offer better performance both in practice and in theory.

Some of our algorithms have already been integrated (by others) into different commercial and open source constraint programming libraries such as Choco [44], Gecode [2], Koalog [43], Mistral [30], and Disolver [29].

The algorithms presented in this thesis have real life applications. Indeed, the constraints we study in this thesis model many problems occurring in industry.

For example, in manufacturing, the ALL-DIFFERENT constraint and the GCC allow modelling, planning, and scheduling of tasks when resources are limited. Problems can range from scheduling the daily operations of a plant in order to satisfy demand to planning the annual operations based on an expected demand. The ALL-DIFFERENT constraint, the GCC, the INTER-DISTANCE constraints allow for the optimization of plant utilization, management of events on a supply-chain, and inventory control. For instance, the wood industry has established different techniques to dry wood. One can dry wood with a dryer or simply by storing the wood in a yard. Each process leads to different wood qualities and processing times. The GCC and the ALL-DIFFERENT constraint can help in modelling a plant to ensure that the capacity of the dryers and the yard is not exceeded while still satisfying the demands throughout the year.

In planning, it is often the case that one desires to order a sequence of operations. The All-Different constraint ensures that each operation appears once and only once in the sequence. Fast propagators are required in order to reschedule tasks when perturbations occur in a schedule or the demand for goods fluctuates. The need for robust solutions often increases the demand on computation time. It is therefore more important than ever to have time efficient algorithms.

For the All-Different, GCC, and Inter-Distance constraints, we propose some propagators whose complexities are independent of the range of possibilities a variable can be assigned. This is particularly relevant in scheduling problems where variables are assigned time points usually expressed in hours, minutes, or seconds. The more precise the schedule is, the larger the numbers get (e.g. 1 hour = 3600 seconds). We present in this thesis solutions such that the increase in precision does not alter the running time of the algorithms. This is a critical requirement in air traffic control [6]. In this field, computer scientists solve scheduling problems for plane landing such that a gap of time g is maintained between each landing for safety reasons. The propagator we present for the Inter-Distance constraint solves this scheduling problem without being affected by the requested precision.

The propagator we propose for the Inter-Distance constraint is particularly useful to model the frequency allocation problem. Consider an environment where many communication devices need to communicate through radio signals. Each transmitter and receiver can emit or receive signals within a specific range of the radio spectrum. Moreover, each pair of transmitter and receiver must use a frequency that does not interfere with other communications. In other words, each radio system within a same geographical area must be assigned to frequencies spaced out by a given gap g that ensures interference-free communications. The Inter-Distance constraint models this invariant. As for scheduling problems, our propagator is as efficient when solutions are expressed in *MHz* as when expressed in *Hz*. The magnitude of the solution numbers does not affect the running time.

Compilers need to efficiently schedule sequences of instructions to enter the execution pipeline of the processor. Some processors allow one, two, or even four instructions to enter the pipeline at the same time. Malik et al. [53] used our propagators for the GCC and the All-Different constraint to compute the

optimal schedule that leads to the fastest execution of a program. They showed that even though the problem in its general form is NP-Hard, constraint programming is time competitive with the heuristic based approach and returns optimal solutions.

Part of the work presented in this thesis has already been published in the following form:

1. C.-G. Quimper, A. López-Ortiz, G. Pesant. *A Quadratic Propagator for the Inter-Distance Constraint*, In Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 06), 2006.

2. C.-G. Quimper and T. Walsh, *The All Different and Global Cardinality Constraints on Set, Multiset and Tuple Variables*, to appear in Recent Advances in Constraints: Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, Revised Selected and Invited Papers. Lecture Notes in Artificial Intelligence, Vol. 3419 M. Carlsson, F. Fages, B. Hnich, and F. Rossi (Eds.) 2006.

3. C.-G. Quimper and T. Walsh. *Beyond Finite Domains: the All Different and Global Cardinality Constraints*, In Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, pages 812–816 and in Proceedings of 1st International Workshop on Constraint Programming: Beyond Finite Integer Domains, pages 5–17, Sitges, Spain, 2005.

4. C.-G. Quimper, A. Golynski, A. López-Ortiz, and P. van Beek, *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint*, Invited paper for the Special Issue of the Ninth International Conference on Principles and Practice of Constraint Programming, Constraint Journal, 10(2):115-135, 2005

5. C.-G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. *Improved Algorithms for the Global Cardinality Constraint*, In Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, Toronto, Ontario, September, pages 542-556, 2004.

6. C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. Bashir Sadjad. *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint*, In Proceedings of the 9th International Conference on Principles and Practice of

Constraint Programming (also available as a technical report, CS-2003-10, School of Computer Science, University of Waterloo), Kinsale, Ireland, September, pages 600-614, 2003.

7. C.-G. Quimper. *Enforcing Domain Consistency on the Extended Global Cardinality Constraint is NP-hard*. Technical Report, CS-2003-39, School of Computer Science, University of Waterloo, 2003.

8. A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. 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 (IJCAI03) (also available as a technical report, CS-2003-05, School of Computer Science, University of Waterloo), Acapulco, Mexico, August, pages 245-250, 2003.

The thesis is structured as follows. In Chapter 2, we present the general notions used throughout the thesis. The reader familiar with graph theory and constraint programming can skip these sections. In Chapter 3, we present a survey of different existing propagators for the GCC and the All-Different constraint. In Chapter 4, we present our propagators for the All-Different constraint. In Chapter 5, we present our propagators for the GCC. Finally, In Chapter 6, we present a propagator for the Inter-Distance constraint. We conclude in Chapter 7.

Chapter 2

Theoretical Background

2.1 Introduction

We present in this chapter the fundamental notions upon which our research is based. First, we introduce network flows and matchings in graphs. Finally, we introduce the main concepts in constraint programming. Note that we limit our survey to those aspects within the scope of our research as each of the topics discussed here forms a research field on its own.

2.2 Graph Theory

Network flows form an important class of satisfaction and optimization problems. Dantzig, Ford, and Fulkerson [19] settled the basics of the field by studying graph problems. We consider three main problems. In the first one, called the transportation problem, one tries to minimize the cost of carrying goods from warehouses to stores under constraints for demand and supply. In the second problem called the shortest path problem, the goal is to minimize the distance traveled in a graph to go from a vertex a to a vertex b. Finally, the assignment problem consists in assigning one task to a machine, given the set of tasks each machine can accomplish. These problems served as a starting point from which many aspects of graph theory

emerged. We provide in this section the basics of flow theory that are used in the context of our research. We follow the notation of Ahuja, Magnanti, and Orlin [4]. We then specialize to matching theory, an instance of flow theory.

2.2.1 Network Flows

A *flow* in a directed graph $G = \langle V, E \rangle$ is a mapping $f : E \mapsto \mathbb{N}$ between an edge and an integer. A *source* is a node with only outgoing edges and a *sink* is a node with only incoming edges. All other nodes are called *inner nodes*. In this work, we consider graphs with only one source s and one sink t. Any inner node n is subject to the conservation equilibrium condition which states that the amount of incoming flow in node n is equal to the amount of outgoing flow. More formally, for a given node $n \in V$ we have:

$$\sum_{(a,n)\in E} f(a,n) \quad = \quad \sum_{(n,b)\in E} f(n,b) \tag{2.1}$$

The *flow value* $v(f)$ is the amount of flow going out of the source i.e., $v(f) = \sum_{(s,u)\in E} f(s,u)$. Notice that by propagation of the conservation equilibrium, the flow value is also the amount of flow arriving at the sink.

Flows are often subject to capacity constraints. For example, we might wish to have a minimum amount of flow on a given edge. At the same time, an edge might have a maximum capacity. Formally, a flow $f(e)$ on an edge $e \in E$ is bounded by the lower capacity $l(e)$ and the upper capacity $u(e)$ of the edge i.e., $l(e) \leq f(e) \leq u(e)$. A flow that satisfies all capacity constraints is called a *feasible flow*. Finally, a *maximum flow* is a feasible flow whose flow value is maximum.

Given a feasible flow f in a graph G, the *residual graph* G_f of f is a directed graph which captures the unused or left over capacity from the flow. The residual graph G_f has the same nodes as graph G, and for each edge $e = (u, v)$ in G, we have,

a) a *forward* edge (u, v) in G_f with weight $u(e) - f(e)$ if $f(e) < u(e)$,

b) a *backward* edge (v, u) in G_f with weight $f(e)$ if $f(e) > l(e)$,

c) no other edges.

Residual graphs are used in the Ford-Fulkerson algorithm [19] to construct maximum flows. Indeed, suppose f is a feasible flow. Let an *augmenting path* p be a path in the residual graph G_f that connects the source s to the sink t. Let Δf be the smallest weight associated to an edge in p. Then, adding (subtracting) Δf to the flow of each forward (backward) edge in p results in a flow f' of greater value. It is known that successively finding augmenting paths and consequently modifying the flow f until no paths exist between the source s and the sink t leads to a maximum flow. A depth-first-search finds an augmenting path in $O(|E|)$ steps resulting in an overall running time complexity of $O(v(f)|E|)$.

The Ford-Fulkerson algorithm successively finds augmenting paths until no augmenting path connects the source s to the sink t. The following theorem attributed to Berge [10] guarantees that the resulting flow is maximal. Petersen [59] proved this theorem before Berge for a more restrictive class of flows called matchings (see next section).

Theorem 2.1 (Berge [10]) *A flow f is a maximum flow if and only if there is no augmenting path in the residual graph G_f connecting the source s to the sink t.*

The Ford-Fulkerson algorithm finds a maximum flow in a graph as long as an initial feasible flow is provided. When the lower capacities $l(e)$ are null for every edge in the graph, the trivial flow in which $f(e) = 0$ for every edge e is a feasible flow to start with. When some lower capacities are strictly positive i.e., $l(e) > 0$, there is no trivial feasible flow. In fact, some graphs do not have any feasible flows. Berge [11] showed how to construct an initial feasible flow, if one exists.

Berge's algorithm [11] transforms a graph G with positive lower capacities into a graph G' where all lower capacities are null. For every edge $(a, b) \in E$ with a positive lower capacity $l(a, b) > 0$, he suggests to create an edge (s, b) in G' connecting the source to the end point b with an upper capacity $u(s, b) = l(a, b)$ and a null lower capacity $l(s, b) = 0$. Similarly, we create an edge (a, t) in G' connecting

the starting point of the edge to the sink with upper capacity $u(a,t) = l(a,b)$ and a null lower capacity $l(a,t) = 0$. The upper capacity of edge (a,b) is modified to become $u(a,b) - l(a,b)$ while the lower capacity is set to 0. We now find a flow that saturates all edges that have been added in G'. To do so, we successively find augmenting paths in the residual graph of G' that pass through the new edges until all these edges become saturated. If there is no augmenting path passing by a new unsaturated edge, then there exists no feasible flow and the problem is unsolvable. To retrieve the feasible flow in G, we simply delete the edges that were added in G' and add the lower capacity $l(e)$ to the flow $f(e)$ for every edge in the graph. Figure 2.1 illustrates the process using a small graph as an example.

Residual graphs hold an interesting property: they quickly allow to determine if there exists a maximum flow in which a specific edge carries a flow. More formally, we have the following theorem which is common knowledge in flow theory. See for instance Ahuja et al. [4].

Theorem 2.2 *Let f be a maximum flow in graph G, G_f its associated residual graph, and e an edge in G. There exists a maximum flow f' such that $f'(e) > 0$ if and only if $f(e) > 0$ or e belongs to a cycle of G_f.*

2.2.2 Matchings

The *assignment problem* is a special case of the maximum flow problem. It is also known as the *maximum matching* problem. Consider a bipartite graph $G = \langle V, E \rangle$ where the nodes V are divided into two sets: the left nodes set L and the right nodes set R. All edges in E connect a node in L with a node in R. A *matching* M is a subset of E such that no two edges are adjacent to the same node. A *maximum matching* M is a matching such that the number of edges in M is maximal. A *matched edge* is an edge in M and a *matched node* is a node adjacent to a matched edge. A *free edge* is an edge that is not in M and a *free node* is a node that is not adjacent to any edge in M.

One can compute a maximum matching by adding a source and a sink to the graph G. The source is connected to all edges in L and the sink to all edges in R.

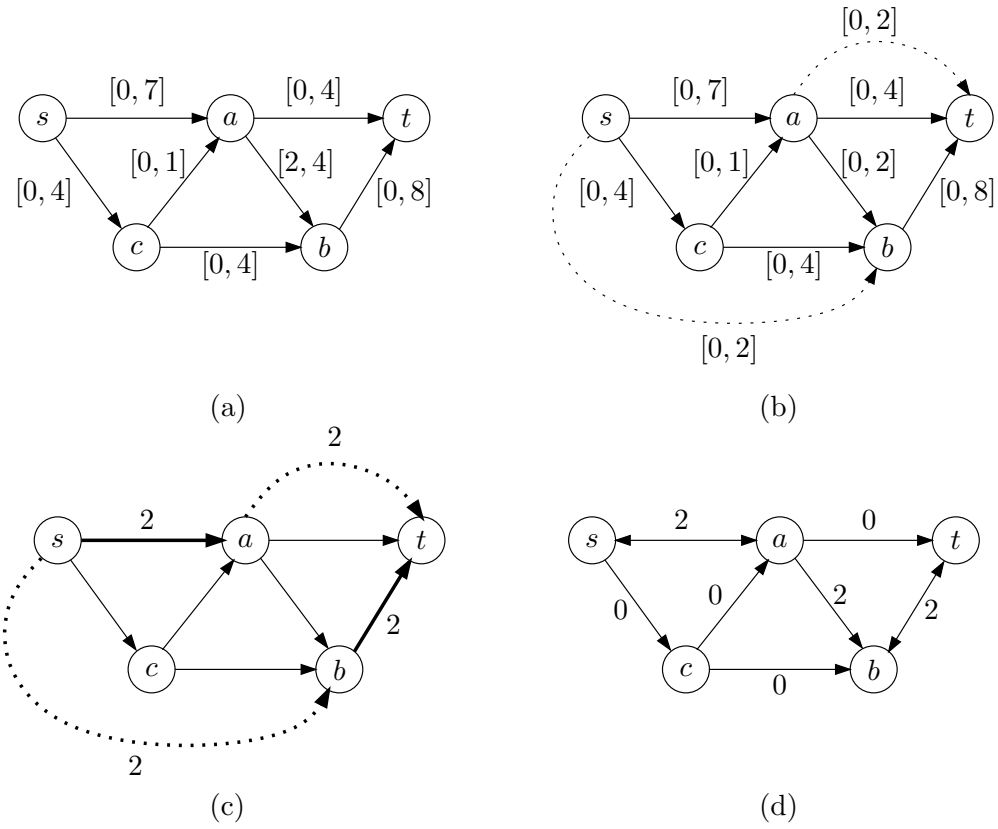

Figure 2.1: Finding a feasible flow in a graph where some edges have a positive lower capacity. a) Original graph with lower and upper capacities. b) Modified graph with new edges. c) Augmenting paths saturating the new edges. d) Feasible flow and resulting residual graph.

All edges have unit upper capacity i.e., $u(e) = 1$. In a maximum flow f, the edges in E that admit a flow of one form the maximum matching M. Figure 2.2 shows a matching (edges in bold) obtained from a network flow.

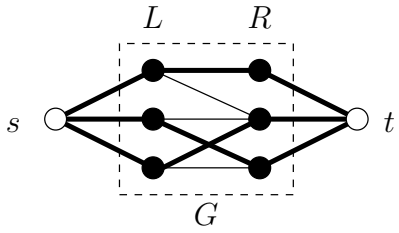

Figure 2.2: Matching in a bipartite graph obtained from a network flow.

The residual graph of a matching M is the directed graph G_M where matched edges are oriented from right nodes to left nodes and free edges are oriented from left nodes to right nodes. An augmenting path is a path connecting a free node in L to a free node in R. Let M be a matching and P be the edges of an augmenting path. Observe that a matching of higher cardinality can be obtained by computing the symmetric difference $M \oplus P$ defined as $M \oplus P = (M \cup P) - (M \cap P)$.

Hopcroft and Karp [36] developed an algorithm that finds a maximum matching M in a bipartite graph in $O(\sqrt{M}|E|)$ steps. Their algorithm starts with an initially empty matching $M = \emptyset$ which is improved at each iteration by finding a set of disjoint shortest augmenting paths. An iteration that finds a set of augmenting paths proceeds in two steps.

The first step consists of performing a breadth-first search (BFS) on the residual graph G_M starting from the set of free nodes in L. The BFS labels nodes with value i if the node is at distance i from a free node in L. This distance is minimal by construction of the BFS. Let d be the length of the shortest path between a free node in L and a free node in R. The BFS labels all nodes whose distance from empty nodes in L is not greater than d.

The second step of the algorithm uses a stack to perform a depth-first search (DFS). The DFS starts from a free node in L and is only allowed to traverse an edge (a, b) such that node a is labelled with i and b is an unvisited node labelled with

$i+1$. After traversing such an edge, the DFS marks both nodes a and b as visited. When the DFS reaches a free node in R, the nodes stored in the DFS stack form a path starting from a free node in L to a free node in R. By definition, this path is an augmenting path. The DFS restarts on a different free node in L and tries to reach another free node in R without visiting nodes that have been visited in a previous DFS. When the DFS finishes, a collection of disjoint shortest augmenting paths have been found. The matching can consequently be augmented. Hopcroft and Karp proved that both steps need to be executed a maximum of $\sqrt{|M|}$ times to obtain a maximum matching M. This observation leads to a running time analysis of $O(\sqrt{|M|}|E|)$.

A natural question is to determine if there exists a maximum matching M that contains a specific edge e in E. Clearly, a maximum matching M is a certificate that proves that each of its edges belongs to a least one maximum matching. How about for edges not in M? Suppose we have an alternating cycle C in G_M. By *alternating cycle*, we mean an even length cycle in the residual graph G_M whose edges are alternately in M and in $E - M$. One could apply the symmetric difference $M \oplus C$ to obtain a new matching of equal cardinality where every edge in the cycle C is added if it does not belong to M or removed if it already belongs to M. Therefore, the maximum matchings M and $M \oplus C$ are two certificates proving that every edge on an alternating cycle belongs to at least one maximum matching.

A second transformation can be applied to obtain a new matching of equal cardinality. An *alternating path* is a path whose edges are alternately in M and in $E - M$. Let p be an alternating path of even length starting at a free node. By applying the symmetric difference $M \oplus p$, we obtain a new matching of equal cardinality where edges in p are removed from M if they belong to M or added to M if they do not belong to the matching. The matchings M and $M \oplus p$ are two certificates proving that edges in p belong to at least one maximum matching.

Note that an edge might have many different certificates proving that it belongs to some maximum cardinality matching. Some other edges might not belong to any maximum cardinality matching and therefore do not have any certificate. The following theorem specifies when an edge belongs to some but not all maximum matchings.

Theorem 2.3 (Berge [11]) *An edge belongs to some but not all maximum matchings if and only if, for an arbitrary maximum matching M, it belongs to either an even alternating path which begins at a free vertex, or an even alternating cycle.*

Proof ($\Leftarrow$) Let M be the edges of a maximum matching. Let P be the edges of an even alternating paths which begins at a free vertex or the edges of an even alternating cycle. Let $M' = M \oplus P$ be a maximum matching formed by the symmetric difference of M and P. Clearly, every edge in P belongs to M or M' but not both.

($\Rightarrow$) Let M and M' be two maximum matchings such that edge e belongs to only one of these matchings. Let $P = M \oplus M'$ be the mutual difference of M and M'. Since the degree of a node in M and M' is at most one, the degree of a node in P is at most two. There are therefore only paths and cycles in P. The only way a node $n \in P$ can have a degree of 2 is by having an adjacent edge in M and an other adjacent edge in M'. Therefore, every path and cycles are alternating paths of M and M'. The paths in P cannot be augmenting paths in M or M' since they are maximum matchings. Consequently, the paths in P necessarily have an even length. To conclude the proof, since edge e belongs to only one of the matchings M and M', then it belongs to an alternative path or cycle in P of even length. □

From Theorem 2.3 we conclude the following. If an edge belongs to an even alternating path or an even alternating cycle, then it belongs to some but not all matchings. If an edge does not belong to any even alternating path or even cycle, it either belongs to all or no matchings. If such an edge belongs to a matching M then it belongs to all matchings and if it does not belong to matching M then it belongs to no matchings.

Finding a maximum matching M, the alternating cycles, and the even alternating paths is therefore sufficient to decide if each edge belongs to a maximum matching.

2.2.3 Hall's Marriage Theorem

Hall [28] studied the problem of testing the existence of a complete system of distinct representatives (CDR). Consider m sets $S_1, \ldots, S_m$. A CDR is a set of m distinct elements $t_1, \ldots, t_m$ such that $t_i \in S_i$. Hall's research led to what is now known as Hall's marriage theorem which we quote verbatim.

Theorem 2.4 (Hall [28]) *In order that a CDR shall exist, it is sufficient that for each $k = 1, 2, \ldots, m$, any selection of k of the sets shall contain between them at least k elements.*

In other words, there exists a CDR if and only if the union of any k sets among $S_1, \ldots, S_m$ contains at least k elements.

This result has varied applications in matching theory. Consider a bipartite graph G with left-nodes L and right-nodes R such that $|L| \leq |R|$. Let adj(n) be the set of nodes adjacent to node n. From Hall's marriage theorem, we deduce the following corollary.

Corollary 2.1 *There exists a matching of size $|L|$ if and only if for any set S of $k \leq |L|$ nodes, we have $|\cup_{n \in S} adj(n)| \geq k$.*

Hall's marriage theorem will be particularly useful in Chapter 4 and Chapter 5.

2.3 Constraint Programming

2.3.1 Historical Background

Constraint programming is the result of a long evolution that traces its roots to logic programming. This programming paradigm evolved to become logic programming and finally became constraint programming. We will describe each of these programming paradigms and the main languages that use these paradigms.

Logic programming was first introduced with the programming language PLANNER [35] and later popularized with Prolog [15]. Contrarily to procedural languages

such as C and Pascal that require a clear sequence of instructions to achieve a computation, logic programming only requires a set of predicates stating the facts about the problem and a set of rules allowing to generate new facts. For instance, to define a function returning *true* if an element e belongs to a sequence L, one simply has to mention that e belongs to L if the sequence L, or one of its subsequences, starts with the element e. Example 2.1 shows the Prolog code associated to these rules.

Example 2.1 The following code defines a function `isIn(E, L)` that returns *true* if element E appears in the list L.

```
isIn(E, [E | _]).
isIn(E, [_ | L]) :- isIn(E, L).
```

The user states its request on the form of a goal. For instance `isIn(4, [2, X, 4])`, the following goal requests if there exists an X such that the expression is satisfied. The solver refers to the rules entered in the inference engine to derive new goals that need to be achieved. In our example, applying the second rule leads to the new goal `isIn(4, [X, 4])` which can be satisfied by applying the first rule. Prolog was first developed to process natural languages [15]. It later became popular in many areas, from theorem proving [79] to planning [48].

The *logic programming* paradigm evolved to become the *constraint logic programming* paradigm. With this paradigm, one can use constraints instead of predicates. For instance, the rule `A(X) :- X > 100, B(X)` has two predicates $A(X)$ and $B(X)$ and one constraint $X > 100$. Suppose while trying to satisfy the goal $A(X)$, the inference engine encounters another constraint $X < 50$. Then the inference engine concludes that the goal $A(X)$ is unreachable. This is done without attempting to assign a value to the variable X.

The programming language CHIP [17] introduced finite domains to constraint logic programming. A variable X can be required to belong to a finite domain such as $X \in \{1, 4, 6\}$, $X \in [1, 5]$, or $X \in \{red, green, blue\}$. The presence of constraints can eliminate some inconsistent values in the domains. ECLiPSe [1] is another example of a constraint logic programming language.

Constraint programming is a programming paradigm based on constraint logic programming. This programming paradigm allows to state the problem one wants to solve by declaring the variables of the problem, the domains associated to the variables, and some constraints describing the relations between the variables. The variables, the domains, and the constraints form a *constraint satisfaction problem*. The aim of constraint programming is to solve any problem simply by stating *what* to solve while minimizing the stated information about *how* to solve it. OPL [32] is an example of a constraint programming language.

Constraint programming libraries are now implemented with imperative languages. Some libraries such as Gecode [2], ILog [3], Disolver [29], Mistral [30], Koalog [43], and Choco [44] allow to state the variables and the constraints of a problem in languages such as Java and C++. This architecture allows a better integration of constraint programming to legacy code. Many programming languages among B-Prolog [88], CHIP V5 [17], Ciao Prolog [12], ECLiPSe [1], Oz/Mozart [73], SICStus [38], and OPL [31] also offer integration with other programming languages.

2.3.2 General Concepts

We present in this section the main concepts related to constraint programming. A *Constraint Satisfaction Problem* (CSP) is defined by a set of variables $X = \{x_1, x_2, \ldots, x_n\}$ and a set of constraints $C = \{C_1, C_2, \ldots, C_m\}$. The *domain* $\mathrm{dom}(x_i)$ of a variable x_i defines which values can be assigned to the variable x_i. An *assignment* is an n-tuple $t = (t_1, \ldots, t_n)$ such that $x_i = t_i$ for all $x_i \in X$. Each *constraint* $C_i \in C$ is associated to a set of variables $\mathrm{Var}(C_i) \subseteq X$ called the *scope* of the constraint. Formally, a constraint C_i is a set of $|\mathrm{Var}(C_i)|$-tuples i.e., a set of tuples of size $|\mathrm{Var}(C_i)|$. Let $i_1, \ldots, i_m$ be the indices of the variables in $\mathrm{Var}(C_i)$. An assignment t satisfies the constraint C_i if $(t_{i_1}, \ldots, t_{i_m}) \in C_i$. An assignment t is a *solution* if it belongs to the Cartesian product of all variable domains i.e., $t_i \in \mathrm{dom}(x_i)$ and it satisfies all constraints. Example 2.2 gives an example of a CSP.

Example 2.2 Let x_1, x_2, and x_3 be three variables such that $\mathrm{dom}(x_1) = \{2, 3, 4\}$, $\mathrm{dom}(x_2) = \{1, 2, 3, 4\}$, and $\mathrm{dom}(x_3) = \{3, 4, 5\}$. Let $C_1 : x_1 < x_2$ and $C_2 : x_1 + x_2 =$

x_3 be two constraints such that $\text{Var}(C_1) = \{x_1, x_2\}$ and $\text{Var}(C_2) = \{x_1, x_2, x_3\}$. The unique solution t to this problem is given by the tuple $t = \langle x_1, x_2, x_3 \rangle = \langle 2, 3, 5 \rangle$.

The arity of a constraint C_i is the number of variables in its scope $\text{Var}(C_i)$. When the arity is one, we say that the constraint is a *unary constraint*. When the arity is two, the constraint is a *binary constraint*. When the arity is an arbitrary but fixed n, the constraint is an n-ary constraint.

Some constraints can be decomposed into simpler constraints i.e, constraints of smaller arity. For instance, consider the constraint $\textsc{Sum}([x_1, \ldots, x_n], z)$ which is satisfied if $\sum_{i=1}^{n} x_i = z$. This constraint, after introducing the variables $y_1, \ldots, y_n$, can be decomposed into the binary and ternary constraints $y_1 = x_1$, $y_i = y_{i-1} + x_i$, and $z = y_n$. The Sum constraint is a global constraint.

Definition 2.1 (Global Constraint) A *global constraint* is a constraint of arbitrary arity that can be decomposed into constraints of smaller arity.

Global constraints gained in popularity with the CHIP system [9, 17]. Beldiceanu et al. [8] list more than 270 global constraints in their catalog. We will see in the following sections and chapters that some global constraints can solve CSPs more efficiently.

2.3.3 Consistencies

Filtering the variable domains is an important step in the resolution of CSPs [52, 74]. It consists of removing from the variable domains the values that cannot be part of a solution. We present the notions of *support* and *consistency* that will later be used to filter the variable domains.

Definition 2.2 (Support) Let C be a constraint and $x_{i_1}, \ldots, x_{i_m}$ be the variables in the scope $\text{Var}(C)$. A *support* for value $v \in \text{dom}(x_{i_j})$ is an assignment t such that $t_{i_j} = v$ and $(t_{i_1}, \ldots, t_{i,m}) \in C$ i.e., a tuple satisfying the constraint C, such that $t_{i_j} = v$.

We study in particular two types of support.

A *domain support* determines for which values there exists an assignment satisfying a given constraint such that each variable is assigned to a value in its domain.

Definition 2.3 (Domain Support) Given a constraint C and a variable $x_i \in Var(C)$, a value $v \in \text{dom}(x_i)$ has a *domain support* with respect to x_i if there exists an assignment $t \in C$ such that $t_i = v$ and $t_j \in \text{dom}(x_j)$ for any variable $x_j \in \text{Var}(C)$.

We assume there exists a total ordering among the values in the universe D i.e., for any two elements v and u in D, the transitive operator $v < u$ is either true or false but never undefined. An *interval support* is an approximation of the *domain support*. Each variable domain $\text{dom}(x_i)$ is approximated by the interval $[\min(\text{dom}(x_i)), \max(\text{dom}(x_i))]$. If all variable domains are intervals, then the interval support is equivalent to the domain support.

Definition 2.4 (Interval Support) Given a constraint C and a variable $x_i \in Var(C)$, a value $v \in \text{dom}(x_i)$ has an *interval support* if there exists an assignment $t \in C$ such that $t_i = v$ and $\min(\text{dom}(x_j)) \leq t_j \leq \max(\text{dom}(x_j))$ for any variable $x_j \in \text{Var}(C)$.

Clearly, removing from a variable domain a value that does not have a support preserves the set of solutions. One can consequently reduce the search space by successively removing values that do not have a support for a constraint. A constraint is *consistent* when there are no more unsupported values in the domains. From the two types of support defined above, we define four different local consistencies. A constraint C is

1. *domain consistent* if for each variable $x_i \in \text{Var}(C)$, all values in $\text{dom}(x_i)$ have a domain support.

2. *range consistent* if for each variable $x_i \in \text{Var}(C)$, all values in $\text{dom}(x_i)$ have an interval support.

3. *bounds consistent* if for each variable $x_i \in \mathrm{Var}(C)$, both values $\min(\mathrm{dom}(x_i))$ and $\max(\mathrm{dom}(x_i))$ have an interval support.

4. *bounds($\mathcal{D}$) consistent* if for each variable $x_i \in Var(C)$, both values $\min(\mathrm{dom}(x_i))$ and $\max(\mathrm{dom}(x_i))$ have a domain support.

Bounds consistency was first introduced for solving numerical problems [49, 33]. The term *bounds consistency* is actually used for different types of consistencies. See [14] for a disambiguation. In this thesis, we will restrict ourselves to the definition stated above.

We can introduce a partial order among consistencies in which consistency A is stronger than consistency B if all problems that are A-consistent for a constraint C_j are also B-consistent for the same constraint [16]. In this case, we write $A \succeq B$. Enforcing a stronger consistency prunes to a greater extent the variable domains than a weaker consistency but usually requires more computational power. A weaker consistency is generally faster to enforce but prunes less the domains, resulting in a longer backtracking search. The best choice of consistencies depends on the problem to solve and generally requires some experiments to determine.

In the consistencies studied above, domain consistency is stronger than range consistency since every value that has a domain support also has an interval support. Range consistency is stronger than bounds consistency since more values in the former consistency have an interval support. Example 2.3 illustrates this comparison. Figure 2.3 shows the partial order among the consistencies.

Example 2.3 Consider the following variable domains $\mathrm{dom}(x_1) = \{1, 3, 5\}$, $\mathrm{dom}(x_2) = \mathrm{dom}(x_3) = \mathrm{dom}(x_4) = \{2, 4\}$, and an ALL-DIFFERENT constraint over all these variables that ensures that all variables are assigned to distinct values. The variable domains are bounds consistent since the smallest value and the greatest value of each variable domain have an interval support.

Value 3 in $\mathrm{dom}(x_1)$ does not have an interval support. In order to make the problem range consistent, we have to remove 3 from the domain of x_1.

Finally, none of the values have a domain support since there exist no solutions

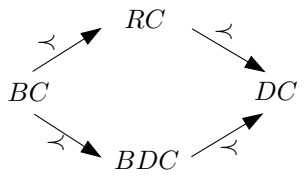

Figure 2.3: Relationship between bounds consistency (BC), range consistency (RC), bounds($\mathcal{D}$) consistency (BDC), and domain consistency (DC).

satisfying the constraint with the given variable domains. When testing for domain consistency, we detect that the constraint is unsatisfiable.

Domain consistency, range consistency, and bounds consistency are canonical in the field but the bounds($\mathcal{D}$) consistency mentioned in [14] is rarely used in practice. Generally, testing for domain support is more expensive than testing for bounds support. Moreover, it is often the case that when testing for a domain support of one value, we implicitly test for the domain support of all values in the variable domains. Therefore, domain consistency seems to offer a stronger consistency than bounds($\mathcal{D}$) consistency for the same computational cost. For this reason, we will focus our study on the first three consistencies and will no longer mention the bounds($\mathcal{D}$) consistency.

Enforcing a consistency on a constraint C_j consists in removing from the domains of the variables $\text{Var}(C_j)$ the values that do not have a support. Enforcing domain consistency on constraint C_1 in Example 2.2 would remove the values 1 and 2 from $\text{dom}(x_2)$ since both values cannot be greater than the smallest value in $\text{dom}(x_1)$. Similarly, the value 4 would be removed from the domain of x_1. If a variable domain becomes empty after removing all its values then the constraint is unsatisfiable. Each constraint requires a specialized filtering algorithm called a *propagator* to remove values that do not have a support.

Definition 2.5 (Propagator) A *propagator* is an algorithm associated to a particular constraint C that removes from the domains of the variables in $\text{Var}(C)$ some values that do not have a support for this constraint.

A propagator achieves domain, range, or bounds consistency on a constraint C if, after being executed, the constraint C is domain, range, or bounds consistent.

Finally, we define what is consistency at the level of the problem. The level of consistency for each constraint is implicit.

Definition 2.6 (Local Consistency) A problem is said to be *locally consistent* if all its constraints are consistent.

2.3.4 Propagation

Searching in a solution tree is a commonly used technique to solve CSPs. The algorithm chooses a variable x_i and instantiates it to a value v in its domain by eliminating all values in $\text{dom}(x_i)$ but v. It then makes the problem locally consistent by iteratively enforcing a consistency on the constraints by calling the constraint propagators. If not all domains are bounded by a single value, another variable is instantiated and the operation is repeated. If a constraint cannot be satisfied, the algorithm backtracks to the previous instantiation and tries to assign another value to the variable. The search reaches a solution when all variable domains contain a single value and all constraints are satisfied.

To efficiently make a problem locally consistent, constraint programming solvers such as ILOG [3] maintain a queue Q of modified variables that initially only contains the newly instantiated variable x_i. The algorithm pops a variable x_p from the queue and enforces consistency on all constraints restricting variable x_p. If a variable domain gets modified when enforcing consistency, it is pushed on the queue unless it is already there. The problem becomes locally consistent when the queue gets empty. The whole process is called *constraint propagation*. The algorithm shown in Figure 1 performs constraint propagation until the problem becomes locally consistent.

It is also possible to maintain a queue of constraints. A constraint C is pushed on the queue whenever the domain of a variable $x_i \in \text{Var}(C)$ is modified. ILOG [3] uses both a queue of variables and a queue of constraints. Constraints that require

Algorithm 1: LocalConsistency(x_m) makes a problem locally consistent.

```
Input: The variable x_m that has been instantiated
Q ← {x_m}
while Q ≠ ∅ do
    x_i ← pop(Q)
    for C_j such that x_i ∈ Var(C_j) do
        // Function prune enforces consistency on constraint C_j
        // and returns a list of modified variables.
        M ← prune(C_j)
        if ∃ x_k ∈ M, dom(x_k) = ∅ then
            The problem is unsatisfiable
        Q ← Q ∪ M
```

more computational power to propagate are pushed on the queue of constraints and are propagated only when the queue of variables becomes empty.

There are several known heuristics which accelerate the search for a solution. For example, the choice of which variable to instantiate is an important one. Gent et al. [24] compare different heuristics such as choosing the variable with the smallest domain first, the most constrained variable first, the variable that maximizes the solution density first, or the variable that maximizes the expected number of solutions first. A depth-first search is commonly used to explore the search tree but other searches can also be used. For instance, when the minimum discrepancy search [86] backtracks, it does not necessarily backtrack to the node representing the last variable instantiation but rather backtracks to a visited node that is *closer* to the solution.

2.3.5 Constraints

In this thesis, we will closely study two constraints in particular: the ALL-DIFFERENT and the Global Cardinality Constraint (GCC) both introduced by Régin in [67, 68].

Definition 2.7 (All-Different) The constraint ALL-DIFFERENT($x_1, \ldots, x_n$) is satisfied if and only if all variables are assigned to distinct values. Formally, the

ALL-DIFFERENT constraint is satisfied if and only if $x_i \neq x_j$ for all $i \neq j$.

The ALL-DIFFERENT constraint naturally occurs in many CSPs such as scheduling problems. For example, consider a set of tasks contending for a single non-shared resource. We can model this problem in a CSP with an ALL-DIFFERENT constraint over the execution time variables. Another example is the Golomb ruler problem (see problem 6 in CSPLib [25]) in which a ruler is to be assigned n marks for which the $\frac{n(n-1)}{2}$ possible distances between two marks are all distinct. The ALL-DIFFERENT constraint guarantees the distinctness between all distances. This problem has applications in x-ray crystallography and radio astronomy. The n-queens problem consists of finding a placement of n queens on a $n \times n$ chessboard such that no queen attacks another queen. The n-queens problem can be modeled with 3 ALL-DIFFERENT constraints: one for the rows and one for each of the diagonals. The columns are implicitly different by the formulation of the problem. The n-queen problem is similar to the quasigroup problem (problem 3 in CSPLib [25]) that requires to colour the cells of an $n \times n$ table with n colours such that cells on a same row or column are of different colours. Finally, the ALL-DIFFERENT constraint appears in industrial problems such as the car sequencing problem (see problem 1 in CSPLib [25]) that requires an ordering of cars on an assembly lane such that machines assembling different options are not overloaded.

The global cardinality constraint is a generalization of the ALL-DIFFERENT constraint.

Definition 2.8 (Global Cardinality Constraint) The global cardinality constraint $\mathrm{GCC}(x_1, \ldots, x_n, l, u, D)$ holds if and only if for each value v in D, at least $l[v]$ and at most $u[v]$ variables are assigned to v.

If l is the null vector and u the vector whose components are equal to one, we obtain the ALL-DIFFERENT constraint.

The GCC occurs in many scheduling problems where resources are limited. For example, in a scheduling problem with k processors, the GCC ensures that no more than k scheduling time variables are assigned to the same time slot. In other problems, the variables t_i may represent tasks to be accomplished by a group of

people i.e., the assignment $t_i = John$ indicates that John will execute task number i. A GCC on the task variables with $l_v = 1$ and $u_v = 2$ guarantees that each individual will accomplish at least 1 task but not more than 2 tasks.

Chapter 3

Existing Propagators for the All-Different and GCC Constraints

3.1 Introduction

In this chapter, we present the main known algorithms developed by others to propagate the All-Different and GCC constraints. We present the propagators according to the level of consistency they achieve. This order is similar to the chronological order of publication. In Chapters 4 and 5, in turn, we introduce new propagators that improve those presented in this chapter.

3.2 Binary Constraints

Recalling that the All-Different constraint is satisfied when all variables are pairwise different, a simple way to propagate the constraint is to post $\frac{n(n-1)}{2}$ difference constraints ($\neq$) between each pair of variables. The All-Different constraint is therefore decomposed into simpler constraints to be propagated.

The constraint $x \neq y$ is propagated as follows. When the domain of variable x contains a single value, this value is removed from the domain of y. The same rule applies when the domain of y contains only one value. When simultaneously considering the $\frac{n(n-1)}{2}$ binary constraints, we obtain the following simple rule: if a variable contains a single value in its domain, this value is removed from all other variable domains.

The propagator only needs to be triggered when a variable domain gets bound to a single value. When it happens, the propagator prunes the domains of other variables in $O(n)$ steps.

This technique is simple and easy to implement but lacks pruning power, as the following example illustrates.

Example 3.1 Consider the following variable domains subject to an ALL-DIFFERENT constraint: $\text{dom}(x_1) = \text{dom}(x_2) = \{1, 3\}$ and $\text{dom}(x_3) = \{1, 2, 3\}$. The values 1 and 3 from the domain of x_3 do not have a domain support. A propagator using binary constraints of difference is unable to detect these values since all variable domains contain more than one element.

3.3 Domain Consistency

Régin [67] was the first to consider merging the $\frac{n(n-1)}{2}$ constraints of difference into a single global constraint which is the ALL-DIFFERENT constraint. His goal was to obtain a higher level of consistency able to detect all unsupported values in the variable domains, i.e. to achieve domain consistency.

3.3.1 ALL-DIFFERENT

Given an ALL-DIFFERENT constraint, Régin's algorithm constructs a bipartite graph G called the *value-graph*. There is a node in graph G for each variable x_i and a node for each value v. There is an edge between variable-node x_i and the value-node v if and only if value v belongs to the domain of x_i, i.e. $v \in \text{dom}(x_i)$.

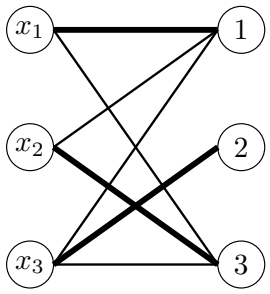

Figure 3.1: Value-graph of Example 3.1. Bold edges form a matching of maximum cardinality

Figure 3.1 shows the value graph corresponding to the problem described in Example 3.1 in Section 3.2.

Clearly, there is a one to one relationship between an assignment satisfying the ALL-DIFFERENT constraint and a matching of maximum cardinality in the value graph G. For instance, bold edges in Figure 3.1 form a matching of cardinality 3. The matching also corresponds to the assignment $x_1 = 1, x_2 = 3, x_3 = 2$.

Régin draws two conclusions from the relationship between assignments and matchings. First, for a problem with n variables, there is a matching of cardinality n if and only if the constraint is satisfiable. Second, an edge (x_i, v) belongs to a matching of cardinality n if and only if value v has a domain support in $\text{dom}(x_i)$. Consequently, propagating the ALL-DIFFERENT constraint consists of determining if there exists at least one matching of cardinality n to ensure the satisfiability of the constraint. In a second pass, one needs to mark all edges that could belong to such matching. Each edge remaining unmarked corresponds to a value to be removed from a variable domain.

As seen in Section 2.2.2, using the Hopcroft-Karp algorithm, we can find a matching of maximum cardinality in $O(\sqrt{n}m)$ steps where n is cardinality of the maximum matching and m the number of edges. In our case, we have $n = |X|$ and $m = |X||D|$; hence we can find the matching in $O(|X|^{1.5}|D|)$ time where $|X|$ is the number of variables and $|D|$ the number of distinct values in the domains. Otherwise, if the cardinality of the maximum matching is inferior to the number

of variables, we conclude that the constraint is unsatisfiable. If the maximum matching M contains $|X|$ edges, then we now proceed to the identification of the edges that do not belong to any maximum matching.

Régin's algorithm first creates a residual graph G_M which is a directed version of the graph G. Edges that belong to the maximum cardinality matching M are oriented from the variable-nodes to the value-nodes. Edges that do not belong to M are oriented from value-nodes to variable-nodes. From Theorem 2.3, we know that all edges that belong to a cycle of G_M belong to a matching of maximum cardinality. Régin therefore finds all strongly connected components in G_M using an algorithm for that end by Tarjan [80] and marks all edges that connect two nodes in the same component. Those edges necessarily belong to at least one maximum cardinality matching. A simple depth-first search (DFS) starting from free value-nodes marks all edges that lie on an even-length path starting at a free node. Theorem 2.3 in Section 2.2.2 certifies that these edges are part of at least one maximum matching. Moreover, all unmarked edges that do not belong to the matching M do not belong to any other maximum cardinality matching. An unmarked edge (x_i, v) that is not in M signifies that the value v does not have a domain support in $\text{dom}(x_i)$. Such values are removed from the variable domains and the problem becomes domain consistent.

From Example 3.1, given the maximum matching represented in Figure 3.1, we obtain the residual graph depicted in Figure 3.2. Notice that the edges $(x_3, 1)$, and $(x_3, 3)$ do not belong to any even-length path starting from a free node or any cycle. Therefore, these edges do not belong to any maximum matching. Values 1 and 3 should be removed from the domain of x_3 since they do not have any domain support.

Régin shows how to make his algorithm dynamic, i.e. being able to reuse previous computations to reestablish domain consistency on the variables. Suppose that during the constraint propagation phase another constraint removes a value v from a variable domain $\text{dom}(x_i)$. Two situations could occur. Either the edge (x_i, v) belongs to the previously computed matching M, or it does not. If the edge (x_i, v) does belong to the matching M, then we no longer have a matching of cardinality $|X|$. We can perform in $O(|X||D|)$ steps a DFS to find a path connecting the free

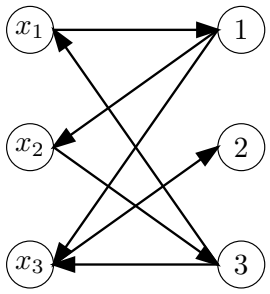

Figure 3.2: Residual graph associated to the matching represented in Figure 3.1

variable node x_i to a free value node. Applying this augmenting path to the matching reestablishes a matching of cardinality $|X|$. If the removed edge (x_i, v) does not belong to the matching M, we still have a matching of cardinality $|X|$ and we do not need to find an augmenting path. Once we have a matching of cardinality $|X|$, we only need to recompute the strongly connected components and find the even-length paths starting from a free node to prune the domains. This can be done in $O(|X||D|)$ steps. In summary, two cases can occur: the edge (x_i, v) belongs to the matching M or it does not belong to the matching M. In either case, the algorithm maintains domain consistency in $O(|X||D|)$ steps instead of $O(|X|^{1.5}|D|)$ steps.

3.3.2 GCC

Régin [68] modified his algorithm for the ALL-DIFFERENT constraint to propagate the global cardinality constraint (GCC). The approach is similar to the one used for the ALL-DIFFERENT constraint but uses flow theory instead of matching theory. The former is a generalization of the latter. The following example illustrates the tasks that the propagator needs to accomplish.

Example 3.2 Let the variables $x_1, \ldots, x_6$ have the domains $\text{dom}(x_1) = \{2\}$, $\text{dom}(x_2) = \{1, 2\}$, $\text{dom}(x_3) = \{2, 3\}$, $\text{dom}(x_4) = \{2, 3\}$, $\text{dom}(x_5) = \{1, 2, 3, 4\}$, and $\text{dom}(x_6) = \{3, 4\}$ and a single global cardinality constraint $\text{GCC}(x_1, \ldots, x_6, l, u)$ with the following lower and upper bounds on the occurrences of values,

v	1	2	3	4
l_v	0	1	1	2
u_v	3	2	1	3

Value 4 has to be assigned to at least 2 variables and only variables x_5 and x_6 contain it in their domain. Clearly, variables x_5 and x_6 must be assigned to value 4. Together, values 2 and 3 can be assigned to at most 3 variables and variables x_1, x_3, and x_4 can only be assigned to one of these values. Therefore, 2 and 3 must be removed from the domain of all variables except x_1, x_2, and x_4. Enforcing domain consistency results in the following domains: $\text{dom}(x_1) = \{2\}$, $\text{dom}(x_2) = \{1\}$, $\text{dom}(x_3) = \{2, 3\}$, $\text{dom}(x_4) = \{2, 3\}$, $\text{dom}(x_5) = \{4\}$, and $\text{dom}(x_6) = \{4\}$.

Régin [68] proved a one-to-one relationship between an assignment that satisfies the GCC and a maximum flow in a graph. In his proof, he constructs a value graph of the problem where each variable x_i is a variable-node and each value v is a value-node. There is an edge (x_i, v) if and only if value v belongs to the domain of x_i. A source node s is connected to all variable nodes and all value nodes are connected to a sink node t. Each edge is associated to two values: a lower and an upper capacity. The lower capacity of an edge connecting a value node v to the sink is l_v while its upper capacity is u_v. All other edges have a null lower capacity and a unit upper capacity. Figure 3.3 shows the value graph corresponding to Example 3.2.

As for the All-Different constraint, Régin uses Theorem 2.3 in Section 2.2.2 to find edges that belong to a maximum flow. He first computes in $O(|X|^2|D|)$ steps a maximum flow f using the Ford-Fulkerson algorithm. According to Theorem 2.2 in Section 2.2.1, an edge can carry a flow in some maximum flow if and only if it carries a flow in f or it belongs to a cycle in the residual graph G_f. Therefore, enforcing domain consistency on the GCC is reduced to computing a feasible flow f and finding the strongly connected components in the residual graph G_f. Finding the strongly connected components requires $O(|X||D|)$ steps. Therefore, the total running time complexity of the propagator is $O(|X|^2|D|)$.

In contrast to the binary constraint representation, the global formulation of the All-Different constraint and the GCC allow us to prune all inconsistent values with respect to the constraints. This extra pruning is achieved at a higher cost since

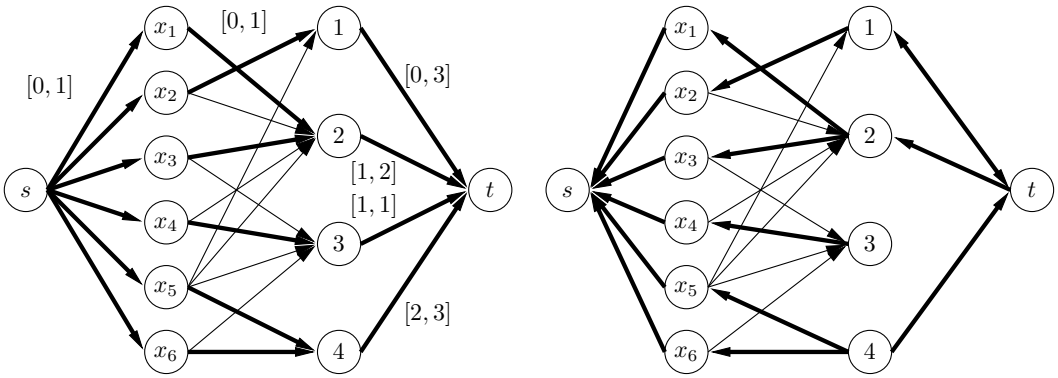

Figure 3.3: a) Régin's value graph for the Global Cardinality Constraint. Bold edges are edges admitting a flow of 1. Other edges have a null flow. This flow corresponds to the assignment $x_1 = 2$, $x_2 = 1$, $x_3 = 2$, $x_4 = 3$, $x_5 = 4$, and $x_6 = 5$ b) The residual graph associated to the assignment.

the algorithms are more complex. Experiments prove that most of the time, for the ALL-DIFFERENT and the GCC constraints, it is more efficient overall to spend more time pruning more values. However, there are some specific problems for which it is advantageous to prune the domains less but at a higher speed (see [46, 63] for instance). The next sections describe weaker consistencies that are faster to enforce.

3.4 Range Consistency

Based on the success of Régin [67] with the ALL-DIFFERENT constraint, Leconte [46] designed an algorithm for range consistency. This consistency is weaker than domain consistency but is easier to enforce.

Leconte's algorithm approximates all variable domains with intervals. For instance, if the domain of variable x_i is the set $\{2, 5, 8\}$, his algorithm approximates $\text{dom}(x_i)$ with the interval $[2, 8]$. We call the *lower bound* of a variable the smallest element in its domain. Similarly, we call the *upper bound* of a variable the highest element in its domain. Given the new problem where all variable domains are

intervals, removing all inconsistent values results in enforcing range consistency in the original problem. In other words, enforcing domain consistency on the approximated problem enforces range consistency on the original problem.

Leconte was the first to use the notion of *Hall interval* to propagate the ALL-DIFFERENT constraint.

Definition 3.1 (Hall Interval) A Hall interval is an interval H such that there are exactly $|H|$ variables whose domains are contained in H.

The following example illustrates the concept of Hall interval.

Example 3.3 Consider the following variable domains subject to an ALL-DIFFERENT constraint: $\text{dom}(x_1) = \text{dom}(x_2) = [3, 4]$, $\text{dom}(x_3) = [2, 3]$, and $\text{dom}(x_4) = [1, 5]$. The interval $[3, 4]$ is a Hall interval since 2 variables, x_1 and x_2, have their domain contained in this interval of size 2. The interval $[2, 4]$ is also a Hall interval since this interval of 3 elements contains the domains of 3 variables, namely x_1, x_2, and x_3. There are no other Hall intervals in this problem.

The values in a Hall interval H are fully consumed by the variables that form the Hall interval and become unavailable for all other variables. In Example 3.3, because of the Hall interval $[3, 4]$, we know that values 3 and 4 will be assigned to variables x_1 and x_2. These values become unavailable for other variables like x_3 and x_4. The interval $[3, 4]$ should therefore be removed from $\text{dom}(x_3)$ and $\text{dom}(x_4)$. To enforce range consistency, it is necessary and sufficient to remove all Hall intervals from the domains of the variables that are not fully contained in these Hall intervals.

Leconte's algorithm considers all intervals $[a, b]$ where a is the lower bound of a variable domain and b is an upper bound and checks if the interval contains $b-a+1$ variable domains. Such an interval is a Hall interval and its values are removed from every variable domains that are not contained in $[a, b]$. The algorithm uses a proper iteration over all variables in order to reach a running time complexity of $\Theta(n^2)$. Figure 2 shows the pseudo-code of the algorithm[1].

[1]The last `for` loop has been modified to correct a flaw in the original pseudo-code

Algorithm 2: Leconte's algorithm [46] enforces range consistency in $O(n^2)$ steps.

```
for x_j in decreasing order of max(dom(x_j)) do
    count ← 0;
    k ← −1;
    for x_i in decreasing order of min(dom(x_i)) do
        if k ≥ 0 then
            Remove [min(dom(x_k)), max(dom(x_j))] from dom(x_i);
        if max(dom(x_i)) ≤ max(dom(x_j)) then
            count ← count + 1;
            if count = max(dom(x_j)) − min(dom(x_i)) + 1 then
                k ← i;
    for x_i such that max(dom(x_j)) < max(dom(x_i)) do
        if min(dom(x_k)) ≤ min(dom(x_i)) then
            Set minimum of dom(x_i) to min(dom(x_k)) + 1;
```

Leconte shows that his algorithm is optimal in the worst case with the following example.

Example 3.4 (Leconte 96 page 24 [46]) *Let $x_1, \ldots, x_n$ be n variables subject to an* ALL-DIFFERENT *constraint whose domains contain all distinct odd numbers ranging from 1 to $2n-1$ and let $x_{n+1}, \ldots, x_{2n}$ be n variables whose domains are $[1, 2n-1]$. An algorithm maintaining range consistency needs to remove the n odd numbers from each of the n variable domains which is done in $\Theta(n^2)$.*

This example proves that some instances of the problem require $\Theta(n^2)$ steps to achieve range consistency. However, in practice, this situation rarely occurs. In the next chapter, we will present an output sensitive algorithm that runs in $O(n^2)$ on the problem presented in Example 3.4 but can run in amortized linear time on a sequence of similar problems as is usually the case in a backtracking search.

3.5 Bounds Consistency

Puget [63] was the first to design a propagator for the bounds consistency of the ALL-DIFFERENT constraint. As in range consistency, the strategy consists in approximating the variable domains with the smallest covering intervals, i.e. $\text{dom}(x_i) \approx [\min(\text{dom}(x_i)), \max(\text{dom}(x_i))]$. To achieve bounds consistency, intervals must be shrunk until all interval bounds have an interval support. It is not necessary to create holes in the intervals, only the bounds are updated. Bounds consistency is the most studied form of consistency by number of publications on the topic (e.g. [39, 51, 54, 63, 66]). We will highlight the main contributions in this section, and we will present our own contributions in Chapters 4 and 5.

3.5.1 Puget's Propagator

Puget [63] discovered a very useful property while designing the first propagator for the bounds consistency of the ALL-DIFFERENT constraint. He found that in order to achieve bounds consistency, it is sufficient to design an algorithm that only shrinks the lower bounds of the variable domains. The upper bounds can be shrunk by constructing a symmetric problem where all variable domains are negated, i.e. $\text{dom}(x'_i) = [-\max(\text{dom}(x_i)), -\min(\text{dom}(x_i))]$. The algorithm that shrinks the lower bound can then be applied on the modified problem to obtain the new upper bounds for the original variable domains.

Puget's algorithm detects some Hall intervals $H = [a, b]$. Variable domains $\text{dom}(x) = [c, d]$ having their lower bound in H but their upper bound outside of H are shrunk to the interval $[b + 1, d]$. To achieve a better running time complexity than Leconte's $O(n^2)$ algorithm for range consistency, Puget understood that some Hall intervals must be ignored. Indeed, a problem with n variables can have as many as n^2 Hall intervals. Let $H_1 = [a_1, b], H_2 = [a_2, b], \ldots, H_m = [a_m, b]$ be m different Hall intervals sharing the same upper bound b and let the *left-most Hall interval* be the Hall interval with the smallest lower bound. Puget's algorithm only uses left-most Hall intervals to shrink the variable lower bounds. The number of left-most Hall intervals is bounded by n. Using a balanced binary tree to store

these Hall intervals, Puget achieves bounds consistency in $O(n \log n)$ steps.

Puget proved that it is possible to achieve bounds consistency within a better running time complexity than range consistency. His algorithm was later improved in both the theoretical and the empirical sense.

3.5.2 Mehlhorn and Thiel's Propagator

Mehlhorn and Thiel [54] were the first to propose a linear time algorithm for the bounds consistency of the ALL-DIFFERENT constraint. Their approach is similar to that of Régin for domain consistency since they also use matching theory. They construct a similar value graph as Régin, i.e. a bipartite graph with a node for each variable and each distinct value in the domains. The graph differs from Régin's graph on the set of edges. Their graph G has an edge between a variable node x_i and a value node v if and only if the value v lies between $\min(\text{dom}(x_i))$ and $\max(\text{dom}(x_i))$. In order to keep the running time complexity linear in terms of variables, the algorithm must take great care to never enumerate or iterate through the entire list of edges since their number might be quadratic in the number of variables.

The constructed graph leads to a restricted class of bipartite graphs called convex bipartite graphs.

Definition 3.2 (Convex bipartite graph) A convex bipartite graph $G = (L, R, E)$ is a bipartite graph whose right-nodes R can be labeled with numbers from 1 to $|R|$ such that the neighbors of every left-node $l \in L$ have labels forming an interval.

Clearly, the value graph for the ALL-DIFFERENT constraint when all variable domains are intervals is a convex bipartite graph. Algorithms operating on this class of graphs are generally more efficient. Given a lower-bound and an upper-bound ordering of the variable domains, Mehlhorn and Thiel compute a maximal matching in $O(|X|)$ steps which is independent of the domain sizes. Then they compute and mark the strongly connected components and observe some properties that allow to mark the nodes on an even-length path leading to a free node in $O(|X|)$ steps

regardless of the number of value nodes. Finally, their algorithm finds the smallest and highest marked value in each variable domain and assigns the lower and upper bound of the domain to these values. The time complexity of their propagator is $O(t + |X|)$ where t is the time required for sorting the $|X|$ variables by lower and upper bounds.

3.5.3 Katriel and Thiel's Propagator

Katriel and Thiel [39, 40] generalized the propagator for the bounds consistency of the ALL-DIFFERENT constraint developed by Mehlhorn and Thiel [54] in order to enforce the same consistency on the GCC. Since GCC requires a flow instead of a matching, the authors developed an algorithm to compute a flow in a convex bipartite graph.

We recall that we are looking for a flow f where the amount $f(e)$ of flow on edge $e \in E$ lies between two constants $l(e)$ and $u(e)$. Katriel and Thiel designed an algorithm that proceeds in three phases to find such a flow in a convex bipartite graph. In the first phase, the algorithm finds a flow satisfying the condition $f(e) \leq u(e)$ for each edge $e \in E$. In the second phase, the algorithm finds a flow that satisfies the inequality $f(e) \geq l(e)$ for each edge $e \in E$. On the third phase, both flows are merged together in order to fully satisfy the capacity conditions, i.e. $l(e) \leq f(e) \leq u(e)$ for all $e \in E$.

Once the flow has been found, the algorithm proceeds as usual by marking all edges in the same strongly connected component or on an even-length path leading to a free node. The algorithm assigns as new lower bound for the domain the first marked value in this domain. Similarly, the highest marked value in a variable domain becomes the new upper bound.

The algorithm operates in $O(|X| + |D|)$ steps where $|X|$ is the number of variables and $|D|$ the number of distinct values in the variable domains.

Pruning the Cardinality Variables

The algorithm Katriel and Thiel developed in [39] does not only achieve bounds consistency on the variable domains, it also achieves bounds consistency on the cardinality variables. The cardinality variables occur in a generalization of the GCC called the extended global cardinality constraint (EXT-GCC).

Definition 3.3 (Extended Global Cardinality Constraint) The extended global cardinality constraint EXT-GCC($[x_1, \ldots, x_n], [C_1, \ldots, C_m], D$) holds if and only if for each value v in D, there are C_v variables among $x_1, \ldots, x_n$ assigned to v.

To enforce bounds consistency on the cardinality variables, one has to determine what is the minimum number and the maximum number of variables that can be assigned a specific value. The following example shows the result of enforcing bounds consistency on the EXT-GCC.

Example 3.5 Consider the following variable domains $\text{dom}(x_1) = \text{dom}(x_2) = [1, 2]$, $\text{dom}(x_3) = [2, 4]$, and $\text{dom}(x_4) = [3, 4]$ and the following cardinality variable domains $\text{dom}(C_1) = \text{dom}(C_2) = \text{dom}(C_4) = [0, 1]$, and $\text{dom}(C_3) = [0, 3]$. Enforcing bounds consistency on EXT-GCC($x_1, \ldots, x_4, C_1, \ldots, C_4, [1, 4]$) results in the following variable domains: $\text{dom}(x_3) = [3, 4]$, $\text{dom}(C_1) = \text{dom}(C_2) = [1, 1]$, $\text{dom}(C_3) = [1, 2]$, and all other variable domains remain unchanged.

Katriel and Thiel enforce bounds consistency on the cardinality variables by iteratively modifying the flow on the value graph. Each modification to the flow ensures that the maximum amount of flow is pushed on the edge connecting a specific value node to the sink. This amount is exactly the value corresponding to $\max(\text{dom}(C_v))$. A similar algorithm is derived for the lower bounds of the cardinality variables.

3.6 Variations on the Problem

There exist many variations of the ALL-DIFFERENT and the GCC constraints. Régin [70] describes a *cost-based* GCC where an assignment between a variable x_i

and a value v generates a cost of c_v^i. In order to satisfy the constraint, the sum of the costs associated to each variable must lie in a given interval. The algorithm given has running time complexity of $O\left(|X|\, S(|X||D|, |X|+|D|)\right)$ where $S(m,n)$ is the time required to find a shortest path in a graph with n nodes and m edges. The propagator for the Cost-GCC is similar to the one for the GCC. Minimum-cost maximum-flows (see [4]) are used instead of maximum flows.

The soft version of the All-Different constraint called Soft-All-Different is used in over-constrained problems, i.e. problems that do not have a solution satisfying all constraints. In many problems, having variables assigned to different values is more a preference than a requirement. The propagator of Petit et al. [60] relaxes the All-Different constraint and directs the search to a solution where the number of variables assigned to the same value is minimized. Unfortunately, their algorithm does not enforce domain consistency and suffers from a high running time complexity of $O(|X|^{3.5}|D|^2)$. Van Hoeve's propagator [83] fixes both deficiencies by enforcing domain consistency in $O(|X|^2|D|)$ steps. The same algorithm can be adapted for the soft version of GCC called Soft-GCC [85].

Régin and Gomes [72] propose the *cardinality matrix constraint*. Consider a set of variables disposed in a table with one variable per cell. The cardinality matrix constraint is a collection of GCC's: one for each row and each column. Enforcing domain consistency on this constraint is NP-Hard. Hence, the authors decompose the constraint into several constraints: one GCC per row, one GCC per column, and one 0/1-Cardinality-Matrix constraint for each value in the variable domains. They show how enforcing domain consistency on these constraints is strictly stronger ($\succeq$) than enforcing domain consistency on GCC's on rows and columns. The algorithms presented are similar to those used for the propagation of the GCC and the All-Different constraint.

Chapter 4

New Propagators for the All-Different Constraint

4.1 Introduction

In this chapter, we present our propagators for the All-Different constraints. In all cases, they improve either the theoretical or practical performance (or both) of the best known algorithms.

4.2 Bounds Consistency for the All-Different Constraint

We propose a propagator based on Hall's theorem that enforces bounds consistency on the All-Different constraint [51]. The algorithm runs in $O(t + |X|)$ where t is the time required to sort the set X of variables by lower and upper bounds. There exist algorithms able to sort the variables in linear time. In practice, bubble-sort offers the best performance since the propagator is generally called with the variable list almost sorted. The propagator we propose is simpler and outperforms in practice the one suggested by Mehlhorn and Thiel [54].

Our algorithm proceeds as follows. We process each variable by non-decreasing upper bounds. If the current variable domain $\mathrm{dom}(x_i) = [a, b]$ has its lower bound inside an already discovered Hall interval $H = [c, d]$, we shrink the interval by shifting the lower bound to $d + 1$. We then check if there is a Hall interval whose upper bound is b. If this is the case, we record the left-most Hall interval having this upper bound, i.e. the largest Hall interval with upper bound b. We shrink the upper bounds using a symmetric algorithm. To illustrate how variable domains are pruned, we use a simple example first introduced by Puget [63].

Example 4.1 Let $\mathrm{dom}(x_1) = [3, 4]$, $\mathrm{dom}(x_2) = [2, 4]$, $\mathrm{dom}(x_3) = [3, 4]$, $\mathrm{dom}(x_4) = [2, 5]$, $\mathrm{dom}(x_5) = [3, 6]$, and $\mathrm{dom}(x_6) = [1, 6]$ be six variable domains. Enforcing bounds consistency prunes the domains as follows: $\mathrm{dom}(x_1) = [3, 4]$, $\mathrm{dom}(x_2) = [2, 2]$, $\mathrm{dom}(x_3) = [3, 4]$, $\mathrm{dom}(x_4) = [5, 5]$, $\mathrm{dom}(x_5) = [6, 6]$, and $\mathrm{dom}(x_6) = [1, 1]$

An interval I can contain up to $|I|$ variable domains. If this limit is exactly met then I is a Hall interval. We describe a technique to compute the capacity of candidate Hall intervals. Each time we process a variable domain, we decrement by one the capacity of the interval into which the domain lands. When the capacity reaches 0, we report the Hall interval.

Let all n variable domains be semi-open intervals of the form $\mathrm{dom}(x_i) = [a_i, b_i)$ and let E be the set of endpoints such that $E = \{a_i\}_{i=1}^{n} \cup \{b_i\}_{i=1}^{n} \cup \{m, n\}$ where m is a sentinel value of at least two less than any other value in E and n is at least two more. We construct a union-find data structure S (see [21] for a survey on these structures) over the elements in E where each set is represented by a tree and labeled with the root of this tree. Initially, all elements in S are singletons. Let r_1 and r_2 be two adjacent roots such that $r_1 < r_2$. There is a directed edge (r_2, r_1) whose weight $\mathit{capacity}(r_2)$ represents the capacity of the semi-open interval $[r_1, r_2)$. In Example 4.1 we have $S = \{1, 2, 3, 5, 6, 7, 9\}$ and the tree data structure is as follows.

$$1 \xleftarrow{1} 2 \xleftarrow{1} 3 \xleftarrow{2} 5 \xleftarrow{1} 6 \xleftarrow{1} 7 \xleftarrow{2} 9$$

Adjacent trees can be merged together by making the root with the highest value the parent of the root with the lowest value.

We process variables by non-decreasing order of upper bound. Let $\text{dom}(x_i) = [a_i, b_i)$ be the current variable domain. Using the data structure S described above, one can compute the capacity of any interval $[d, b_i)$ such that $d \in S$ by summing up all capacity numbers on the path from b_i to the root of the tree that contains d. When we process a variable domain $\text{dom}(x_i) = [a_i, b_i)$, we look for the successor of a_i in S and find the root r_2 of its tree. Let r_1 be the root that precedes r_2 and r_3 the root that follows r_2. We decrement the capacity $\mathit{capacity}(r_2)$ of $[r_1, r_2)$ by 1. If it becomes null, we merge the tree rooted at r_2 with the tree rooted at r_3 by making r_2 a child of r_3. We also make r_3 point to r_1 since a root always points to the root of the preceding tree.

If after this operation, b_i is no longer a root, the interval $[r_1, b_i)$ has now a capacity of $\mathit{capacity}(r_3) - (r_3 - b_i) \leq 0$. If the capacity is null, we have exactly $b_i - r_1$ domains contained in the interval $[r_1, b_i)$ and by definition, $[r_1, b_i)$ is a Hall interval. If the capacity is negative, we are trying to assign $b_i - r_1$ values to more than $b_i - r_1$ variables which is impossible. The constraint is therefore unsatisfiable. The first graph of each step in Figure 4.1 shows the evolution of the data structure and indicates which Hall intervals are discovered. We keep the height of the trees low by means of a standard union-find path compression process [21].

We now show how to store Hall intervals in an efficient data structure. Observe that a Hall interval is necessarily of the form $[a, b)$ where $\{a, b\} \subset S$. We consider a new data structure where each endpoint in E is a node that points to its predecessor except for the first element in E that does not have a predecessor. If a Hall interval $[a, b)$ is discovered, we follow the path from b to a and make all elements on this path point to b. We also make b point to where a was originally pointing to. With this configuration, a lower bound a_i belongs to a Hall interval if it points to a greater element than itself. To retrieve the upper bound of this Hall interval, we follow the links from a_i until we reach a node b that points to a node smaller than itself. This element b is the open upper bound of a Hall interval $[a, b)$. Each time we follow links, we do path compression to make future calls faster. The second graph of each step in Figure 4.1 shows how Hall intervals are stored.

Putting it all together, we design an algorithm that updates both the capacity trees and the Hall intervals. Algorithm 3 uses the following variables and functions:

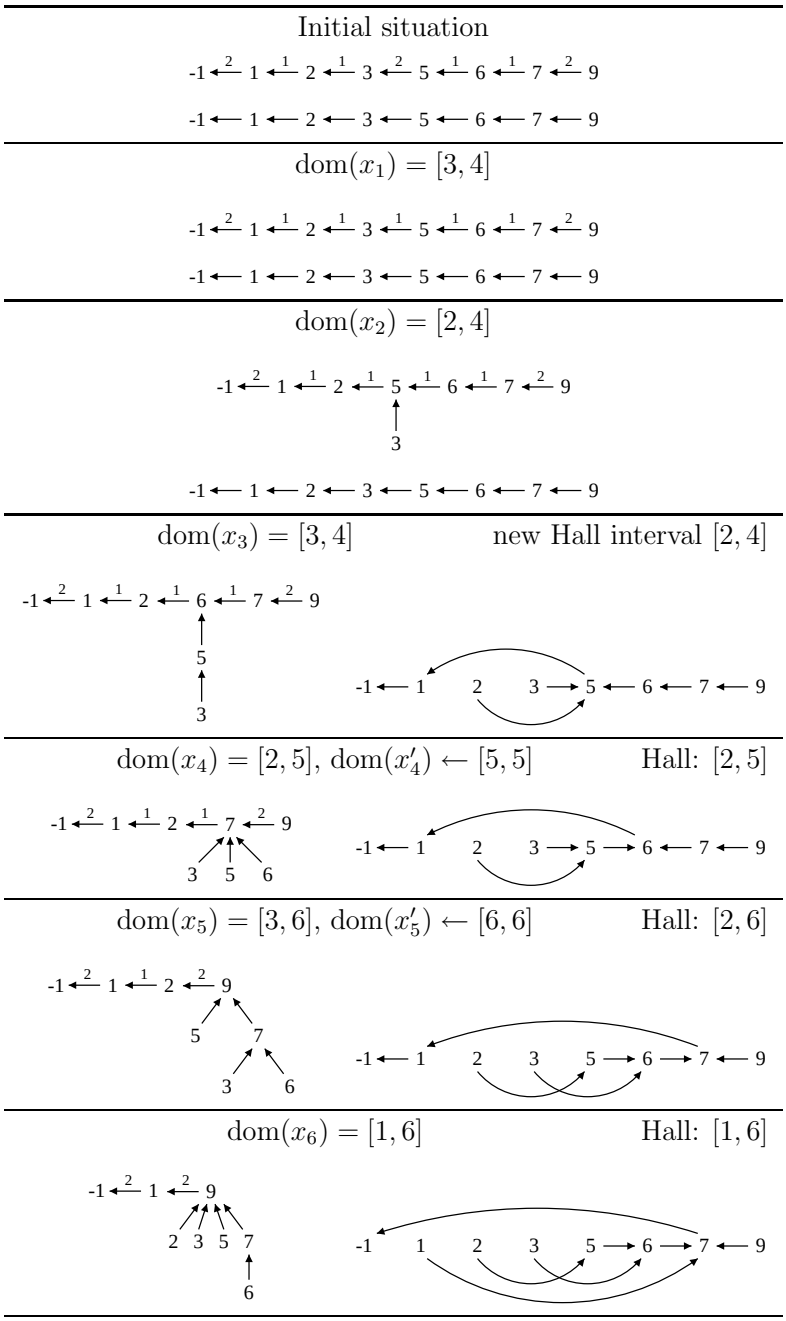

Figure 4.1: Trace of Example 4.1: each row represents an iteration where a variable is processed. The first graph illustrates the state for the vectors `bounds`, `t` and `d` while the second graph shows the state of the vectors `bounds` and `h`.

- $\mathsf{maxsorted}[0 \ldots |X| - 1]$ is the array of variables sorted by upper bounds.
- $\mathsf{bounds}[0 \ldots |S| - 1]$ is a sorted array containing the elements of S.
- $\mathsf{tree}[0 \ldots |S| - 1]$ are the pointers of the tree structure where $s_i \in S$ points to $\mathsf{tree}[i]$.
- $\mathsf{capacity}[0 \ldots |S| - 1]$ holds capacities such that $\mathsf{capacity}[i]$ is the capacity of the half-open interval $[\mathsf{bounds}[\mathsf{tree}[i]], \mathsf{bounds}[i])$.
- $\mathsf{minrank}$ and $\mathsf{maxrank}$ are the indices of the above array bounds such that $\mathsf{bounds}[\mathsf{minrank}(x_i)] = \min(\mathrm{dom}(x_i))$ and $\mathsf{bounds}[\mathsf{maxrank}(x_i)] = \max(\mathrm{dom}(x_i))$.
- $\mathsf{hall}[0 \ldots |S| - 1]$ is the Hall interval data structure such that if $\mathsf{hall}[i] < i$ then the half-open interval $[\mathsf{bounds}[\mathsf{hall}[i] + 1], \mathsf{bounds}[i])$ is a Hall interval. If $\mathsf{hall}[i] > i$ then $\mathsf{bounds}[i]$ belongs a Hall interval such that following the path $i, \mathsf{hall}[i], \mathsf{hall}[\mathsf{hall}[i]], \ldots$ until an element pointing to an element smaller than itself leads to the open-upper bound b of the Hall interval $[a, b)$.
- $\texttt{pathmax}(a, x)$ follows the pointers $x, a[x], a[a[x]], \ldots$ until an element points to an index smaller than itself. The function returns the greatest index visited.
- $\texttt{pathset}(a, x, b, c)$ follows the pointers $x, a[x], a[a[x]], \ldots$ until an element points to b and set all these pointers to c.

Algorithm 3: Enforcing Bounds Consistency on ALL-DIFFERENT(X)

```
Sort X by non-decreasing upper bounds
for i ← 1 to |S| − 1 do
    tree[i] ← hall[i] ← i − 1
    capacity[i] ← bounds[i] − bounds[i − 1]
for i ← 0 to |X| − 1 do
    x ← maxsorted[i].minrank, y ← maxsorted[i].maxrank
    z ← pathmax(tree, x + 1)
    j ← tree[z]
    capacity[z] ← capacity[z] − 1
    if capacity[z] = 0 then
        // Merge the tree rooted at z with the previous tree
           rooted at tree[z].
        tree[z] ← z + 1
        z ← pathmax(tree, tree[z])
        tree[z] ← j
    pathset(tree, x + 1, z, z)
    if capacity[z] < bounds[z] − bounds[y] then return Inconsistent
    if hall[x] > x then
        // The lower bound of dom(maxsorted[i]) is in a Hall
           interval
        w ← pathmax(hall, hall[x])
        maxsorted[i].min ← bounds[w] // Increase the lower bound
        pathset(hall, x, w, w)
    if capacity[z] = bounds[z] − bounds[y] then
        // Mark the newly discovered Hall interval
        pathset(hall, hall[y], j − 1, y)
        hall[y] ← j − 1
return Consistent
```

4.2.1 Time Complexity Analysis

To compute the running time complexity of the algorithm, note that each instruction is repeated at most $|X|$ times and all of them are constant time operations except for `pathset` and `pathmax`. Since each call to `pathmax` is followed by a call on `pathset` over the same nodes of the tree, the latter function determines the time complexity of the algorithm.

The cost of a call to `pathset` is highly dependent on the state of the forest of trees `tree`. If the trees are tall, a call to `pathset` is expensive; yet path compression makes subsequent calls cheaper for a greater number of nodes than before. Since the actual cost of a call to `pathset` varies during the execution of the algorithm, we will perform an amortized analysis in order to bound the cost of the function on a sequence of n calls.

The amortized analysis of our algorithm is similar to that of the union-find data structure with naive union and path compression (see [21] for a survey). Tarjan and Leeuwen [81] give an amortized analysis of this specific case. Hopcroft and Ullman [37] attribute this result to Paterson [58]. For the sake of clarity, we present the amortized analysis adapted to our problem.

Let the *reference trees* be the trees obtained at the end of the execution of the algorithm when path compression has been disabled, i.e. all calls to `pathset` are omitted. Let $h(a)$ be the height of node a in the reference trees. Let F_t be the forest generated by the algorithm after the t^{th} call to `pathset` with path compression enabled. Let $p_t(a)$ be the parent of node a in F_t. Notice that $h(a)$ refers to the reference trees while $p(n)$ refers to the actual trees F_t. Finally, we define the following potential function over each node.

$$\Phi_a(t) = \begin{cases} 0 & \text{If } a \text{ is a root} \\ \lfloor \log(h(p_t(a)) - h(a)) \rfloor & \text{otherwise} \end{cases}$$

Notice that this potential function can take a maximum of $\lceil \log(n) \rceil$ distinct values. Procedure `pathset` performs a path compression over all nodes on the path connecting a node a to the root r of the tree. There can be up to n nodes on this

path but no more than $\lceil \log(n) \rceil$ distinct potential values. For each potential value v, we choose the highest node i with $\Phi_i(t) = v$ and pay 1 unit of computation for it. This charges a cost of $O(n \log n)$ to procedure `pathset`. For all other nodes with potential value v, we prove that their potential function goes up by at least one.

Lemma 4.1 *Let i be the highest node with potential $\Phi_i(t)$ on the compression path and let j be a lower node with the same potential $\Phi_j(t) = \Phi_i(t)$. Then after path compression, we have $\Phi_j(t+1) \geq \Phi_j(t) + 1$.*

Proof The length of the path connecting j to the root of the tree r in the reference tree is necessarily greater than the sum of the length of the two sub-segments connecting j to its parent $p(j)$ and i to its parent $p(i)$. We have

$$\begin{aligned} h(r) - h(j) &\geq h(p(i)) - h(i) + h(p(j)) - h(j) && (4.1) \\ \lfloor \log(h(r) - h(j)) \rfloor &\geq \lfloor \log(h(p(i)) - h(i) + h(p(j)) - h(j)) \rfloor && (4.2) \end{aligned}$$

Suppose the segment connecting i to its parent $p(i)$ in the reference tree is longer than the segment connecting j to its parent $p(j)$. In other words, suppose $h(p(i)) - h(i) \geq h(p(j)) - h(j)$. We obtain.

$$\begin{aligned} \lfloor \log(h(r) - h(j)) \rfloor &\geq \lfloor \log(2(h(p(j)) - h(j))) \rfloor && (4.3) \\ \lfloor \log(h(r) - h(j)) \rfloor &\geq \lfloor \log(h(p(j)) - h(j)) \rfloor + 1 && (4.4) \\ \Phi_j(t+1) &\geq \Phi_j(t) + 1 && (4.5) \end{aligned}$$

On the other hand, if the segment from i to $p(i)$ is shorter than the segment from j to $p(j)$, i.e. $h(p(i)) - h(i) \leq h(p(j)) - h(j)$ we obtain the following inequalities.

$$\begin{aligned} \lfloor \log(h(r) - h(j)) \rfloor &\geq \lfloor \log(2(h(p(i)) - h(i))) \rfloor && (4.6) \\ \lfloor \log(h(r) - h(j)) \rfloor &\geq \lfloor \log(h(p(i)) - h(i)) \rfloor + 1 && (4.7) \\ \Phi_j(t+1) &\geq \Phi_i(t) + 1 && (4.8) \end{aligned}$$

Since both nodes i and j have the same potential at time t, this is equivalent to $\Phi_j(t+1) \geq \Phi_j(t) + 1$. In either case, the potential of node j increases by at least one. □

The following theorem concludes the running time analysis of the algorithm.

Theorem 4.1 *Algorithm 3, implemented with the tree data structure, has a running time complexity of* $\Theta(|X| \log |X|)$ *in the worst case.*

Proof We charged to procedure `pathset` a cost of $\Theta(\log |X|)$. The rest of the actual cost is absorbed by the potential function of each node. Since the potential of a node cannot increase more than $\log |X|$ times, a sequence of $|X|$ calls to `pathset` costs at most $O(|X| \log |X|)$.

Fischer [56] gives an example of an instance for which the calls to `pathset` takes $\Theta(|X| \log |X|)$ steps. This occurs when tree is a binomial tree and the `pathset` function is called on certain elements. The amortized complexity of $\Theta(|X| \log |X|)$ is therefore tight. □

However, the worst case complexity can be improved. The data structures tree and hall are in fact union-find data structures that support the operations FIND-MAX(e) which returns the largest value in the set containing e, FIND-PREV-MAX(e) which returns the largest element in the set preceding the set containing element e, and UNION(i, j) which merges the set containing element i with the set containing element j. The same operations can be realized with a weighted-union-find data structure such that each root holds the largest element in the tree and a pointer to the largest element in the previous tree. This brings the worst case complexity to $O(t+|X|\alpha(|X|))$ where α is the inverse of Ackermann's function and t the time for sorting $|X|$ variables by lower and upper bounds. This can be further improved to a running time of $O(t + |X|)$ by using Gabow and Tarjan's union-find data structure [20] we call the interval union-find data structure.

Theorem 4.2 *Algorithm 3, implemented with the interval union-find data structure, has a running time complexity of* $\Theta(t + |X|)$ *in the worst case.*

Proof The values in the union-set data structures used to store the capacities and the Hall intervals can be totally ordered in a sequence from the smallest value to the highest value. Each time the algorithm calls the UNION operator, it merges two sets A and B together. There always exist an element $a \in A$ and an element $b \in B$ such that a and b are adjacent in the ordered sequence. This condition is sufficient to use the interval union-find. The worst case running time complexity of a call to UNION and FIND with this data structure is constant time. Hence the running time complexity of $\Theta(t + |X|)$ for Algorithm 3. □

Despite our best efforts in optimizing the code, the $O(t + |X| \log |X|)$ version provides the best performance when run with input of reasonable size due to its small hidden constant. We recall that this is a worst case analysis and experiments show an almost linear behavior in the average case for our sample data set.

4.2.2 Experiments

We compare five propagators for the ALL-DIFFERENT constraint: (i) our propagator introduced in Section 4.2, (ii) our implementation of the propagator of Mehlhorn and Thiel [54], (iii) ILog's version of Leconte's propagator [46] for range consistency, (iv) Régin's propagator [67] for domain consistency, and (v) a propagator for value consistency. We use in all seven experiments the ILOG Solver 4.2 software library [3], a highly optimized library for constraint programming. We label our propagator BC, our implementation of Mehlhorn and Thiel's propagator MT, Leconte's propagator RC, Regin's propagator DC, and the value consistency propagator VC. We implemented Mehlhorn and Thiel propagator (MT) before having sketched our own bounds consistency propagator (BC) with the intention to make it as fast as possible. We do not have an implementation of Puget's algorithm [63] but using his experimental results and RC as a calibration point, we can perform a fair comparison with other algorithms over Puget's selection of problems. We believe the comparison to be accurate since we use a similar version of ILOG. All experiments ran on a 300 MHz Pentium II with 228 MB of memory. We averaged all results over 10 experiments unless otherwise stated.

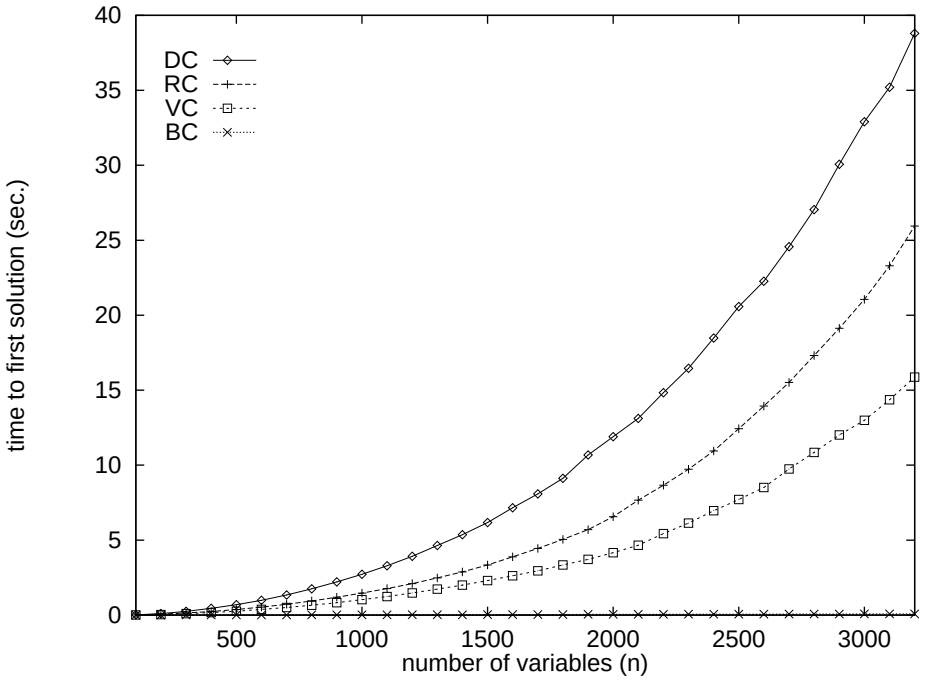

Figure 4.2: Time to find the first solution to the pathological problem

The Pathological Problem The pathological problem was designed by Puget as an adversary for a propagator enforcing bounds consistency. Consider $2n + 1$ variables x_i for $0 \leq i \leq 2n$ such that $\text{dom}(x_i) = [i - n, 0]$ for $0 \leq i \leq n$ and $\text{dom}(x_i) = [0, i - n]$ for $n < i \leq 2n$. There is a unique ALL-DIFFERENT constraint over all variables. This completes the problem. Observe now that the ALL-DIFFERENT constraint requires most of the computation time. We solve the problem using a lexicographic ordering of the variables. Figure 4.2 shows how bounds consistency is faster than other forms of consistencies. This result is not too surprising since the problem is designed specifically to the advantage of bounds consistency propagators. Therefore we emphasize the fact that BC was twice as fast as MT and about 5 times faster than Puget's algorithm which makes BC the fastest propagator for this problem.

m	VC	RC	DC	MT	BC
8	0.9	0.5	0.6	0.6	0.3
9	9.0	3.8	4.6	4.4	2.3
10	87.3	31.7	39.5	36.5	18.9
11	1773.6	688.1	871.2	841.4	437.6

Table 4.1: Time (sec.). to find an optimal solution of the Golomb ruler problem

The Golomb Ruler Problem A Golomb ruler (see problem 6 in CSPLib [25]) is a ruler with n marks where the $\frac{n(n-1)}{2}$ possible distances between two marks are all distinct. Smith et al. [77] model the problem of finding a Golomb ruler as n variables x_i such that $\text{dom}(x_0) = [1, 1]$ and $\text{dom}(x_i) = [1, 2^n]$ for $1 \leq i < n$ plus $\frac{n(n+1)}{2}$ difference variables d_i^j for $0 \leq i < j < n$ such that $\text{dom}(d_i^j) = [1, 2^n]$. The difference variables are subject to $d_i^j = x_j - x_i$ for $0 \leq i < j < n$ and we have the ordering constraints $x_{i-1} < x_i$ for $0 < i < n$. We finally apply an ALL-DIFFERENT constraint over all difference variables d_i^j. We use a lexicographic variable ordering to reproduce the same search Puget did. Using RC as a calibration point, our propagator BC turns out to be 1.5 times faster than Puget's propagator. Table 4.1 compares all other propagators together and show that BC is approximately 1.6 times faster than MT.

Instruction scheduling problems Instruction scheduling is used to optimize the object code produced by a compiler. A scheduling problem with n instructions consists of n variables $\{x_i\}_{i=1...n}$ whose domain represent the arrival time of the instruction in the execution pipeline. Latency constraints of the form $x_i \geq x_j + d$ where d is a small integer ensure that instruction j is completed before instruction i is processed. Finally, since only one instruction at a time can be queued, we apply an ALL-DIFFERENT constraint over all execution times x_i for $1 \leq i \leq n$. Redundant constraints called "distance constraints" are also added as explained in [82]. We used a minimum-domain-size ordering to solve 15 hard scheduling problems selected from the SPEC95 floating point, the SPEC2002 floating point and the MediaBench [47] benchmarks. Table 4.2 shows that our BC propagator

n	VC	RC	DC	MT	BC
69		0.02	0.04	0.04	0.02
70		0.02	0.05	0.04	0.02
111	0.08	0.07	0.12	0.11	0.07
211					
214	40.64	0.67	1.20	0.94	0.46
216		0.31	1.08	0.72	0.38
220		0.29	0.93	0.66	0.32
377		0.84	3.94	2.41	0.79
381	0.28	0.50	3.18	1.15	0.39
394		2.66	7.15	2.66	1.65
556					
690		1.91	26.88	3.95	1.61
691		14.47	40.50	6.30	3.14
856		7.72	17.09	10.86	5.48
1006		10.35	87.23	15.90	5.95

Table 4.2: Time (sec.) to find the optimal solution of an instruction scheduling problem. Problems that were not solved in 10 minutes are represented by blank entries.

outperforms all other propagators.

Random Problems We next consider some random problems to measure the scalability of the propagators. We consider problems of n variables whose domains $dom(x_i) = [a_i, b_i]$ are uniformly chosen in $[1, n]$. We consider a unique ALL-DIFFERENT constraint over all variables and solve the problem in lexicographic order. Since the problem has only one constraint, most of the computational time is spent in the ALL-DIFFERENT propagator. Figure 4.3 clearly shows the quadratic complexity of RC and DC. Both BC and MT are almost linear although MT (not shown) is 2.5 to 3 times slower than BC. The propagator VC was unable to solve even the smallest problems in less than 10 minutes.

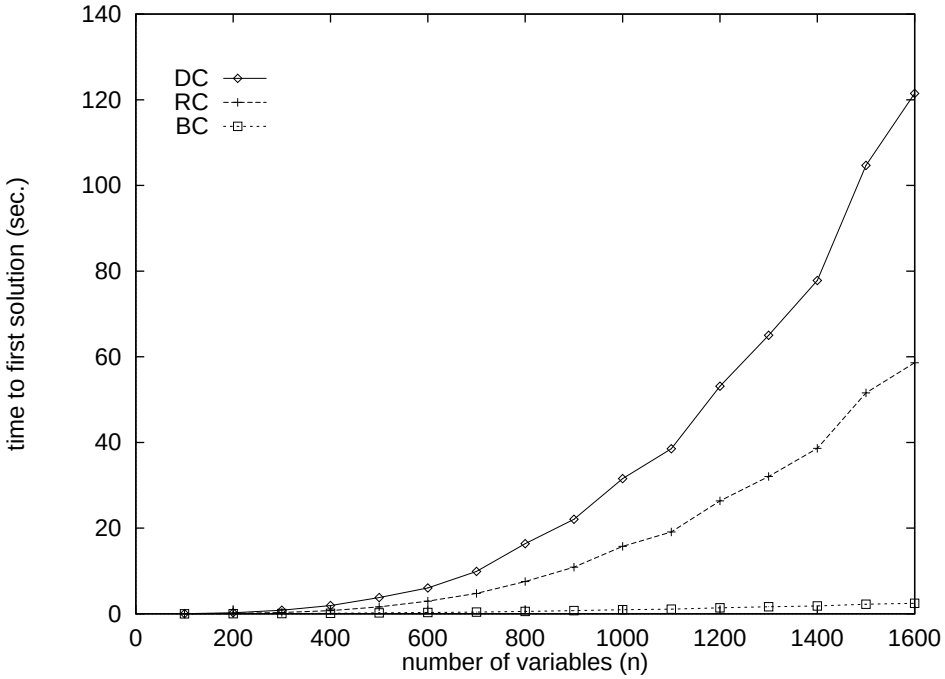

Figure 4.3: Time (sec.) required to find the first solution or to detect an inconsistency on random problems. Each entry is averaged over 100 problems.

Schulte and Stuckey [76] studied problems where bounds and domain consistency lead to the same search space. The Golomb problem is one of these problems where bounds consistency clearly offers a better performance than domain consistency. With the purpose to investigate the weaknesses of our propagator, we study problems where bounds and domain consistency do not lead to the same search space.

Random problems with holes Consider the random problems studied previously where variable domains are of the form $[a_i, b_i]$. We create holes in the domains by randomly removing each value in $[a_i + 1, b_i - 1]$ with a probability p. These problems are trivial for domain consistency but are not so for bounds and range consistency. We therefore use Puget's strategy that consists of running VC followed by BC to form the BC+ propagator. We similarly obtain MT+ by running VC followed by MT. Using the minimum domain size heuristic, we recorded the percentage of problems that were solved within a period of time of 5 seconds (see Figure 4.4). As expected, we observed that the more values were removed from the domains, the more inefficient was bounds and range consistency to solve a problem. When the value p passes 90%, few values remain in the domains and problems become easier to solve.

The n-Queens problem The n-Queens problem consists of placing on an $n \times n$ chessboard n queens such that no queen is attacked by another queen. Following the example provided by ILog [3], we model the problem with n variables q_i for $1 \leq i \leq n$. Queen number i is positioned on column i and row q_i. We have the three following constraints: ALL-DIFFERENT$(q_1, \ldots, q_n)$, ALL-DIFFERENT$(q_1 - 1, \ldots, q_n - n)$, and ALL-DIFFERENT$(q_1 + 1, \ldots, q_n + n)$. The first constraint states that no two queens are on the same row. The two other constraints model the fact that queens should not be on the same diagonal. Indeed, if two queens q_i and q_j were on the same diagonal, we would have $\frac{q_i - q_j}{i - j} = \pm 1$ which means that either $q_i - i = q_j - j$ or $q_i + i = q_j + j$ would be true. We branch on the variable with the smallest domain and break ties on the variable with the lower minimum value in its domain. On this problem, both BC and MT were unable to solve the problem

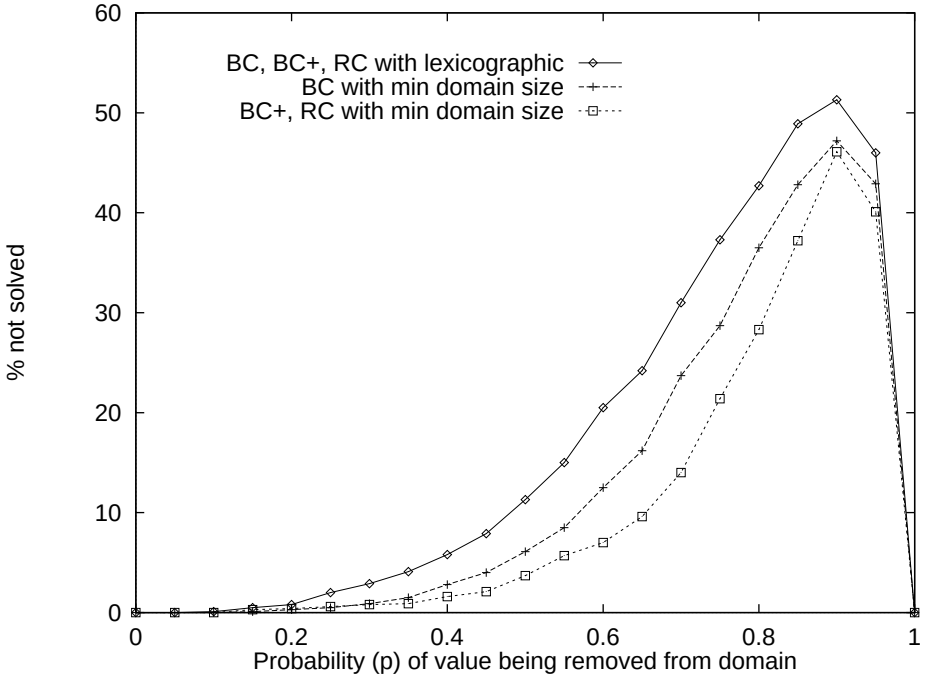

Figure 4.4: Percentage of random instances with holes not solved within 5 seconds for problems with 100 variables. Each data point is the average of 100 problems. The cutoff (5 seconds) was chosen to be the value that was at least two orders of magnitude slower than domain consistency, the fastest propagator on these problems.

within a reasonable time bound. The key to solve this problem is to remove values from the domains other than variable bounds. Even when BC+ and MT+ were able to solve the problem, they were not competitive with VC. Table 4.3 reports the time to find the first and all solutions to the n-queen problem.

	n	VC	RC	DC	MT^+	BC^+
all	8	0.0	0.0	0.1	0.2	0.1
	9	0.2	0.2	0.3	0.6	0.3
	10	0.6	0.8	1.0	2.3	1.0
	11	2.7	3.5	4.4	10.4	4.6
	12	13.1	17.0	21.6	52.3	22.6
first	100	0.1	0.1	0.4	0.2	0.1
	200	0.3	0.5	2.4	1.0	0.5
	400	1.1	2.9	17.5	4.3	1.9
	800	5.7	20.3	160.0	22.0	10.1
	1600	32.5	164.4	1264.2	111.8	56.2

Table 4.3: Time (sec.) to find all or first solutions for n-queens problems.

The Quasigroup Problem The quasigroup problem consists of filling the cells of an $n \times n$ table with numbers from 1 to n such that each number appears on each row and each column. In addition to this requirement, certain permutation constraints are imposed as described in Problem 3 of CSPLib [25]. We model the problem with one variable per cell with an ALL-DIFFERENT constraint on each row and each column. We used the minimum domain heuristic for order of instantiation. Table 4.4 shows the time in seconds for each problem we solved.

In all our experiments, RC was never the best choice. In problems where values other than bounds in domains were important, DC and VC outperformed RC. In other problems where pruning the bounds was more important, bounds consistency algorithms clearly were the best choice.

axiom	order	VC	RC	DC	MT+	BC+
3	8	0.2	2.9	2.7	0.2	0.2
	9	773.8	630.5	557.0	667.3	641.6
4	8	9.0	7.5	6.9	8.1	7.7
	9	1.6	1.3	0.8	1.4	1.4
5	8	0.2	0.1	0.2	0.1	0.1
	9	1.1	0.9	0.9	0.9	0.9
	10	14.4	12.0	6.3	12.3	12.1
	11	67.1	21.1	9.7	21.5	21.2
	12	1077.2	708.9	231.6	721.9	712.9
6	8	0.0	0.0	0.0	0.0	0.0
	9	0.2	0.2	0.2	0.2	0.2
	10	2.0	1.9	1.9	2.1	2.0
	11	32.4	31.7	32.1	33.6	32.0
	12	842.1	815.8	820.2	864.6	829.5
	13	10.0	10.0	10.3	10.8	10.3
7	8	0.9	0.8	0.9	1.0	0.9
	9	2.1	2.1	2.3	2.4	2.2
	10	499.8	483.8	506.0	529.9	495.2

Table 4.4: Time in seconds to find the first solution of a quasigroup problem. We solved problems of order 8 to 13 for each axiom.

4.3 Range Consistency for the ALL-DIFFERENT Constraint

In Section 3.4, we showed how Leconte designed an $\Theta(n^2)$ propagator for range consistency of the ALL-DIFFERENT constraint and proved its optimality. In this section, we present a new algorithm whose complexity is linear plus a time proportional to the number of values removed from the variable domains. Since Leconte proved there can be up to n^2 values to remove, our algorithm has the same worst case complexity as Leconte's algorithm, i.e. $\Omega(n^2)$. In practice however we observed

that generally not that many values need to be removed during the propagation of the ALL-DIFFERENT constraint. In fact, often, the propagator does not prune any variable domains. These observations are promising for propagators whose complexity depends on the number of values pruned. It is certainly efficient to remove $O(n^2)$ values from the domains in $O(n^2)$ instructions but rather inefficient to detect if the problem is already consistent using that many steps. We therefore suggest a propagator that detects in linear time if the ALL-DIFFERENT constraint is satisfiable and then spends a constant amount of time per value removed from a domain. Furthermore, under amortized complexity, our new propagator enforces range consistency in linear time instead of the worst case of $O(n^2)$.

The rest of the section is as follows. We refine concepts related to Hall's theorem and present their properties. Based on these concepts, we design a propagator for the range consistency of the ALL-DIFFERENT constraint. We study the asymptotic behavior of our propagator and finally present some experiments.

4.3.1 Basic Hall Intervals

We introduce a subclass of Hall intervals that we call basic Hall intervals.

Definition 4.1 (Basic Hall Interval) A *basic Hall interval* is a Hall interval that cannot be expressed as the union of two Hall intervals.

The basic Hall intervals have several interesting properties in a bounds consistent problem. The first property concerns how basic Hall intervals overlap with each other.

Lemma 4.2 *In a bounds consistent problem, two basic Hall intervals can only intersect if one Hall interval is a subset of the other Hall interval.*

Proof Let H_1 and H_2 be two basic Hall intervals such that H_1 and H_2 overlap on B i.e. $B = H_1 \cap H_2$. By way of contradiction, suppose that $H_1 \not\subseteq H_2$ and $H_2 \not\subseteq H_1$.

Since the problem is bounds consistent, a variable domain that has a bound in a Hall interval must have the other bound in the same Hall Interval. Let $A = H_1 - B$

and $C = H_2 - B$. A variable domain has both bounds in A, B, or C or it has no bounds in H_1 nor H_2. Since the sets A, B and C cannot contain more variable domains than their cardinality and H_1 and H_2 have to contain as many variables as values, we conclude that A, B and C are Hall intervals. This contradicts our hypothesis that H_1 and H_2 are basic Hall intervals since they can be expressed by the union of two Hall intervals ($H_1 = A \cup B$ and $H_2 = B \cup C$). Therefore two basic Hall intervals cannot overlap if one is not a subset of the other. □

We prove by way of an example that a basic Hall interval can be a subset of another basic Hall interval. Consider the following variable domains that are bounds consistent to the ALL-DIFFERENT constraint: $\text{dom}(x_1) = \text{dom}(x_2) = [1, 3]$ and $\text{dom}(x_3) = [2, 2]$. The two Hall intervals $H_1 = [1, 3]$ and $H_2 = [2, 2]$ cannot be expressed by the union of two other Hall intervals. They are therefore basic Hall intervals.

The second property relates to bounds of Hall intervals.

Lemma 4.3 *In a bounds consistent problem, two different basic Hall intervals cannot share a common bound.*

Proof Let H_1 and H_2 be two different basic Hall intervals. According to Lemma 4.2, a basic Hall interval cannot have for lower bound the upper bound of a second basic Hall interval. Therefore we consider the case where two basic Hall intervals share the same lower bound or the same upper bound.

Suppose that H_1 and H_2 are two different basic Hall intervals sharing the same lower bound i.e. $H_1 = [a, b]$ and $H_2 = [a, c]$ such that $b < c$. Let $H_3 = H_2 - H_1 = (b, c]$. The number of variable domains contained in H_2 can be decomposed as follows.

$$C(H_2) \quad = \quad C(H_1) + C(H_3) + |\{x \mid \min(\text{dom}(x)) \in H_1 \wedge \max(\text{dom}(x)) \in H_3\}|$$

Since the problem is bounds consistent and H_1 is a Hall interval, we know that no variables have one bound in H_1 and the other bound in H_3. We now show that

H_3 is also a Hall interval. Observe that

$$\begin{aligned} C(H_2) &= C(H_1) + C(H_3) + \{x \in X | \min(x) \in H_1 \wedge \max(x) \in H_3\} \\ C(H_2) &= C(H_1) + C(H_3) \\ C(H_3) &= C(H_2) - C(H_1) \\ C(H_3) &= (c - a + 1) - (b - a + 1) \\ C(H_3) &= c - b \end{aligned}$$

Since H_2 can be expressed by the union of H_1 and H_3 and both of them are Hall intervals, we conclude that H_2 is not a basic Hall interval which is in contradiction with our hypothesis. □

Since there must be at least one variable domain contained in a basic Hall interval and every basic Hall interval is uniquely defined by one of its bounds, there are at most n basic Hall intervals in a problem that counts n variables.

4.3.2 A New Algorithm for Range Consistency

Using the properties of basic Hall intervals, we suggest a new algorithm that makes a problem range consistent and has a time complexity of $O(t + n + u)$ where t is the time complexity for sorting n variables and u is the number of values removed from the domains. This algorithm proceeds in four steps:

1. Make the problem bounds consistent.
2. Sort the variables by increasing lower bounds.
3. Find the basic Hall intervals.
4. Prune the variable domains.

Steps 1 and 2 are not a concern since they can be performed in linear time using the bounds consistency algorithm described in Section 4.2 and an implementation of any sorting algorithm. We therefore focus our attention on Steps 3 and 4.

To find the basic Hall intervals, we maintain a stack S of tuples $\langle interval, count \rangle$ where *interval* is an interval of values and *count* indicates how many processed variable domains are contained in *interval*. If the number of elements in *interval* equals *count*, then *interval* is a basic Hall interval. We maintain an invariant on the stack such that an interval at a higher level is a subset of the interval at a lower level. Moreover, intervals on the stack have distinct lower and upper bounds. We initialize the stack by pushing the infinite interval represented by the tuple $\langle [-\infty, \infty], 0 \rangle$. Here, the *count* variable is set to 0 since no variables have been processed so far. We also add to the problem a dummy variable domain that contains values greater than any other values contained in other variable domains. This dummy variable domain should be large enough (i.e. at least 2) to never create a Hall interval.

We process each variable domain in ascending order of lower bound. While the interval on top of the stack does not intersect with the current variable domain, the algorithm pops the tuple out of the stack. When popping an element out of the stack, we add to the *count* variable of the tuple on top of the stack the *count* variable of the removed tuple. This operation maintains the number of variable domains contained in the interval on top of the stack. Each time the *count* variable is updated, we check if the interval is a Hall interval, i.e. if the *count* variable is equal to the number of values in the interval. The algorithm prints the Hall intervals as it discovers them.

After popping out the tuples, we are now ready to process the current variable. We push the current variable domain on the stack and consider it as a potential basic Hall interval. Let $\langle I_1, c_1 \rangle$ and $\langle I_2, c_2 \rangle$ be two elements on top of the stack. If these two elements on the stack do not satisfy the properties of basic Hall intervals stated in Lemma 4.2 and Lemma 4.3, we pop them out and push the tuple $\langle I_1 \cup I_2, c_1 + c_2 \rangle$ which represents the new candidate for a basic Hall interval. Notice that we might have to pop tuples and repeatedly merge them together to restore the stack invariant. Algorithm 4 uses this technique to print all basic Hall intervals.

Algorithm 4: Prints the basic Hall intervals in a bounds consistent problem

```
push(S,⟨[−∞, ∞], 0⟩)
dom(d) ← [max(D) + 1, max(D) + 3]
X ← X ∪ {d}
for x ∈ X in non decreasing order of min(dom(x)) do
    while max(int(top(S))) < min(dom(x)) do
        if |interval(top(S))| = count(top(S)) then
            print interval(top(S))
        n ← count(pop(S))
        count(top(S)) ← count(top(S)) + n
    T ← ⟨dom(x), 1⟩
    while min(interval(T)) ≤ min(interval(top(S))) or
    max(interval(top(S))) ≤ max(interval(T)) do
        ⟨I, c⟩ ← pop(S)
        T ← ⟨I ∪ int(T), c + count(T)⟩
        if S is empty then break
    push(S, T)
```

Algorithm 4 runs in $O(n)$ since each variable is pushed, popped, merged, and printed at most once. One can prune the variable domains to achieve range consistency by simply removing each basic Hall interval H from the variable domains that are not fully contained in H. Let n be the number of variables and H the set of values that belong to a Hall interval. Pruning the variable domains requires $O(n\,|H|)$ time. Putting it all together, the total time complexity of Algorithm 4 is $O(t + n + n\,|H|)$ where t is the complexity to sort n variables by lower and upper bounds.

The performance can be further improved by making the propagator incremental. An incremental propagator is a propagator that reuses computations performed during previous propagation steps and uses them during the current propagation. Our propagator can be modified to reuse basic Hall intervals discovered in previous propagation steps. If no backtrack occurred, we know that variable domains could only have been reduced since the last propagation step. This reduction of domain can create new Hall intervals, yet previously discovered Hall intervals never disappear.

Lemma 4.4 *Let H be a Hall interval in a problem containing an* All-Different *constraint. If some values are removed from the variable domains of the original problem, H remains a Hall interval in the modified problem.*

Proof By definition, a Hall interval H is an interval that contains as many variable domains as its cardinality. If variable domains are shrunk, they are still contained in H and therefore the Hall interval remains a Hall interval. □

While Hall intervals remain Hall intervals under the reduction of domains, this is not the case for basic Hall intervals. A basic Hall interval H can be segmented into several basic Hall intervals $H_1, \ldots, H_n$ such that $H = H_1 \cup \ldots \cup H_n$. Therefore, H is no longer a basic Hall interval.

We can make our propagator incremental by only considering new basic Hall intervals that are not part of a fragmentation of a previously discovered basic Hall interval. These Hall intervals are the only ones that can create new holes in variable domains since other Hall intervals have already been processed during previous

propagations. Since Algorithm 4 lists basic Hall intervals in increasing order of upper bounds, a linear scan can quickly identify new and old basic Hall intervals. Let H be the set of values belonging to a new basic Hall intervals that is not part of a segmentation of a previously discovered Hall interval. The running time complexity of the incremental algorithm is therefore $O(t+n+n|H|)$. The improvement in the running time complexity is better perceived in the amortized analysis.

Lemma 4.5 *The amortized complexity of the incremental propagator over a branch in the search tree is* $O(t+n)$.

Proof When run for the i^{th} time, the incremental algorithm has complexity $O(t+n+n|H_i|)$. For a problem with n variables, the maximum number of values appearing in a basic Hall interval is n. Suppose the propagator is called m times from the root to a leaf of the search tree. We have $\sum_{i=1}^{m} |H_i| \leq n$. The total cost is therefore $O(m\,(t+n)+n^2)$. It follows that the amortized cost over a sequence of m calls for $m \in \Omega(n)$ is $O(t+n)$. □

Lemma 4.5 proves that in the amortized case, bounds consistency and range consistency have the same running time complexity.

4.3.3 Experiments

We implemented the propagator described in Section 4.3.2 which we label LRC. We compared it with already existing propagators on the Golomb ruler problem (see problem 6 in CSPLib [25]). We essentially used the same model and the same notation to identify propagators as in Section 4.2.2. All reported times are averaged over 10 runs.

	Time (s)						# Fails	
n	VC	RC	DC	BC	BC+	LRC	VC	others
7	0.0	0.0	0.0	0.0	0.0	0.0	355	110
8	0.5	0.2	0.3	0.2	0.3	0.3	2735	697
9	4.8	2.0	2.5	1.2	2.1	2.2	19445	3740
10	46.7	16.8	20.6	10.3	17.6	18.4	140746	23464
11	951.1	363.0	450.5	237.5	375.9	399.1	2230979	374888

Table 4.5: Left: time (s) to find the first solution to the Golomb ruler problem with n marks. Right: number of fails during the search. RC, DC, BC, BC+, and LRC all have the same number of fails.

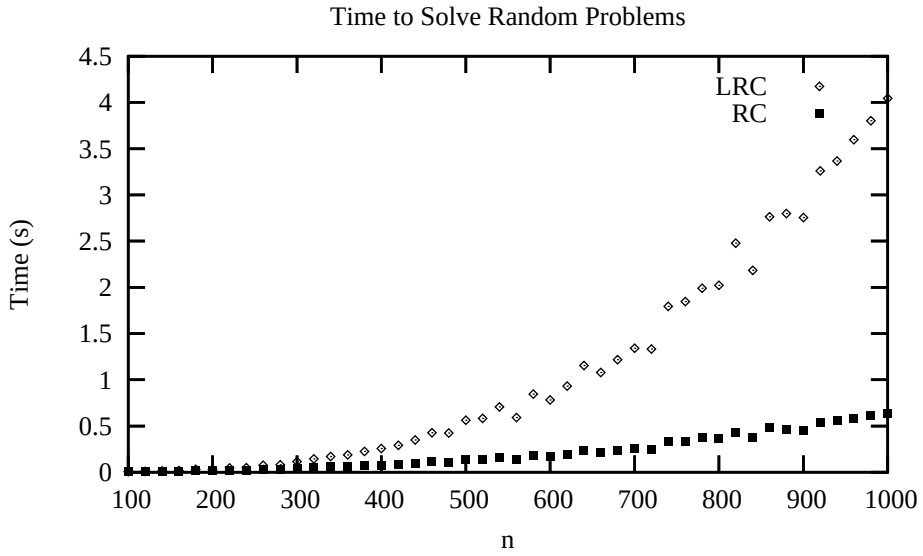

Figure 4.5: Time in seconds to solve random problem instances using the incremental propagator LRC and Leconte's propagator for the range consistency of the ALL-DIFFERENT constraint.

We see on Table 4.5 that our propagator (LRC) is about 10% slower than the propagator provided by ILog (RC) for the same consistency. This is due to the hidden constant in the running time complexity. The next experiment shows that our propagator is faster than RC on larger datasets.

We consider the same random problems studied in Section 4.2.2 to measure the scalability of the propagators. Recall that in these problems, there are n variables whose domains $dom(x_i) = [a_i, b_i]$ are uniformly chosen in $[1, n]$. We consider a unique ALL-DIFFERENT constraint over all variables and solve the problem in lexicographic order. Since the problem has only one constraint, most of the computational time is spent in the ALL-DIFFERENT propagator. We used the same instances of random problems for testing RC and LRC. The running time for a specific n is averaged over 100 instances of random problems. Figure 4.5 clearly shows that the running complexity of LRC is inferior to the complexity of RC.

4.4 The All-Different Constraint on Non Integer Variables

So far, we have studied the All-Different constraint on integer variables. Under certain circumstances, we might want to propagate this constraint to variables with more structure. For instance, we might want to propagate the All-Different constraint over set variables [27, 26, 62, 57], multiset variables [87], or ordered tuple variables. Structured variables offer many advantages compared to integer variables. First, structured variables reduce the memory space required to represent variable domains. With set variables for instance, instead of listing a large number of possible sets to be assigned to a variable, one can simply list the potential elements and the required elements of the set. Second, structured variables improve the efficiency of propagators by making important information more accessible to the algorithms. For instance, a propagator might want to know what are the definite elements in a set without having to iterate through all possible set assignments. Finally, structured variables offer the usual benefits from data abstraction like ease of debugging and code maintenance.

The All-Different constraint on non integer variables occurs on a variety of problems. Consider for instance the round robin sports scheduling problem (problem 026 in CSPLib [25]). In this problem, we must schedule a tournament such that all sport teams meet each other. One could see a match as a set of size two containing the two teams playing together. An All-Different constraint posted on all sets ensure that every match is unique. We show how to propagate the All-Different constraint on set, multiset, and tuple variables.

4.4.1 Beyond Integer Variables

A propagator designed for integer variables can be applied to any type of variable whose domain can be effectively enumerated. For instance, let the following variables be sets whose domains are expressed by a set of required values and a set of allowed values.

$$\{\} \subseteq S_1, S_2, S_3, S_4 \subseteq \{1, 2\} \text{ and } \{\} \subseteq S_5, S_6 \subseteq \{2, 3\}$$

Variable domains can be expanded as follows:

$$S_1, S_2, S_3, S_4 \in \{\{\}, \{1\}, \{2\}, \{1, 2\}\} \text{ and } S_5, S_6 \in \{\{\}, \{2\}, \{3\}, \{2, 3\}\}$$

And then by enforcing domain consistency on the ALL-DIFFERENT constraint, we obtain

$$S_1, S_2, S_3, S_4 \in \{\{\}, \{1\}, \{2\}, \{1, 2\}\} \text{ and } S_5, S_6 \in \{\{3\}, \{2, 3\}\}$$

We can now convert the domains back to their initial representation.

$$\{\} \subseteq S_1, S_2, S_3, S_4 \subseteq \{1, 2\} \text{ and } \{3\} \subseteq S_5, S_6 \subseteq \{2, 3\}$$

This technique while always correct can lead to intractable problems since variable domains might have exponential size. For instance, the domain of $\{\} \subseteq S_i \subseteq \{1, \ldots, n\}$ contains 2^n elements.

As we did for our propagator for the bounds consistency of the ALL-DIFFERENT constraint (see Section 4.2), we use Hall's marriage theorem (see Theorem 2.4 in Section 2.2.3) to enforce domain consistency on the ALL-DIFFERENT constraint. The definition of a Hall interval can be generalized to sets as follows.

Definition 4.2 (Hall Set) A *Hall set* is a set H such that there are $|H|$ variables whose domains are contained in H.

In our example, the set $H = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$ is a Hall set since its cardinality is four and exactly four variables (S_1, S_2, S_3, S_4) have their domains contained in H.

The following important lemma allows us to ignore variables whose domains are too large to be enumerated and only focus on those with "small" domains.

Lemma 4.6 *Let n be the number of variables and let F be a set of variables whose domains are not contained in any Hall set. Let $x_i \notin F$ be a variable whose domain contains more than $n - |F|$ values. Then $dom(x_i)$ is not contained in any Hall set.*

Proof The largest Hall set can contain the domain of $n-|F|$ variables and therefore has at most $n-|F|$ values. If $|\mathrm{dom}(x_i)| > n-|F|$, then $\mathrm{dom}(x_i)$ cannot be contained in any Hall set. □

Using Lemma 4.6, we can iterate through the variables and append to a set F those whose domain cannot be contained in a Hall set. A propagator for the All-Different constraint can prune the domains not in F and find all Hall sets. Values in Hall sets can then be removed from the variable domains in F. This technique ensures that domains larger than n do not slow down the propagation. Algorithm 5 exhibits the process for a set of (possibly non-integer) variables X.

Algorithm 5: All-Different propagator for variables with large domains

$F \leftarrow \emptyset$
for $x_i \in X$ **do**
1 **if** $|dom(x_i)| > n - |F|$ **then** $F \leftarrow F \cup \{x_i\}$
2 Expand domains of variables in $X - F$.
Find values H belonging to a Hall set and propagate the All-Different constraint on variables $X - F$.
for $x_i \in F$ **do**
$\mathrm{dom}(x_i) \leftarrow \mathrm{dom}(x_i) - H$;
3 Collapse domains of variables in $X - F$.

To apply our new techniques, three conditions must be satisfied by the representation of the variables:

1. Computing the size of the domain must be tractable (Line 1).
2. Domains must be efficiently enumerable (Line 2).
3. Domains must be efficiently computable from an enumeration of values (Line 3)

The next sections describe how different representations of domains for set, multiset and tuple variables can meet these three conditions.

4.4.2 The ALL-DIFFERENT Constraint on Sets

Several representations of domains have been suggested for set variables. We show how their cardinality can be computed and their domain enumerated efficiently. One of the most common representations for a set variable S uses the set of required elements $lb(S)$ and the set of allowed elements $ub(S)$. Any set S satisfying $lb(S) \subseteq S \subseteq ub(S)$ belongs to the domain [27, 62]. To simplify notation, we denote $lb(S)$ and $ub(S)$ with lb and ub each time the clarity is not compromised. The number of sets in the domain is given by $2^{|ub-lb|}$. We can enumerate all these sets simply by enumerating all subsets of $ub - lb$ and adding them to the elements from lb. A set can be represented as a binary vector where each element is associated to a bit. A bit equals 1 if its corresponding element is in the set and equals 0 if its corresponding element is not in the set. Enumerating all subsets of $ub-lb$ is reduced to the problem of enumerating all binary vectors between 0 and $2^{|ub-lb|}$ exclusively which can be done in $O(2^{|ub-lb|})$ steps, i.e. $O(|\text{dom}(S_i)|)$ steps [42].

Representing the domains with a lower bound and an upper bound might lead in some cases to a poor approximation. For instance, to represent the domain $\text{dom}(S) = \{\{1, 2\}, \{3\}\}$ using a lower and an upper bound, one has to set the lower bound to $lb = \{\}$ and the upper bound to $ub = \{1, 2, 3\}$. With this representation, sets such as $\{\}$ and $\{1, 2, 3\}$ are contained in the variable domain. In order to exclude from the domain these undesired sets, one can also add a cardinality variable [3]. The domain of a set variable is therefore expressed by $\text{dom}(S_i) = \{S \mid lb \subseteq S \subseteq ub, |S| \in \text{dom}(C)\}$ where C is an integer variable. We assume that C is consistent with lb and ub, i.e. $\min(C) \geq |lb|$ and $\max(C) \leq |ub|$. The size of the domain is given by Equation 4.9 where $\binom{a}{b}$ is the binomial coefficient.

$$|\text{dom}(S_i)| = \sum_{j \in C} \binom{|ub - lb|}{j - |lb|} \tag{4.9}$$

Algorithm 6: Enumerate the $\binom{n}{t}$ combinations of t elements between 0 and $n-1$. (Source: Algorithm T, Knuth [41] p.5)

```
c_j ← j − 1, ∀j 1 ≤ j ≤ t
c_{t+1} ← n
c_{t+2} ← 0
repeat
    visit c_t, c_{t−1}, ..., c_1
    j ← 1
    while c_j + 1 = c_{j+1} do
        c_j ← j − 1
        j ← j + 1
    c_j ← c_j + 1
until j > t
```

The binomial coefficients can be efficiently computed (see for example Chapter 6.1 of [61]). The identity $\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$ can be particularly useful to compute the summation when the domain of C is an interval. The number of steps required to compute $|\text{dom}(S_i)|$ is bounded by $O(|\text{dom}(C)|)$.

Algorithm 6 enumerates all combinations of t elements chosen from elements 0 to $n-1$. Each element i in a combination is mapped to the i^{th} element in $ub - lb$. By enumerating all t-combinations for $t \in \text{dom}(C)$ to which we add the required elements lb, we enumerate all sets in $|\text{dom}(S_i)|$. This algorithm has a time complexity of $O(t + \binom{n}{t})$. Since we call it for each $t \in \text{dom}(C)$, the total time complexity simplifies to $O(\max\{|ub - lb|, |\text{dom}(S_i)|\})$.

Sadler and Gervet [75] suggest adding a lexicographic ordering constraint to the domain description. This gives more expressiveness to the domain representation and can eliminate sets that do not belong to the domain. Given two sets S_1 and S_2, we say that $S_1 < S_2$ holds if S_1 comes before S_2 in the given lexicographical order. The new domain representation now involves two lexicographic bounds l and u.

$$\text{dom}(S_i) \quad = \quad \{S \mid lb \subseteq S \subseteq ub, |S| = C, l \leq S \leq u\} \tag{4.10}$$

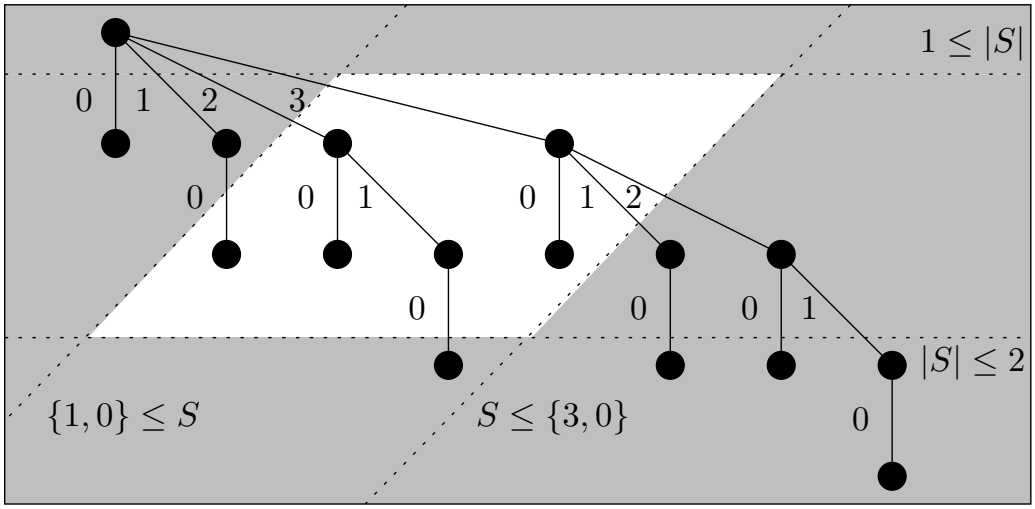

Figure 4.6: Binomial tree representing the domain $\emptyset \subseteq S_i \subseteq \{0,1,2,3\}$, $1 \leq |S_i| \leq 2$, and $\{1,0\} \leq S_i \leq \{3,0\}$.

Knuth [41] represents all subsets of a set using a binomial tree like the one shown in Figure 4.6. The empty set $\emptyset$ is the root of the tree to which we can add elements by branching to a child. One can list all sets in lexicographical order by visiting the tree from left to right with a depth-first-search (DFS). We clearly see that the lexicographic constraints are orthogonal to the cardinality constraints.

Based on the binomial tree, we compute, level by level, the number of sets that belong to the domain. Notice that sets at level k have cardinality k. A set in the variable domain can be encoded with a binary vector of size $|ub - lb|$ where each bit is associated to a potential element in $ub - lb$. A bit set to one indicates that the element belongs to the set while a bit set to zero means that the element does not belong to the set. The number of sets of cardinality k in the domain is equal to the number of binary vectors with k bits set to one and that lexicographically lie between l and u. Let $[u_m, \ldots, u_1]$ be the binary representation of the lexicographic upper bound u. Assuming $\binom{b}{a} = 0$ for all negative values of a, let $C([u_m, \ldots, u_1], k)$ be defined as follows.

$$C([s_m,\ldots,s_1],k) = \sum_{i=1}^{m} s_i \binom{i-1}{k-\sum_{j=i+1}^{m} s_j} + \delta(\vec{s},k) \quad (4.11)$$

$$\delta([s_m,\ldots,s_1],k) = \begin{cases} 1 & \text{if } \sum_{i=1}^{m} s_i = k \text{ and } s_0 = 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.12)$$

Lemma 4.7 *The function $C([s_m,\ldots,s_1],k)$ computes the number of binary vectors that are lexicographically smaller than or equal to u and that have k bits set to one.*

Proof We prove correctness by induction on m. For $m = 1$, Equation 4.11 holds with both $k = 0$ and $k = 1$. Suppose the equation holds for m, we want to prove it also holds for $m + 1$. We claim

$$C([s_{m+1},\ldots,s_1],k) = s_{m+1}\binom{m}{k} + C([s_m,\ldots,s_1],k-s_{m+1}) \quad (4.13)$$

If $s_{m+1} = 0$, the lexicographic constraint is the same as if we only consider the m first bits. We therefore have $C([s_{m+1},\ldots,s_1],k) = C([s_m,\ldots,s_1],k)$ which corresponds to Equation 4.13 with $s_{m+1} = 0$. Alternatively, if $s_{m+1} = 1$, $C(s,k)$ returns $\binom{m}{k}$ which corresponds to the number of vectors with k bits set to 1 and the $(m+1)^{th}$ bit set to zero plus $C([s_m,\ldots,s_1],k-1)$ which corresponds to the number of vectors with k bits set to 1 including the $(m+1)^{th}$ bit. Recursion 4.13 is therefore correct. Successively applying the recursion consists of summing up the term $s_{m+1}\binom{m}{k}$ for every $s_{m+1} = 1$. This process results in Equation 4.11 which takes into account the base case. □

Let a and b be binary vectors respectively representing the lexicographical bounds l and u where bits associated to the required elements lb are omitted. We denote by $a-1$ the binary vector that precedes a in the lexicographic order. The size of the domain is given by the following equation:

$$|\mathrm{dom}(S_i)| = \sum_{k\in C} (C(b,k) - C(a-1,k))$$

The function C can be evaluated in $O(|ub - lb|)$ steps. Therefore, the size of the domain $\text{dom}(S_i)$ requires $O(|ub - lb||C|)$ steps to compute. Enumerating can also proceed level by level without taking into account the required elements lb since they belong to all sets in the domain. The first set on level k can be obtained from the lexicographic lower bound l. If $|l| \neq k$, we have to find the first set l' of cardinality k that is lexicographically greater than l. If $|l| < k$, we simply add to set l the $k - |l|$ smallest elements in $ub - lb - l$. Suppose $|l| > k$ and consider the binary representation of l. Let p be the k^{th} most significant bit set to 1 in l. We add one to bit p and duly propagate the carry. We set all bits before p to 0. This gives a bit vector l' representing a set with no more than k elements. If $|l'| < k$, we add the first $k - |l'|$ elements in $ub - lb - l'$ to l' and obtain the first set of cardinality k.

Once the first set at level k has been computed, subsequent sets can be obtained using Algorithm 6. Obtaining the first set of each level takes time $O(|\text{dom}(C)||ub - lb|)$ and cumulative calls to Algorithm 6 take time $O(\sum_{i \in \text{dom}(C)} i + |dom(S)|)$. Enumerating the domain therefore requires $O(|\text{dom}(C)||ub - lb| + |\text{dom}(S)|)$ steps.

4.4.3 The ALL-DIFFERENT Constraint on Tuples

A tuple t is an ordered sequence of n elements that allows multiple occurrences of a same element. Like sets, there are different ways to represent the domain of a tuple. The most common way is simply by associating an integer variable to each of the tuple elements. A tuple of size n is therefore represented by n integer variables $x_1, \ldots, x_n$.

To apply an ALL-DIFFERENT constraint to a set of tuples, a common solution is to create an integer variable t for each tuple. If each element x_i ranges from 0 to c_i exclusively, we add the following channeling constraint between tuple t and its elements.

$$t = ((((x_1 c_2 + x_2)c_3 + x_3)c_4 + x_4)\ldots)c_n + x_n = \sum_{i}^{n} \left(x_i \prod_{j=i+1}^{n} c_j \right)$$

This technique suffers from either inefficient or ineffective channeling between the variable t and the elements x_i. Most constraint libraries enforce bounds consistency on t. A modification to the domain of x_i does not affect t if the bounds of $\mathrm{dom}(x_i)$ remain unchanged. Conversely, even if all tuples encoded in $\mathrm{dom}(t)$ have $x_i \neq v$, value v will most often not be removed from $\mathrm{dom}(x_i)$. On the other hand, enforcing domain consistency typically requires $O(n^{|T|})$ steps where $|T|$ is the size of the tuple.

To address this issue, one can define a tuple variable whose domain is defined by the domain of its elements.

$$\mathrm{dom}(t) \quad = \quad \mathrm{dom}(x_1) \times \ldots \times \mathrm{dom}(x_n)$$

The size of such a domain is given by the following equation which can be computed in $O(n)$ steps provided that there is enough bits in a word to store the result.

$$|\mathrm{dom}(t)| \quad = \quad \prod_{i=1}^{n} |\mathrm{dom}(x_i)|$$

The domains of tuple variables can be enumerated using Algorithm 7. Assuming the domain of all element variables have the same size, Algorithm 7 runs in $O(|dom(t)|)$ which is optimal.

As Sadler and Gervet [75] did for sets, we can add lexicographical bounds to tuples in order to better express the values the domain contains. Let l and u be these lexicographical bounds.

$$\mathrm{dom}(t) \quad = \quad \{t \mid t[i] \in \mathrm{dom}(x_i), l \leq t \leq u\}$$

Let $idx(v, x)$ be the number of values smaller than v in the domain of the integer variable x. More formally, $idx(v, x) = |\{w \in \mathrm{dom}(x) \mid w < v\}|$. Assuming $idx(v, x)$

Algorithm 7: Enumerate tuples of size n in lexicographical order. (Source: Algorithm T, Knuth [42] p.2)

Initialize first tuple: $a_j \leftarrow \min(x_j)$, $\forall j\ 1 \leq j \leq n$
repeat
  visit $(a_1, a_2, \ldots, a_n)$
  $j \leftarrow n$
  while $j > 0$ and $a_j = \max(dom(x_j))$ **do**
    $a_j \leftarrow \min(x_j)$
    $j \leftarrow j - 1$
  $a_j \leftarrow \min(\{a \in \text{dom}(x_j) \mid a > a_j\})$
until $j = 0$

has a running time complexity of $O(\log(|\text{dom}(x)|))$, the size of the domain can be evaluated in $O(n + \log(|\text{dom}(t)|))$ steps using the following equation:

$$|\text{dom}(t)| \quad = \quad 1 + \sum_{i=1}^{n} \left((idx(u[i], x_i) - idx(l[i], x_i)) \prod_{j=i+1}^{n} |\text{dom}(x_i)| \right).$$

We enumerate the domain of tuple variables with lexicographical bounds similarly as tuple variables without lexicographical bounds. We simply initialize Algorithm 7 with tuple l and stop enumerating when tuple u is reached. This operation is performed in $O(|\text{dom}(t)|)$ steps using Algorithm 7.

4.4.4 The ALL-DIFFERENT Constraint on Multi-Sets

Unlike sets, multi-sets allow multiple occurrences of the same element. We use $occ(v, S)$ to denote the number of occurrences of element v in multi-set S. An element v belongs to a multi-set A if and only if its number of occurrences $occ(v, A)$ is greater than 0. We say that set A is contained in set B ($A \subseteq B$) if for all element v we have $occ(v, A) \leq occ(v, B)$. The domain representation of multi-sets is generally similar to the one for standard sets. We have a multi-set of essential elements lb and

a multi-set of allowed elements ub. Equation 4.14 gives the domain of a multi-set and Equation 4.15 shows how to compute its size in $O(|ub|)$ steps.

$$\text{dom}(S_i) = \{S \mid lb \subseteq S \subseteq ub\} \tag{4.14}$$

$$|\text{dom}(S_i)| = \prod_{v \in ub} (occ(v, ub) - occ(v, lb) + 1) \tag{4.15}$$

Multisets can be represented by a vector where each element represents the number of occurrences of an element in the multi-set. Of course, for the multi-set to be in the domain, this number of occurrences must lie between $occ(v, lb)$ and $occ(v, ub)$. Therefore a multi-set variable is equivalent to a tuple variable where the domain of each element is given by the interval $[occ(v, lb), occ(v, ub)]$. Enumerating the values in the domain is done as seen in Section 4.4.3. The same approach can be used to introduce lexicographical bounds to multi-sets.

4.4.5 Indexing Domain Values

Propagators for the ALL-DIFFERENT constraint, such as the one proposed by Régin [67] (see Section 3.3.1), need to store information about some values appearing in the variable domains. When values are integers, the simplest implementation is to create a table T in which information related to value v is stored in entry $T[v]$. Algorithm 5 ensures that the propagator is called over a maximum of n variables each having no more than n (possibly distinct) values in their domain. We therefore have a maximum of n^2 values to consider. When these n^2 values come from a significantly greater set of values, the table T becomes sparse. In some cases, it might not even be realistic to consider such a solution. To allow direct access memory when accessing the information of a value, we need to map the n^2 values to an index in the interval $[1, n^2]$.

For this, we build an indexing tree able to index sets, multi-sets, or tuples. Each node is associated to a sequence. The root of the tree is the empty sequence ($\emptyset$). We append an element to the current sequence by branching to a child of the current node. There are at most n^2 nodes corresponding to a value in a variable domain.

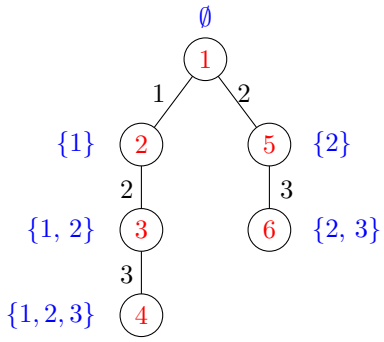

Figure 4.7: Indexing tree representing the following domains: $\emptyset \subseteq S_1 \subseteq \{1,2\}, \{1\} \subseteq S_2 \subseteq \{1,2\}, \{2\} \subseteq S_3 \subseteq \{1,2\}, \emptyset \subseteq S_4 \subseteq \{1\}, \{2\} \subseteq S_5 \subseteq \{1,2,3\}$.

These nodes are labeled with integers from 1 to n^2. Figure 4.7 shows the indexing tree based on the domain of 5 set variables.

This simple data structure allows to index and retrieve in $O(l)$ steps the number associated to a sequence of length l.

4.4.6 Experiments

To test the efficiency and effectiveness of these generalizations to the propagator for the ALL-DIFFERENT constraint, we ran a number of experiments on a well known problem from design theory. A *latin square* is an $n \times n$ table where cells can be colored with n different colors. We use integers between 1 and n to identify the n colors. A *græco-latin square* [45] is m latin squares $A_1, \ldots, A_m$ such that the tuples $\langle A_1[i,j], \ldots, A_m[i,j] \rangle$ are all distinct. For a survey about latin squares and græco-latin squares, the reader is referred to [45]. The following tables represent a græco-latin square for $n = 4$ and $m = 2$.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

3	4	1	2
1	2	3	4
2	1	4	3
4	3	2	1

We encode the problem using one tuple variable per cell. The problem has an All-Different constraint on each row and each column. We add a redundant 0/1-Cardinality-Matrix constraint on each value as suggested by Régin [72] to increase the pruning of the variable domains. We use two different encodings for tuples: one is the tuple encoding where each element is an integer variable, the other is the factored representation. We enforce bounds consistency on the channeling constraints between the cell variables and the factored tuple variables. As suggested in [72], our heuristic chooses the variable with the smallest domain and we break ties on the variable that has the most bounded variables on its row and column. We use the same implementation of the All-Different propagator for both tuple encodings.

Table 4.6 and Figure 4.8 clearly illustrates that when tuples get longer, our technique outperforms the factored representation of tuples. This is mainly due to space requirements since the factored representation of long tuples requires too much memory which eventually exceeds the amount of RAM available on the computer. When this happens, the operating system uses the hard drive to store the information which significantly slows down the process.

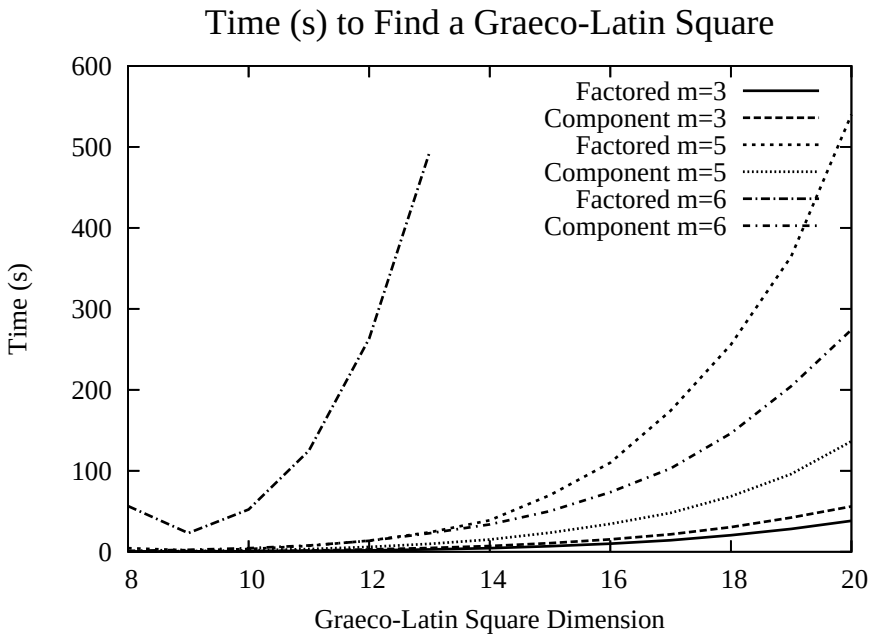

Figure 4.8: Time in seconds to solve a græco-latin square with m different square sizes. The data is extracted from Table 4.6. We see that for $m \geq 5$, the element encoding offers a better performance than the factored encoding.

Table 4.6: Time to solve a græco-latin square using factored and tuple variables

n \ m	3		4		5		6	
	factored	tuple	factored	tuple	factored	tuple	factored	tuple
8	0.48	0.23	0.57	0.35	4.51	0.40	56.48	1.08
9	0.33	0.49	0.31	0.85	1.77	0.94	23.09	2.39
10	0.58	0.91	0.56	1.57	3.44	1.78	52.30	4.36
11	1.05	1.62	1.04	2.97	7.33	3.23	124.95	7.69
12	1.76	2.80	1.79	5.59	13.70	6.04	263.28	13.61
13	2.86	4.69	2.85	9.00	23.96	9.74	493.04	22.80
14	4.37	7.03	4.17	14.34	38.95	15.19		33.79
15	6.88	10.62	6.56	22.18	69.89	23.63		50.23
16	10.11	15.41	9.54	32.52	110.08	34.55		73.60
17	14.21	21.48	13.82	45.35	174.18	47.89		102.98
18	20.41	30.55	19.13	64.87	255.76	68.46		146.21
19	28.28	42.12	25.01	91.45	364.58	95.99		204.45
20	38.31	56.10	34.35	122.30	540.06	136.43		274.29

Chapter 5

New Propagators for the Global Cardinality Constraint

In this chapter, we present novel propagators [66, 64, 65] for the global cardinality constraint (GCC). We recall the definition of the GCC.

Definition 5.1 (Global Cardinality Constraint) The global cardinality constraint $\mathrm{GCC}(x_1, \ldots, x_n, l, u, D)$ holds if and only if for each value v in D, at least l_v and at most u_v variables are assigned to v.

Propagating the constraint consists of removing inconsistent values from variable domains. The following example illustrates the work that needs to be accomplished by a propagator.

Example 5.1 Consider the CSP with six variables $x_1, \ldots, x_6$ with domains, $\mathrm{dom}(x_1) = [2, 2]$, $\mathrm{dom}(x_2) = [1, 2]$, $\mathrm{dom}(x_3) = [2, 3]$, $\mathrm{dom}(x_4) = [2, 3]$, $\mathrm{dom}(x_5) = [1, 4]$, and $\mathrm{dom}(x_6) = [3, 4]$ and a single global cardinality constraint $\mathrm{GCC}(x_1, \ldots, x_6)$ with bounds on the occurrences of values,

v	1	2	3	4
l_v	1	1	1	2
u_v	3	3	3	3.

Each value must be assigned to at least one variable except for value 4 that must be assigned to at least two variables. Each value must be assigned to at most three variables. Enforcing bounds consistency on the constraint reduces the domains of the variables as follows: $\mathrm{dom}(x_1) = [2,2]$, $\mathrm{dom}(x_2) = [1,1]$, $\mathrm{dom}(x_3) = [2,3]$, $\mathrm{dom}(x_4) = [2,3]$, $\mathrm{dom}(x_5) = [4,4]$, and $\mathrm{dom}(x_6) = [4,4]$.

In the following sections, we develop a general approach to propagate the GCC. It consists of dividing the constraint into two simpler constraints. This strategy will allow us to design algorithms for the bounds consistency, range consistency, and domain consistency of the GCC. Moreover, we will study some extensions of the global cardinality constraint and define the limit on these extensions.

A GCC can be decomposed into two constraints: A *lower bound constraint* (LBC) which ensures that all values $v \in D$ are assigned to at least l_v variables, and an *upper bound constraint* (UBC) which ensures that all values $v \in D$ are assigned to at most u_v variables. We will show how to make both constraints locally (bounds or domain) consistent and prove that this is sufficient to make a GCC locally consistent.

5.1 The Upper Bound Constraint (UBC)

The UBC is a generalization of the ALL-DIFFERENT constraint (in the ALL-DIFFERENT constraint $u_v = 1$, for each value v). The algorithm for the bounds consistency of the ALL-DIFFERENT constraint presented in Section 4.2 is based on the concept of Hall intervals (see Definition 3.1). The definition of a Hall interval can be generalized to sets by using the notion of *maximal* capacity. For $S \subseteq D$, let $C(S)$ be the number of variables whose domains are contained in S. The maximal capacity $\lceil S \rceil$ of a set S is the maximum number of variables that can be assigned to the values in S; i.e., $\lceil S \rceil = \sum_{v \in S} u_v$.

Definition 5.2 (Hall Set) A *Hall set* is a set $H \subseteq D$ such that there are $\lceil H \rceil$ variables whose domains are contained in H; i.e., H is a Hall set if and only if $C(H) = \lceil H \rceil$.

The values in a Hall set are fully consumed by the variables that form the Hall set and unavailable for all other variables. Clearly, a UBC is unsatisfiable if there is a set S such that $C(S) > \lceil S \rceil$. We show that the absence of such a set is a sufficient and necessary condition for a UBC to be satisfiable.

Lemma 5.1 *A* UBC *is satisfiable if and only if for any set* $S \subseteq D$, $C(S) \leq \lceil S \rceil$.

Proof We reduce a UBC to an ALL-DIFFERENT constraint. We first duplicate u_v times each value v in the domain of a variable, using different labels to represent the same value. For example, the domain $\{1, 2\}$ with $u_1 = 3$ and $u_2 = 2$ is represented by $\{1a, 1b, 1c, 2a, 2b\}$. Clearly, the UBC is satisfiable if and only if its corresponding ALL-DIFFERENT constraint is satisfiable. In a UBC, the maximal capacity of a set S is given by $\lceil S \rceil$; for the ALL-DIFFERENT constraint, it is given by the cardinality $|S|$ of the set. Hall [28] proved that an ALL-DIFFERENT constraint is satisfiable if and only if for any set S, $C(S) \leq |S|$. Thus, the result holds also for a UBC. □

5.2 The Lower Bound Constraint (LBC)

Next we define some concepts that will be useful for constructing a propagator for the LBC. Let $I(S)$ be the number of variables whose domains intersect the set $S \subseteq D$. The minimal capacity $\lfloor S \rfloor$ of a set S is the minimum number of variables that must be assigned to the values in S; i.e., $\lfloor S \rfloor = \sum_{v \in S} l_v$.

Definition 5.3 (Failure set) A *failure set* is a set $F \subseteq D$ such that there are fewer variables whose domains intersect F than its minimal capacity; i.e., F is a failure set if $I(F) < \lfloor F \rfloor$.

Definition 5.4 (Unstable set) An *unstable set* is a set $U \subseteq D$ such that there are the same number of variables whose domains intersect U as its minimal capacity; i.e., U is an unstable set if $I(U) = \lfloor U \rfloor$.

Definition 5.5 (Stable set) A *stable set* is a set $S \subseteq D$ such that there are more variables whose domains are contained in S than its minimal capacity, and S does

not intersect any failure or unstable sets; i.e., S is a stable set if $C(S) > \lfloor S \rfloor$, $S \cap U = \emptyset$ and $S \cap F = \emptyset$ for all unstable sets U and failure sets F.

In Example 5.1, the set $\{1, 4\}$ is an unstable set since its lower capacity is 3 and only three variable domains (namely $\text{dom}(x_2)$, $\text{dom}(x_5)$, and $\text{dom}(x_6)$) intersect it. The set $\{4\}$ is also an unstable set and $\{2, 3\}$ is a stable set. There are no failure sets in the example but removing variable x_2 would create the failure set $\{1, 4\}$.

Failure, unstable, and stable sets are the main tools to understand how to make an LBC locally consistent. Failure sets determine if an LBC is satisfiable, unstable sets indicate where the domains have to be pruned, and stable sets indicate which domains do not have to be pruned because all of their values have supports.

Lemma 5.2 *An* LBC *is satisfiable if and only if it does not have a failure set.*

Proof To satisfy an LBC, we must associate at least l_v different variables to each value $v \in D$ such that every variable is assigned a single value from its domain. For each value $v \in D$, we construct l_v identical sets T_v^i for $i = 1, \ldots, l_v$ that contain the indices of the variables that have v in their domain; i.e., $T_v^i = \{j \mid x_j \in X \wedge v \in \text{dom}(x_j)\}$. Let $\mathcal{T}$ be the set of all sets T_v^i. To satisfy the LBC, it suffices to find a complete set of distinct representatives (see Section 2.2.3). The variables that are not selected as a representative can be instantiated to any arbitrary value in their domain. From Hall's marriage theorem, we know that such a set of representatives exists if and only if the union of any k sets contains at least k elements. Formally the problem is solvable if and only if $|\bigcup_{T_v^i \in T} T_v^i| \geq |T|$ holds for any $T \subseteq \mathcal{T}$. Applying this theorem here, we have that an LBC is satisfiable if and only if for any set $S \subseteq D$ we have $I(S) \geq \lfloor S \rfloor$. Hence, the absence of a failure set is a necessary and sufficient condition for an LBC to be satisfiable. □

Lemma 5.3 shows that a value in a variable domain that intersects an unstable set has an interval/domain support only if the value belongs to an unstable set.

Lemma 5.3 *A variable whose domain intersects an unstable set cannot be instantiated to a value outside of this set.*

Proof Let U be an unstable set and x a variable whose domain intersects U. If x is instantiated to a value that does not belong to U then U becomes a failure set and the LBC is no longer satisfiable by Lemma 5.2. □

Lemma 5.4 *A variable whose domain is contained in a stable set can be instantiated to any value in its domain.*

Proof By definition, a stable set S does not intersect any unstable or failure set. Thus, for any subset s of S, $I(s) > \lfloor s \rfloor$. If a variable whose domain is contained in S is assigned to a value, the function $I(s)$ will decrease by at most one and therefore s will either stay a stable set or become an unstable set. In either case, no failure set is created and the LBC is still satisfiable. □

A satisfiable LBC has several interesting properties: (i) the union of two unstable sets gives an unstable set, (ii) the union of two stable sets gives a stable set, and (iii) since stable and unstable sets are disjoint, there exists a stable set S and an unstable set U that form a bipartition of D. The bipartition property implies that there are two types of variables: those whose domains are fully contained in a stable set and those whose domains intersect an unstable set.

Lemma 5.5 *If there are no failure sets in the problem, the union of two unstable sets gives an unstable set.*

Proof Let U_1 and U_2 be two unstable sets. We have that,

$$\begin{aligned} I(U_1 \cup U_2) &= I(U_1) + I(U_2) - I(U_1 \cap U_2) && (5.1)\\ &= \lfloor U_1 \rfloor + \lfloor U_2 \rfloor - I(U_1 \cap U_2). && (5.2) \end{aligned}$$

Since there are no failure sets we have

$$\begin{aligned} I(U_1 \cup U_2) &\geq \lfloor U_1 \rfloor + \lfloor U_2 \rfloor - \lfloor U_1 \cap U_2 \rfloor && (5.3)\\ I(U_1 \cap U_2) &\geq \lfloor U_1 \cap U_2 \rfloor && (5.4) \end{aligned}$$

Substituting Inequality 5.4 in Equation 5.2 gives $I(U_1 \cup U_2) \leq \lfloor U_1 \rfloor + \lfloor U_2 \rfloor - \lfloor U_1 \cap U_2 \rfloor$ which is the reverse of Inequality 5.3. We therefore obtain the equality $I(U_1 \cup U_2) = \lfloor U_1 \cup U_2 \rfloor$. □

Lemma 5.6 *If there are no failure sets, there exists a bipartition $\langle U, S \rangle$ of D where U is an unstable set and S is a stable set.*

Proof Let U be the union of all unstable sets. By Lemma 5.5, U is also an unstable set. Since there are no failure sets we have $I(D) \geq \lfloor D \rfloor$. Suppose that $I(D) = \lfloor D \rfloor$, then $U = D$ and $S = \emptyset$. Now suppose that $I(D) > \lfloor D \rfloor$. We have that,

$$\begin{aligned} C(D-U) &= |X| - I(U) \\ &= |X| - \lfloor U \rfloor \\ &> \lfloor D \rfloor - \lfloor U \rfloor \\ &> \lfloor D - U \rfloor. \end{aligned}$$

The set $S = D - U$ is disjoint from all unstable sets and contains more variables than its minimal capacity. Therefore, S is a stable set. Thus there is always a stable and an unstable set that form a bipartition of D □

5.3 An Iterative Algorithm for Local Consistency of the GCC

Suppose we have an algorithm $\mathcal{A}$ that makes the UBC locally consistent and suppose that we have an algorithm $\mathcal{B}$ that makes the LBC locally consistent. To make a GCC locally consistent, we can decompose it into an LBC and a UBC, run $\mathcal{A}$ to prune the domains of the variables, then run $\mathcal{B}$ to further prune the domains. Since the domains can potentially be pruned each time either algorithm is run, we run alternately each algorithm until no more modifications occur. In principle, we might need to repeat this process a large number of times. Surprisingly, only one iteration suffices, as we show with Theorems 5.1, 5.2, and 5.3.

The outline of the proof is as follows. We first prove that if a UBC is satisfiable after running $\mathcal{A}$, the UBC is still satisfiable after running $\mathcal{B}$. We then prove that the UBC is still locally consistent after running $\mathcal{B}$, so there is no need for further pruning.

Theorem 5.1 *Let $\mathcal{A}$ be a propagator that makes the* UBC *bounds (domain) consistent and let $\mathcal{B}$ be a propagator that makes the* LBC *bounds (domain) consistent. If the* GCC *is satisfiable and $\mathcal{B}$ is run after $\mathcal{A}$, $\mathcal{B}$ never creates a set s such that there are more variables whose domains are contained in s than its maximal capacity $\lceil s \rceil$.*

Proof Since the GCC is satisfiable, there are no failure sets and there is an unstable set U and a stable set S that form a bipartition of D. Algorithm $\mathcal{B}$ does not modify the domains of the variables that belong to a stable set. Therefore we know that for all $s \subseteq S$ we have $C(s) \leq \lceil s \rceil$ since the UBC is satisfiable according to $\mathcal{A}$.

We will show that for any set $E \subseteq U \cup S$ we have $C(E) \leq \lceil E \rceil$ and therefore the UBC is still satisfiable after running $\mathcal{B}$. Assume, by way of contradiction, there is a set E that exceeds its capacity; i.e., $C(E) > \lceil E \rceil$. We divide this set into two subsets: let $L = U \cap E$ be the unstable values in E and $F = S \cap E$ be the stable values in E. We also define $R = U - E$ as the unstable values that do not belong to E. We know that $\lceil F \rceil \geq C(F)$ since F is a subset of a stable set and we showed that the property holds for any such a set. We also know that R is not a failure set and U is an unstable set. Therefore we have $I(R) \geq \lfloor R \rfloor$ and $\lfloor L \rfloor + \lfloor R \rfloor = I(L \cup R)$.

$$
\begin{aligned}
\lceil F \rceil + \lfloor L \rfloor + \lfloor R \rfloor &\leq \lceil F \rceil + \lceil L \rceil + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< C(E) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< |\{x \in X \mid \mathrm{dom}(x) \subseteq E \wedge \mathrm{dom}(x) \nsubseteq F\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &< |\{x \in X \mid \mathrm{dom}(x) \cap L \neq \emptyset \wedge \mathrm{dom}(x) \cap R = \emptyset\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(R) &< C(F) + \lfloor R \rfloor \\
\lceil F \rceil &< C(F)
\end{aligned}
$$

The last inequality is incompatible with the hypothesis hence the contradiction hypothesis cannot be true. Notice that the proof holds for both bounds and domain consistency. □

The following definition will be useful to prove the next theorem.

Definition 5.6 A variable $x \in X$ is *consistent with a Hall set* H if the following holds. For bounds consistency, the domain of the variable must have either both or neither bounds in H and for domain consistency, the domain of the variable must be either fully contained in or completely disjoint from H.

Theorem 5.2 *Let $\mathcal{A}$ be a propagator that makes the* UBC *bounds (domain) consistent and let $\mathcal{B}$ be a propagator that makes the* LBC *bounds (domain) consistent. If $\mathcal{B}$ is run after $\mathcal{A}$, the* UBC *is still locally consistent after $\mathcal{B}$ is run.*

Proof Suppose that $\mathcal{A}$ and $\mathcal{B}$ make the constraints locally consistent and neither returns a failure. To prove that the UBC is still locally consistent, we have to show that all variables are still consistent with all Hall sets (see Definition 5.6).

Since $\mathcal{B}$ did not return a failure, there is an unstable set U and a stable set S that form a bipartition of D. Let $H \subseteq D$ be a Hall set. We divide this Hall set into two subsets: $F = H \cap S$ contains the values of H that belong to a stable set and $L = H \cap U$ contains the values of H that belong to an unstable set. We also define $R = U - L$ as the unstable values that do not belong to H. Using these three sets, we will prove that all variables are consistent with H.

The unstable set U can be expressed as the union of L and R and therefore we have $\lfloor L \rfloor + \lfloor R \rfloor = I(L \cup R)$. Similarly, H is the union of F and L and implies $\lceil F \rceil + \lceil L \rceil = C(H) = |\{x \in X \mid \mathrm{dom}(x) \subseteq H \wedge \mathrm{dom}(x) \not\subseteq F\}| + C(F)$. Therefore,

$$\begin{aligned}
\lceil F \rceil + \lfloor L \rfloor + \lfloor R \rfloor &\leq \lceil F \rceil + \lceil L \rceil + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &\leq |\{x \in X \mid \mathrm{dom}(x) \subseteq H \wedge \mathrm{dom}(x) \not\subseteq F\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(L \cup R) &\leq |\{x \in X \mid \mathrm{dom}(x) \cap L \neq \emptyset \wedge \mathrm{dom}(x) \cap R = \emptyset\}| + C(F) + \lfloor R \rfloor \\
\lceil F \rceil + I(R) &\leq C(F) + \lfloor R \rfloor
\end{aligned}$$

By Theorem 5.1 we obtain $C(F) \leq \lceil F \rceil$ and since R is not a failure set, we have $I(R) \geq \lfloor R \rfloor$. Using these two inequalities, we find that R is an unstable set i.e. $I(R) = \lfloor R \rfloor$ and F is a Hall set i.e. $C(F) = \lceil F \rceil$. Using this observation, we now show that all variables whose domains are contained in S are consistent with H.

The Hall set F is a subset of S and since algorithm $\mathcal{B}$ does not modify any variables whose domains are contained in S, algorithm $\mathcal{A}$ already identified F as a Hall set and made all variables consistent with it. Since the variables whose domains are contained in S were not modified by $\mathcal{B}$ they are still consistent with F.

For bounds consistency, a variable whose domain intersects an unstable set such as U and R must have both bounds of the domain in this unstable set. Since $U = L \cup R$, a variable whose domain intersects U must have both bounds in either L or R and therefore be consistent with the Hall set H.

For domain consistency, a variable domain that intersects an unstable set such as U and R must be fully contained in or disjoint from this unstable set. Since $U = L \cup R$, a variable whose domain intersects U must be fully contained in L or R or disjoint from L and R and therefore be consistent with the Hall set H.

We have shown that any variable whose domain is either contained in S or intersects U is consistent with H. Thus all variables are consistent with any Hall set and the UBC is still locally consistent after running $\mathcal{B}$. □

Finally, we show that making the UBC and the LBC locally consistent is equivalent to making the GCC locally consistent.

Theorem 5.3 *A value $v \in dom(x)$ has a support in a* GCC *if and only if it has supports in the corresponding* LBC *and* UBC*.*

Proof Clearly, if there is an assignment t that satisfies the GCC such that $t[x] = v$, this tuple also satisfies the LBC and the UBC. To prove the converse, we consider a value $v \in \text{dom}(x)$ that has a support in the LBC and a (possibly different) support in the UBC. We construct an assignment t such that $t[x] = v$ that satisfies the GCC and therefore prove that $v \in \text{dom}(x)$ also has a support in the GCC. We first instantiate the variable x to v. The LBC and UBC are still satisfiable since the value has a support in both constraints. We now show how to instantiate the other variables.

If there is an uninstantiated variable x whose domain does not intersect any unstable set and is not contained in any Hall set, then the domain of x is necessarily contained in a stable set. By Lemma 5.4 we can instantiate x to any value in its

domain and keep the LBC satisfiable. We therefore choose a solution of the UBC and instantiate x to the same value as it is instantiated in this solution. This operation can create new unstable sets or new Hall sets but keeps both the LBC and the UBC satisfiable. For all variables that intersect an unstable set U, we choose a solution of the LBC and assign the variables to the same values in this solution. We perform the same operation for the variables whose domain is contained in a Hall set H using a solution of the UBC. There will be exactly l_v or u_v variables assigned to a value v depending if the value belongs to U or H, which in either case satisfies both the LBC and UBC. We repeat the above until all variables are instantiated. The constructed tuple t satisfies the LBC and the UBC simultaneously and therefore also satisfies the GCC. □

5.4 Bounds Consistency for the GCC

We showed in the previous section that a propagator for the LBC and a propagator for the UBC are sufficient to enforce bounds or domain consistency on the GCC. We present in this section new algorithms to enforce bounds consistency on the UBC and the LBC. Combined together, these two propagators form a propagator for the bounds consistency of the GCC.

5.4.1 The Upper Bound Constraint (UBC)

Finding an algorithm that makes a UBC bounds consistent is relatively straightforward as we already know of an algorithm for the All-Different constraint that uses the concept of Hall intervals. If there is a variable whose domain is $[a, b]$ and there is a Hall interval $[c, d]$ such that $c \leq a \leq d < b$ holds, the algorithm will update the domain of the variable to $[d+1, b]$. The algorithm introduced in Section 4.2 detects Hall intervals by checking if there are $d-c+1$ variables in an interval $[c, d]$. We can adapt this algorithm to a UBC without altering its complexity provided we can compute the maximal capacity of an interval in constant time. To this end, we use a partial sum data structure, implemented as an array A containing the

partial sums of the maximal capacities $A[i] = \sum_{j=\min(D)}^{i} u_j$. The maximal capacity of an interval $I \subseteq D$ can be computed by subtracting two elements in A since we have $\lceil I \rceil = A[\max(I)] - A[\min(I) - 1]$. Initializing the array A takes $O(D)$ time to compute but this is done once and is reused for any future calls to the propagator. The algorithm time complexity is $O(t + |X|)$ where t is the time required for sorting the variable domains by lower and upper bounds.

5.4.2 The Lower Bound Constraint (LBC)

The Propagator

We now present a propagator for the LBC (see Figure 8) that shrinks the lower bounds of the variable domains received as input. The upper bounds can be updated symmetrically by a similar algorithm and consequently make the LBC bounds consistent.

Algorithm 8: Bounds consistency algorithm for the LBC

```
Let PS be a union-find data structure over the elements in D;
Let Stable = ∅;
for v ∈ D do
    associate l_v empty buckets to the value v;
    if l_v > 0 then
        mark v as a failure element;
D ← D ∪ {−∞, ∞};
associate ∞ buckets to the values −∞ and ∞;
for x_i ∈ X in nondecreasing order of max(dom(x_i)) do
    a ← min(dom(x_i));
    b ← max(dom(x_i));
    z ← min({v ∈ D | v ≥ a, a has an empty bucket});
    if z > a then
        union (PS, a, a + 1, ..., min(b, z));
    if z > b then
        S ←findSet (PS, b);
        Stable ← Stable ∪ {S};
    else
        add a token in one of the empty buckets of z;
        z ← min({v ∈ D | v ≥ a, a has an empty bucket});
        NewMin[i] ← min({v ∈ D | v ≥ a, v has a failure flag});
        if z > b then
            j ← max({v ∈ D | v ≤ b, v has an empty bucket});
            reset the failure flag for all elements in (j, b];
if |{v ∈ D | v has a failure flag}| > 0 then
    return Failure;
for x_i ∈ X such that ∀ S ∈ Stable, dom(x_i) ⊈ S do
    dom(x_i) ← dom(x_i) − [min(dom(x_i)), NewMin[i]);
return Success;
```

The initialization step assigns to each value $v \in D$ exactly l_v empty *buckets* corresponding to the minimal capacity to be filled for v and sets a *failure flag* for v which indicates if v belongs to a failure set. The union-find data structure PS covers all values in D and contains potential stable sets. If the greatest element of a set $S \in PS$ is in a stable set then S is fully contained in this stable set. Stable sets are stored in the variable *Stable*.

Our algorithm processes each variable $x_i \in X$ in nondecreasing order by upper bound. As in the algorithm of Lipski *et al.* [50], it searches for the smallest value $v \in \mathrm{dom}(x_i)$ that has an empty bucket and fills it in with a token. If $v > \min(\mathrm{dom}(x_i))$ and v belongs to a stable set then the interval $I = [\min(\mathrm{dom}(x_i)), v]$ is contained in this stable set. The algorithm regroups all values in I in its variable PS. If there are no empty buckets in $\mathrm{dom}(x_i)$ then $\max(\mathrm{dom}(x_i))$ belongs to a stable set and so do all the values that belong to the same set in PS.

The algorithm initially assumes that all values belong to a failure set. When processing a variable x_i, if an interval $I = [a, \max(\mathrm{dom}(x_i))]$ has no empty buckets then it contains the domains of at least $\lfloor I \rfloor$ variables and thus cannot be a failure set. The algorithm unsets the failure flags for all values in I. If a value still has a failure flag set after processing all the variables then the LBC is unsatisfiable.

To shrink the domains, the algorithm stores in $NewMin[i]$ the smallest value $v \in \mathrm{dom}(x_i)$ with a failure flag. If $\mathrm{dom}(x_i)$ intersected an unstable set U, v would be the smallest value in $\mathrm{dom}(x_i) \cap U$. If no values in $\mathrm{dom}(x_i)$ have a failure flag, x_i belongs to a stable set and $NewMin[i]$ remains undefined. After processing all variables, the algorithm assigns the new lower bound $NewMin$ to the variables that are not contained in a stable set.

Example 5.2 Figure 5.1 shows a trace of the algorithm on the CSP introduced in Example 5.1 at the beginning of this chapter. Initially, all buckets are empty and all values are marked with a failure flag. Figure 5.1 shows the data structures as the algorithm iterates through the variables. The empty circles represent the empty buckets, the crossed circles represent the buckets containing a token, a letter f symbolizes a failure flag, and the state of the variables PS and *Stable* are also represented by the sets of values. Upon completion of the algorithm, the new

domains of the variables are: $x_1 \in [2,2]$, $x_2 \in [1,2]$, $x_3 \in [2,3]$, $x_4 \in [2,3]$, $x_5 \in [4,4]$, and $x_6 \in [4,4]$.

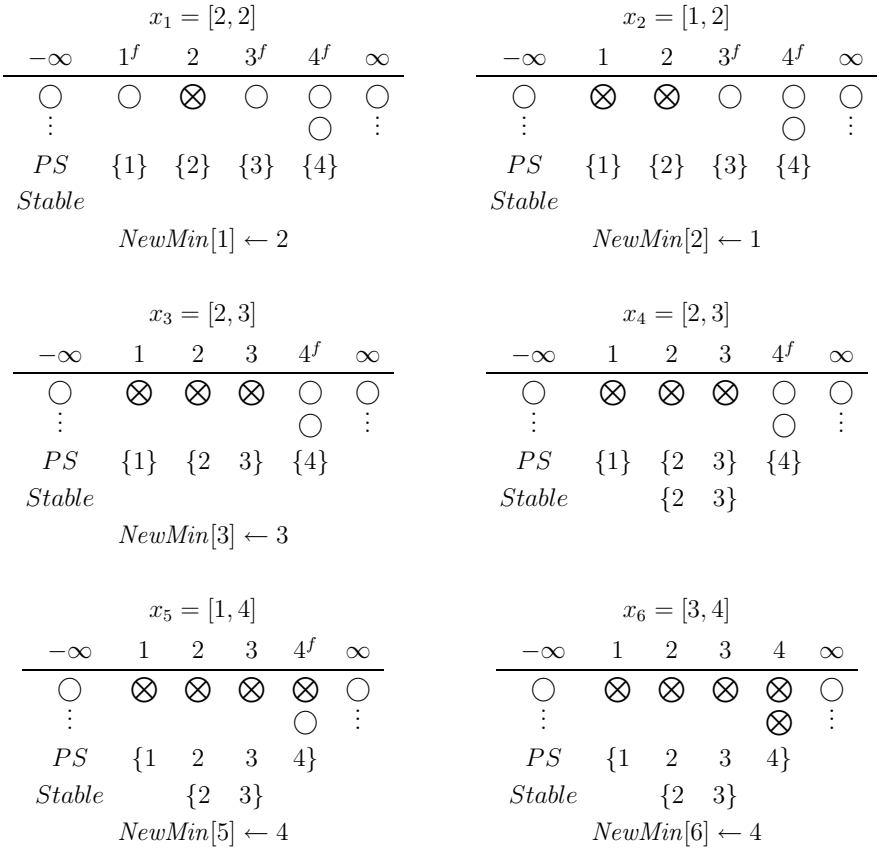

Figure 5.1: Trace of Algorithm 8

Correctness

We must prove that the algorithm depicted in Figure 8 returns *Success* if and only if the LBC is satisfiable.

During its execution, the algorithm constructs an assignment that satisfies the LBC. If when processing variable x_i a token is added to the bucket of value v, we fix $x_i = v$. Otherwise if no token is added to any bucket, then x_j can be assigned to any value in $\text{dom}(x_j)$. When the algorithm returns success, every bucket is filled in with a token. This proves that our assignment satisfies the conditions imposed by the LBC. Therefore, when the algorithm returns *Success*, the LBC is satisfiable. We now want to prove the converse.

Suppose there exists a solution L. Let x_i be the first variable processed by our algorithm such that x_i is assigned to v but v is assigned to x_j in solution L. Suppose that in solution L, x_i is assigned to u. We know that $\max(x_i) \leq \max(x_j)$. Therefore, there exists a solution L' such that $x_i = v$ and $x_j = u$. We replace L by L' and we continue the execution of the algorithm knowing that there still exists a solution. We process all variables by swapping values as described until all variables have been processed. At each iteration, we are always guaranteed that a solution compatible with our assignments exists. Therefore, if there exists a solution, the algorithm returns *Success*.

We now prove that a variable x_i intersecting an unstable set U has its lower bound shrunk to the value $\min(\text{dom}(x_i) \cap U)$. When processing a variable, we suppose that its domain intersects an unstable set and will have to be shrunk. We store in $NewMin[i]$ the lower bound to which it has to be shrunk. Values having their failure flag unset belong to a set S whose lower capacity is at most equal to the number of variable domains contained in S. More formally, we have $C(S) \geq \lfloor S \rfloor$. If variable x_i is assigned to a value in S, a subset of S containing this value would become (or remain) a stable set. Clearly, if x_i intersects an unstable set, it should not be assigned to any value in S. This is why we fix $NewMin[i]$ to the smallest value in $\text{dom}(x_i)$ that has a failure flag. Therefore, it is sufficient to prove that the algorithm properly detects variable domains intersecting unstable sets to conclude the proof of correctness.

If all buckets associated to values in the domain of x_i are filled in when processing x_i, we conclude that $\text{dom}(x_i)$ is contained in a stable set. In fact, x_i is a variable that can be assigned to any value in its domain. If we assign x_i to v, the variable that was assigned to v is now free to be assigned to any value in its domain which in turn can be assigned to another value freeing the variable that was assigned to this value and so on. The stable set starts at the smallest value that can be reached by a free variable and finished at $\max(\text{dom}(x_i))$. The data structure PS allows to quickly find the smallest value that can be reached by a free variable. Suppose variable x_j fills the bucket of value v for $v > \min(\text{dom}(x_j))$. We know that if value v is taken by a free variable then x_j becomes free and can be assigned to any value in $[\min(x_j), v]$. We therefore unite these values in the data structure PS. When we discover that $\max(\text{dom}(x_i))$ belongs to a stable set, we know that the whole set in PS that contains $\max(\text{dom}(x_i))$ is a stable set. Once all stable sets have been discovered and that no failure sets have been discovered, we conclude that every value that does not belong to a stable set belongs to an unstable set. Therefore, if a variable domain is not contained in a stable set, its lower bound must be shrunk to $NewMin[i]$.

Time Complexity Analysis

A naïve implementation of our algorithm has time complexity $O(t+|X|\,|D|)$, where t is the complexity of sorting the intervals by upper bounds. We will show how to improve the complexity to $O(t + |X|)$.

To obtain a complexity independent of $|D|$, we consider the variables as semi-open intervals where $x_i = [a_i, b_i)$ and define the set D' as the union of the lower bounds a_i and the open upper bounds b_i of each variable. The size of D' is bounded by $2\,|X|$. Let c and d be two consecutive values in D' and let $I = (c, d]$ be a semi-open interval. We modify the algorithm to assign $\lfloor I \rfloor$ buckets to the value d using a partial sums data structure (see Section 5.4.1). We then run the algorithm as before using the set D' instead of D. This modification improves the time complexity to $O(t + |X|^2)$.

To get linear complexity, we implement the buckets using a union-find data

structure and an array of integers that stores the number of empty buckets associated to a value v. If all buckets of value v are filled in, the algorithm merges the value v with the next element in D'. Requesting n times the next value with a free bucket is a linear time operation using the interval union-find data structure [20]. The algorithm takes $O(t + |X|)$ steps using the interval union-find for the failure flags, the stable sets *Stable*, and the potential stable sets *PS*.

Although the interval union-find data structure gives the best theoretical time complexity, we found that it did not result in the fastest code in practice in spite of our best efforts to optimize the code. In our experiments (see Section 5.4.3), we use instead the tree data structure described in Section 4.2 to obtain an algorithm with $O(t+|X|\log|X|)$ time complexity. This tree data structure even offers slightly better performance than the standard union-find data structure which runs in $O(t+|X|\alpha(|X|))$ where α is the inverse of a certain Ackerman function.

5.4.3 Experiments

We implemented our bounds consistency propagator for the GCC (denoted BC) using the ILOG Solver C++ library, Version 4.2 [3]. By extending Puget's idea [63] for the All-Different constraint, in addition to enforcing bounds consistency, we remove from the variable domains all values v that have been instantiated u_v times. We denote this new consistency that now allows holes in the domain by BC+. We used Regin's propagator [68] for domain consistency (denoted DC) that is provided in the ILOG library. We also used a propagator distributed with ILOG that enforces a consistency (denoted CC) equivalent to enforcing one GCC per value v where $l_w = 0$ and $u_w = \infty$ for all $w \neq v$ and l_v and u_v are unchanged for value v (see [3, 34]).

We ran the algorithms on different benchmark problems. We used a 2.40 GHz Pentium 4 with 1 GB of main memory. We averaged the running time of each problem over 10 runs and 100 runs for random problems. We used the minimum domain size variable ordering heuristic unless otherwise mentioned.

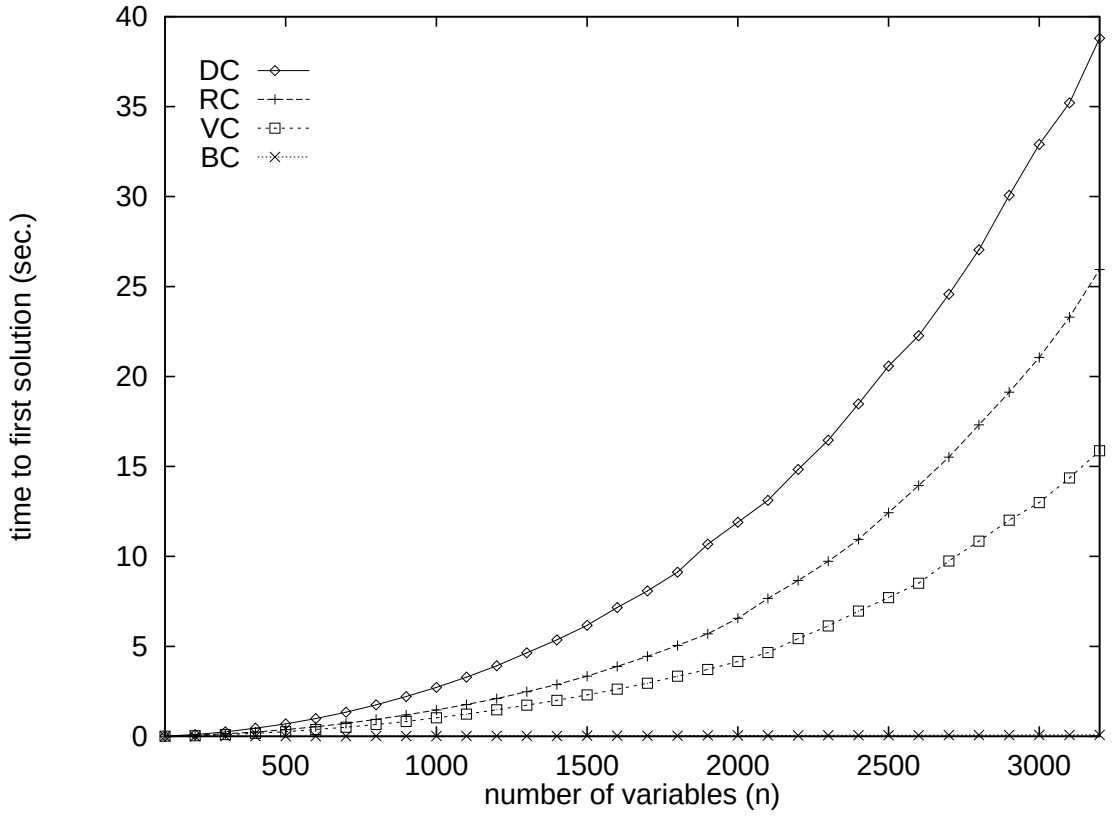

Figure 5.2: Time (sec.) to first solution for Pathological problems.

Pathological Problem We first study the pathological problem introduced by Puget [63] to study the worst case behavior of his propagator for the ALL-DIFFERENT constraint. In the pathological problem, we have $2n + 1$ variables with domains $\text{dom}(x_i) = [i - n, 0]$ for $0 \leq i \leq n$ and $\text{dom}(x_i) = [0, i - n]$ for $n < i \leq 2n$. We have $l_v = 0$ and $u_v = 1$ for every value. Figure 5.2 shows a strong improvement compared to other propagators. We obtain similar results when generalizing with $u_v = c$.

Instruction Scheduling Problems We compared the propagators on some scheduling problems involving multiple-issue pipelined processors. Following [82], we have a set of n variables representing the execution times of tasks. For some

Table 5.1: Time (sec.) to optimal solution for instruction scheduling problems; (left) issue width = 2; (right) issue width = 2 + 2 = 4. A blank entry means the problem was not solved within a 10 minute time bound.

n	CC	DC	BC
69	0.01	0.12	0.00
70	0.00	0.07	0.00
111	0.03	0.75	0.01
211	0.51	9.24	0.07
214	0.60	9.29	0.09
216	2.67	124.07	0.31
220	5.09	285.91	0.52
690	1.34	493.15	1.67
856		471.16	3.84
1006			8.70

n	CC	DC	BC
69	0.00	0.07	0.00
70	0.01	0.07	0.00
111	0.03	0.44	0.01
211	0.56	7.16	0.11
214	0.61	7.85	0.13
216	2.78	89.61	0.48
220	2.90	98.15	0.57
690	2.17	307.20	2.81
856			
1006	307.00		14.44

pairs of variables (x_i, x_j), we have a latency constraint of the form $x_i \geq x_j + l_{ij}$ where l_{ij} is a small integer. The issue width of a processor is the maximum number of instructions that can be issued at each clock cycle. We scheduled instructions for two different processors. One processor has an issue width of two for any kind of instructions. The second processor has an issue width of four divided as follows: an issue width of two for floating point instructions and an issue width of two for integer instructions. A GCC over a set of instructions insures that the schedule respects the issue width constraint. Table 5.1 shows results we obtained when solving problems from the SPEC95 floating point, SPEC2000 floating point, and MediaBench benchmarks. Our BC propagator outperforms the other propagators.

Car Sequencing Problem The car sequencing problem [3] occurs in the car industry where cars on an assembly line must be sequenced in a way that machines are not overloaded. The problem involves n variables, n values and equiprobable configurations of 5 options. Approximately $4n$ GCC's are posted on the variables.

Table 5.2: (left) Time (sec.) to first solution or to detect inconsistency for car sequencing problems; (right) number of backtracks (fails).

n	CC	DC	BC	BC+
10	0.07	0.07	0.09	0.09
15	3.40	3.88	5.39	4.12
20	20.65	30.05	30.95	21.83
25	131.27	203.23	163.97	118.57

n	CC	DC	BC	BC+
10	437	321	460	429
15	13,849	9,609	19,958	13,565
20	55,657	52,581	105,436	55,580
25	255,690	250,042	520,519	255,653

Table 5.3: (left) Time (sec.) to first solution for sports league scheduling problems; (right) number of backtracks (fails). A blank entry means the problem was not solved within a 10 minute time bound.

n	CC	DC	BC	BC+
8	0.19	0.16	0.04	0.18
10	1.10	0.12	0.03	0.19
12	1.98	1.70	51.71	2.07
14	11.82	8.72		9.98

n	CC	DC	BC	BC+
8	1308	914	136	942
10	5767	428	54	689
12	6449	4399	149728	5356
14	33901	19584		22176

Table 5.2 shows that our BC+ propagators is almost as powerful as DC and stronger than other propagators when n increases.

Sport League Scheduling Problem The sport league scheduling problem consists of planning when sport leagues should meet to play a match. For this problem, there are n^2 variables, n values, and $n/2$ GCC's. Table 5.3 shows that our propagator is no more than 15% slower than the fastest propagator.

Random problems To study the asymptotic behavior of our algorithm, we generated and solved several random problems. The problems consists of a single GCC over n variables. Each variable is given an initial domain $[a, b]$ where a and b are uniformly chosen in the interval $[1, \frac{n}{2}]$ such that $a \leq b$. Since GCC is the only constraint, most of the run-time is spent in the propagator. We clearly see in Table 5.4

n	DC	BC
100	0.02	0.01
200	0.23	0.02
400	2.55	0.08
800	26.14	0.33
1600	266.80	1.24

	DC			BC		
n	$d/2$	d	$2d$	$d/2$	d	$2d$
100	0.00	0.01	0.33	0.00	0.00	0.00
200	0.00	0.07	4.81	0.00	0.01	0.01
400	0.01	0.60	74.88	0.00	0.03	0.04
800	0.03	4.58		0.01	0.15	0.16
1600	0.20	34.78		0.02	0.70	0.62

Table 5.4: Time (sec.) to first solution or to detect inconsistency for random problems where the bounds on number of occurrences of each value were (left) $[0, 2]$; (right) chosen uniformly at random from {[0, 1], [0, 2], [1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [2, 4]}. A blank entry means some problems could not be solved within a 10 min. time bound.

the cubic behavior of the DC propagator and the almost linear behavior of our BC propagator. The CC propagator is not shown in the table since it could not solve some of the smallest problems under a threshold of 10 minutes.

Katriel and Thiel provided us with an implementation of their algorithm [39] which we ran on the same problems presented above. We paid attention to eliminating performance gains that would benefit the best implementation as opposed to the fastest algorithm. To reduce dissimilarities in the implementation, we used the same code for sorting the variables. We also disabled the code that prunes the cardinality variables in Katriel and Thiel's algorithm since our propagator does not prune these variables. Our propagator was never slower when solving the Pathological problem. In fact, we recorded a speedup of 75% on some instances. We were never slower on the instruction scheduling problems and the car sequencing problems with maximum recorded speedup of 13% and 26%. Katriel and Thiel's algorithm was never slower in either the sports league scheduling problem nor the random problems, with a maximum speedups of 8% and 13%.

5.5 Range Consistency for the GCC

Enforcing range consistency consists of removing values in variable domains that do not have an interval support. Since we are only interested in interval supports, we assume without loss of generality that variable domains are represented with intervals $\text{dom}(x_i) = [a, b]$.

We introduce an algorithm that achieves range consistency in $O(t + |X| + N)$ steps in the amortized sense where t is the time required to sort $|X|$ variables by lower and upper bounds and N is the number of values with a null lower capacity l_v. To obtain this complexity, we do an amortized running time analysis. Some calls to the propagator can take up to $O(|X|^2 + N)$ instructions but will necessarily be followed by fast calls requiring fewer instructions.

The first step of our algorithm is to make the variable domains bounds consistent using the algorithm described in Section 5.4. We then study Hall intervals and unstable sets in bounds consistent problems.

In order to better understand the distribution of Hall intervals, unstable sets, and stable intervals over the domain D, we introduce the notion of *characteristic interval*.

Definition 5.7 (Characteristic Interval) Given a CSP, a *characteristic interval* I is an interval in D such that all variable domains have either both bounds in I or both bounds outside of I and at least one variable domain has both bounds in I.

From the notion of characteristic interval follows the notion of *basic characteristic interval*.

Definition 5.8 (Basic Characteristic Interval) Given a CSP, a *basic characteristic* interval is a characteristic interval that cannot be expressed as the union of two or more characteristic intervals.

A characteristic interval can always be expressed as the union of basic characteristic intervals. We also observe the following property.

Lemma 5.7 *Basic characteristic intervals are either disjoint or nested.*

Proof Suppose $I_1 = [a, b]$ and $I_2 = [c, d]$ are basic characteristic intervals such that $a \leq c \leq b \leq d$. Then no variable domain has its lower bound in $[a, c)$ and its upper bound in $[c, b]$ since I_2 is a characteristic interval. Then $[a, c)$ and $[c, b]$ are two characteristic intervals which contradict the fact that I_1 is a basic characteristic interval. The argument also applies with intervals $[c, b]$ and $(b, d]$. Therefore a basic characteristic interval can only be fully contained in another characteristic interval or be disjoint. □

We then show how characteristic intervals can be used to make a problem range consistent.

Lemma 5.8 *In a bounds consistent problem, a basic Hall interval is a basic characteristic interval.*

Proof In a bounds consistent problem, no variables have one bound within a Hall interval and the other bound outside of the Hall interval. Therefore every basic Hall interval is a basic characteristic interval. □

Definition 5.9 (Maximum Stable Interval) A *maximum stable interval* S is a stable interval such that any other stable interval is either contained in S or disjoint to S.

Lemma 5.9 *In a bounds consistent problem, a maximum stable interval is a characteristic interval.*

Proof Lemma 5.6 states that in a bounds consistent problem, stable intervals and unstable sets form a partition of the domain D. Therefore, either a variable domain intersects an unstable set and has both bounds in this unstable set or it does not intersect an unstable set and is fully contained in a stable interval. Consequently, a maximum stable interval is a characteristic interval. □

Lemma 5.10 *Any unstable set can be expressed as the union and exclusion of basic characteristic intervals*

Proof Let U be an unstable set and I be the smallest interval that covers U. Since any variable domain that intersects U has both bounds in U, then I is a characteristic interval. Moreover, $I - U$ forms a series of intervals that are in I but not in U. A variable domain contained in I must have either both bounds in an interval of $I - U$ such that it does not intersect U or have both bounds in U. Therefore the intervals of $I' = I - U$ are characteristic intervals and U can be expressed as $U = I - I'$. □

Example 5.3 shows how characteristic intervals appear in a problem.

Example 5.3 Consider the following variable domains subject to a GCC where $l_v = 1$ and $u_v = 2$ for every value v.

$$\begin{array}{lll} \mathrm{dom}(x_1) = [2,4] & \mathrm{dom}(x_2) = [2,5] & \mathrm{dom}(x_3) = [3,3] \\ \mathrm{dom}(x_4) = [3,3] & \mathrm{dom}(x_5) = [4,9] & \mathrm{dom}(x_6) = [5,9] \\ \mathrm{dom}(x_7) = [6,8] & \mathrm{dom}(x_8) = [6,8] & \mathrm{dom}(x_9) = [6,8] \\ \mathrm{dom}(x_{10}) = [7,7] & \mathrm{dom}(x_{10}) = [7,7] & \end{array}$$

Notice that the variable domains are bounds consistent. We detect the following characteristic intervals. The intervals $[3,3]$ and $[7,7]$ are Hall intervals. The interval $[6,8]$ is a stable interval. The interval $[2,9]$ contains values belonging to an unstable set. The unstable set can be computed as follows: $U = [2,9] - [3,3] - [6,8] = \{2,4,5,9\}$.

5.5.1 Finding the Basic Characteristic Intervals

Using the properties of basic characteristic intervals, we propose a new algorithm that makes a problem range consistent and has a time complexity of $O(t + C|X| + N|X|)$ where t is the time complexity for sorting n variables, C is the number of basic characteristic intervals, and N is the number of values whose lower capacity is null. We later show that under amortized analysis, our propagator has a time complexity of $O(t + |X| + N)$. Our algorithm proceeds in four steps:

1. Make the problem bounds consistent in $O(t + |X|)$ steps (see Section 5.4).
2. Sort the variable domains by increasing lower bounds in $O(t)$ steps.
3. Find the basic characteristic intervals in $O(|X|)$ steps.
4. Prune the variable domains in $O(C|X| + N|X|)$ steps.

Steps 1 and 2 are not a concern since they can be implemented using the bounds consistency algorithm described in Section 5.4. We focus our attention on Steps 3 and 4.

Step 3 of our algorithm finds the basic characteristic intervals. In order to discover these intervals, we maintain a stack S of intervals that are potentially basic characteristic intervals. We initialize the stack by pushing the infinite interval $[-\infty, \infty]$. We then process each variable domain in ascending order of lower bound. Let I be the current variable domain and I' the interval on top of the stack. If the variable domain is contained in the interval on top of the stack ($I \subseteq I'$), then the variable domain could potentially be a characteristic interval and we push it on the stack. If the variable domain I has its lower bound in the interval I' on top of the stack and its upper bound outside of this interval, then neither I or I' can be characteristic intervals, although the interval $I \cup I'$ could potentially be a characteristic interval. In this case, we pop I' off the stack and we assign I to be $I \cup I'$. We repeat the operation until I is contained in I'. Note that at any time, the stack contains a set of nested intervals.

If we process a variable domain whose lower bound is greater than the upper bound of the interval I' on the stack, then by construction of the stack, I' is a basic characteristic interval. We pop this characteristic interval off the stack S. We continue popping characteristic intervals off the stack until the current variable domain intersects the interval on the stack. Each characteristic interval that the algorithm finds is pushed on the stack Q and will be processed later.

Once all characteristic intervals have been pushed on the stack Q, the algorithm prints basic characteristic intervals omitting to print characteristic intervals that are stable intervals already included in another stable interval. The algorithm also

identifies the type of each characteristic interval: a Hall interval, a stable interval, or an interval that contains values of an unstable set. This is done by maintaining a counter c_1 that keeps track of how many variable domains are contained in an interval on the stack. Counter c_2 is similar but only counts the first $\lfloor A \rfloor$ variables contained in each sub-characteristic interval A where $\lfloor A \rfloor$ is the lower capacity of the interval A. A characteristic interval I is a stable interval if c_2 is greater than $\lfloor I \rfloor$. A characteristic interval I contains values of an unstable set if $c_2 = \lfloor I \rfloor$ and that I is not contained in any stable interval. Finally, an interval I is a Hall interval if its counter c_1 is equal to the upper capacity of the interval $\lceil I \rceil$. We ignore characteristic intervals with $c_2 < \lfloor I \rfloor$ since those intervals are not used to define Hall intervals, stable intervals or unstable sets.

Algorithm 9: Printing the basic characteristic intervals in a bounds consistent problem.

Input: X denotes the set of variable domains sorted by non-decreasing lower bounds

Output: Prints the basic characteristic intervals and specifies if they are Hall intervals, stable intervals or contain values of an unstable set

$S \leftarrow$ empty stack, $Q \leftarrow$ empty stack
$\texttt{push}(S, \langle [-\infty, \infty], 0, 0 \rangle)$
// Add a dummy variable domain forcing stack clearance at termination
$X \leftarrow X \cup [\max(D) + 1, \max(D) + 3]$
for $x \in X$ **do**
  while $\max(\texttt{top}(S).interval) < \min(dom(x))$ **do**
    $\langle I, c_1, c_2 \rangle \leftarrow pop(S)$
    $\texttt{push}(Q, \langle I, c_1, c_2 \rangle)$
    $\langle I', c_1', c_2' \rangle \leftarrow pop(S)$
    $\texttt{push}(S, \langle I', c_1 + c_1', c_2' + \min(c_2, \lfloor I \rfloor) \rangle)$
  $I \leftarrow \text{dom}(x)$, $c_1 \leftarrow 1$, $c_2 \leftarrow 1$
  while $\max(\texttt{top}(S).interval) \leq \max(I)$ **do**
    $\langle I', c_1', c_2' \rangle \leftarrow \texttt{pop}(S)$
    $I \leftarrow I \cup I'$, $c_1 \leftarrow c_1 + c_1'$, $c_2 \leftarrow c_2 + c_2'$
  $\texttt{push}(S, \langle I, c_1, c_2 \rangle)$
$stableCount \leftarrow 0$
$\texttt{pop}(S)$ // S now only contains the infinite interval.
while $Q \neq \emptyset$ **do**
  $\langle I, c_1, c_2 \rangle \leftarrow \texttt{pop}(Q)$
  while $\max(I) < \min(\texttt{top}(S).interval)$ **do**
    $\langle I', c_1', c_2' \rangle \leftarrow \texttt{pop}(S)$
    if $c_2 > \lfloor I \rfloor$ **then** $stableCount \leftarrow stableCount - 1$
  if $\lceil I \rceil = c_1$ **then** print I is a Hall interval
  else if $stableCount = 0$ **then**
    if $c_2 > \lfloor I \rfloor$ **then** print I is a stable interval
    else print I contains unstable values
  if $c_2 > \lfloor I \rfloor$ **then** $stableCount \leftarrow stableCount + 1$
  $\texttt{push}(S, \langle I, c_1, c_2 \rangle)$

The time complexity of Algorithm 9 is bounded by the calls to *push* and *pop*. In the first part of the algorithm, a variable domain can be pushed on the stack S, popped off S, and merged with another interval only once. Moreover, the lower capacity $\lfloor I \rfloor$ and upper capacity $\lceil I \rceil$ of an interval I can be computed in constant time using the partial sums data structure described in Section 5.4.1. A maximum of $|X|$ characteristic intervals are pushed on Q. In the second part of the algorithm, each characteristic intervals is popped off Q only once and pushed and popped off S only once as well. Algorithm 9 has therefore a time complexity of $O(|X|)$.

Once the basic characteristic intervals are listed in non-increasing order of upper bounds, we can easily enforce range consistency on the variable domains. We simultaneously iterate through the variable domains and the characteristic intervals both sorted by non-increasing order of upper bounds. If a characteristic interval is contained in a variable domain, this characteristic interval should be removed from the variable domain. Moreover, variable domains that are only contained in characteristic intervals containing values from an unstable set should remove from their domains values whose lower capacity is null. The pruning process requires $O(C|X| + N|X|)$ steps where C is the number of characteristic intervals and N the number of values with null lower capacity. We recall that the number of characteristic intervals C is bounded by the number of variables $|X|$. This complexity can be further improved as explained in the next section.

Algorithm 10: Enforcing range consistency on the GCC.

Input: The variable domains sorted by non-increasing upper bound and the characteristic intervals C returned by Algorithm 9 sorted by non-increasing upper-bound

$j \leftarrow 1$ // Index of next characteristic interval to be processed
$I \leftarrow [-\infty, \infty]$ // Current characteristic interval
$V \leftarrow \emptyset$ // Variables contained in current characteristic interval
$T \leftarrow \emptyset$ // Union of all variables on the stack
$S \leftarrow$ empty stack // Stack of tuples $\langle \text{Interval}, \text{Variables} \rangle$
for $x_i \in X$ *in non-increasing order of upper bounds* **do**
  if $j < |C| \wedge dom(x_i) \subseteq C_j$ **then**
    $\texttt{push}(S, \langle I, V \rangle)$
    $T \leftarrow T \cup V$
1   **for** $x_k \in T$ **do**
      $\text{dom}(x_k) \leftarrow \text{dom}(x_k) - C_j$
    $I \leftarrow C$
    $V \leftarrow \emptyset$
    $j \leftarrow j + 1$
  while $dom(x_i) \not\subseteq I$ **do**
    $T \leftarrow T - V$
    $\langle I, V \rangle \leftarrow \texttt{pop}(S)$
  // If interval I contains values from an unstable set
2 **if** $\texttt{unstable}(I)$ **then**
    $\text{dom}(x_i) \leftarrow \text{dom}(x_i) - \{v \in I \mid l_v = 0\}$
  $V \leftarrow V \cup \{x_i\}$

5.5.2 Dynamic Case

We want to maintain range consistency when a variable domain $\text{dom}(x_i)$ is modified by the propagation of other constraints. Notice that if the bounds of $\text{dom}(x_i)$ change, new Hall intervals or unstable sets can appear requiring other variable domains to be pruned. We only need to prune the domains according to these new Hall intervals and unstable sets.

We make the variable domains bounds consistent and find the characteristic intervals as before in $O(t+|X|)$ steps. We compare the characteristic intervals with those found in the previous computation and perform a linear scan to mark in linear time all new characteristic intervals. Two cases can occur. A new characteristic interval could appear or a characteristic interval I can be fragmented into multiple characteristic intervals whose union is I.

We modify Algorithm 10 to become incremental as follows. Before executing the *for* loop on line 1, we should test if C_j is a newly discovered interval. If C_j is not a newly discovered interval or is a fragment of an already known characteristic interval, the values in the variable domains have already been removed. There is no need to execute this part of the code a second time. If the characteristic interval I has already been processed in the *if* statement on line 2, there is no need to process it a second time.

Based on these modifications, the new time complexity of Algorithm 10 is $O(\max(1, C')|X| + N|X|')$ where C' is the number of newly discovered characteristic intervals and $|X|'$ is the number of variables affected by the *if* statement on line 2.

We amortize the time complexity of Algorithm 9 and Algorithm 10 when run k times ($k \geq |X|$) on a branch of the search tree. Let C_i be the number of newly discovered characteristic intervals during the i^{th} execution. Since the number of characteristic intervals is bounded by $|X|$ we have $\sum_i^k C_i \leq |X|$. Let $|X|_i$ be the number of variables processed in the *if* statement on line 2 during the i^{th} iteration. Since each variable is processed only once we have $\sum_i^k |X|_i \leq |X|$. The amortized analysis is as follows:

$$\begin{aligned}\frac{1}{k}\sum_{i}^{k}[O(t+|X|)+O(\max(1,C_i)|X|+N|X|_i)] &= \frac{1}{k}O(kt+k|X|+k|X|+N|X|)\\ &= O(t+|X|+N)\end{aligned}$$

Therefore, the algorithm maintains range consistency of the GCC in $O(t+|X|+N)$ steps where t is the time required to sort $|X|$ variables and N is the number of values whose lower capacity l_v is null. Notice that if $l_v > 0$ for all values, we obtain the same complexity as the algorithm for bounds consistency of the GCC proposed in Section 5.4. Moreover, if $l_v = 0$ for all values, there is no possibility for the existence of an unstable set and therefore the *if* statement on line 2 in Algorithm 10 is never executed. We therefore obtain an amortized complexity of $O(t + |X|)$. Once more, this is the same time complexity as the algorithm we proposed for the bounds consistency of the GCC.

5.5.3 Experiments

We tested our propagator for the range consistency of the GCC on the sport tournament scheduling problem (see problem 26 in CSPLib [25]). This problem consists of planning when sport leagues should meet to play a match. A feasible solution is a grid with $n-1$ columns representing $n-1$ weeks and $\frac{n}{2}$ rows representing as many periods. In each cell, there is an unordered pair of teams representing a match. Each team must appear in each column. Each team must appear at least once and a most twice on each row. Each of the $\frac{n(n-1)}{2}$ possible pairs must appear once in the grid.

We compared our propagator with those provided in the ILog library [3] and our propagators for bounds consistency presented in Section 5.4. We denote by *Basic GCC* the propagator provided in ILog with the *IlcBasic* parameter and by *DC* the propagator in ILog achieving domain consistency. *BC* is the propagator for bounds consistency presented in Section 5.4. *BC+* is the same propagator that, in addition to achieving bounds consistency, removes from the domains the value v when v has already been assigned to u_v variables. We denote by *Basic + BC*

n	Basic GCC	DC	BC	BC+	Basic + BC	Range
6	0.01	0.01	0.01	0.01	0.01	0.01
8	0.56	0.48	0.10	0.49	0.40	0.60
10	3.05	0.37	0.08	0.54	0.40	0.64
12	5.80	5.09	145.31	5.86	5.97	7.22
14	34.17	25.85		28.39	28.76	35.30

Table 5.5: Time (s) to find the first solution to the sport tournament scheduling problem with n teams. Empty entries represent problems that could not be solved within a 10 minute threshold.

n	Basic GCC	DC	BC	BC+	Basic + BC	Range
6	5	5	5	5	5	5
8	1308	914	136	942	932	944
10	5767	428	54	689	441	678
12	6449	4399	149728	5356	4980	5372
14	33901	19584		22176	20172	22147

Table 5.6: Number of backtracks in the search tree before reaching the first solution to the sport tournament scheduling problem with n teams. Empty entries represent problems that could not be solved within a 10 minute threshold.

the combination of the propagator provided in ILog with the *IlcBasic* parameter and our propagator achieving bounds consistency. Finally, we denote by *Range* our propagator achieving range consistency presented in this section.

Table 5.5 reports the time to solve the sport tournament scheduling problem. Table 5.6 reports the number of backtracks that occurred during the search. We see that even though our propagator does not outperform the propagator achieving domain consistency (DC), it remains competitive.

5.6 Domain Consistency for the GCC

Régin [68] showed how to enforce domain consistency on GCC in $O(|X|^2|D|)$ steps (see Section 3.3.2). For the special case of the all-different constraint, the same problem can be solved in $O(|X|^{1.5}|D|)$ steps (see Section 3.3.1). In this section we propose an algorithm for domain consistency on the GCC that runs in $O(|X|^{1.5}|D|)$ time and therefore is as efficient as the algorithm for domain consistency on the all-different constraint.

Our approach is similar to the one used by Régin [67] for propagating the *all-different* constraint except that our algorithm proceeds in two passes. The first pass makes the UBC domain consistent and the second pass makes the LBC domain consistent. As shown in Theorem 5.2 in Section 5.3, this suffices to make the GCC domain consistent.

5.6.1 Matching in a Graph

For the UBC and LBC problems, we will need to construct a special graph. Following Régin [67], let $G(\langle X, D\rangle, E)$ be an undirected bipartite graph such that nodes on the left represent variables and nodes at the right represent values. There is an edge (x_i, v) in E if and only if the value v is in the domain $\mathrm{dom}(x_i)$ of the variable. Let $c(n)$ be the capacity associated to node n such that $c(x_i) = 1$ for all variable-nodes $x_i \in X$ and $c(v)$ is an arbitrary non-negative value for all value-nodes v in D. A *generalized matching* [39, 40] M in graph G is a subset of the edges E such that no more than $c(n)$ edges in M are adjacent to node n. We are interested in finding a generalized matching M with maximal cardinality. We present a algorithm that computes this generalized matching M. Our algorithm exploites the same properties of the graph as the algorithm presented by Even and Tarjan [18]. Both algorithms have a worst running time complexity of $O(\sqrt{|X|}|E|)$.

In a generalized matching M, a *free node* is a node n adjacent to less than $c(v)$ edges in M. We presented in Section 2.2.2 the Hopcroft-Karp algorithm which computes a maximum matching in $O(\sqrt{|X|}|E|)$ steps when $c(n) = 1$ for every node n. We generalize the algorithm to obtain the same complexity when $c(v) \geq 0$ for

the value-nodes and $c(x_i) = 1$ for variable-nodes. This is precisely the case that occurs with GCC.

The Hopcroft-Karp algorithm starts with an initial empty matching $M = \emptyset$ which is improved at each iteration by finding a set of disjoint shortest augmenting paths. An iteration that finds a set of augmenting paths proceeds in two steps.

The first step consists of performing a breadth-first search (BFS) on the residual graph G_M starting with the free variable-nodes. The breadth-first search generates a forest of nodes such that nodes at level i are at distance i from a free node. This distance is minimal by construction of the BFS. Let m be the smallest level that contains a free value-node. For each node n at level $i < m$, we assign a list $L(n)$ of nodes adjacent to node n that are at level $i+1$. We set $L(n) = \emptyset$ for every node at level m or higher.

The second step of the algorithm uses a stack to perform a depth-first search (DFS). The DFS starts from a free variable-node and is only allowed to branch from a node n to a node in $L(n)$. When the algorithm branches from node n_1 to n_2, it deletes n_2 from $L(n_1)$. If the DFS reaches a free value-node, the algorithm marks this node as non-free, clears the stack, and pushes a new free variable-node that has not been visited onto the stack. This DFS generates a forest of trees whose roots are free variable-nodes. If a tree also contains a free value-node, then the path from the root to this free-value node is an augmenting path. Changing in the residual graph G_M the orientation of all edges that lie on the augmenting paths generates a matching of greater cardinality.

In our case, to find a matching when the capacities of value-nodes $c(v)$ are non-negative, we construct the duplicated graph G' where value-nodes v are duplicated $c(v)$ times and the capacity of each node is set to 1. Clearly, a matching in G' corresponds to a matching in G and can be found by the Hopcroft-Karp algorithm. We can simulate a trace of the Hopcroft-Karp algorithm run on graph G' by directly using graph G. We simply let the DFS visit $c(n) - \deg_M(n)$ times a free-node n where $\deg_M(n)$ is the number of edges in M adjacent to node n. This simulates the visit of the free duplicated nodes of node n in G. Even if we allow multiple visits of a same node, we maintain the constraint that an edge cannot be traversed more than once in the DFS. The running time complexity for a DFS is still bounded by

the number of edges $O(|X||D|)$ where $|D|$ is the number of value nodes.

Hopcroft and Karp proved that if s is the cardinality of a maximum cardinality matching, then $O(\sqrt{s})$ iterations are sufficient to find this maximum cardinality matching. In our case, s is bounded by $|X|$, and the complexity of each BFS and DFS is bounded by the number of edges in G_M i.e. $O(|X||D|)$. The total complexity is therefore $O(|X|^{1.5}|D|)$. We will run this algorithm twice, first with $c(v) = u_v$ to obtain a matching M_u and then with $c(v) = l_v$ to obtain a matching M_l.

5.6.2 Pruning the Domains

Using the algorithm described in the previous section, we compute a matching M_u in graph G such that capacities of variable-nodes are set to $c(x_i) = 1$ and capacities of value-nodes are set to $c(v) = u_v$. A matching M_u clearly corresponds to an assignment that satisfies the UBC if it has cardinality $|X|$ i.e. if each variable is assigned to a value.

Consider now the same graph G where capacities of variable-nodes are $c(x_i) = 1$ but capacities of value-nodes are set to $c(v) = l_v$. A maximum matching M_l of cardinality $|M_l| = \sum l_v$ represents a partial solution that satisfies the LBC. Variables that are not assigned to a value can in fact be assigned to any value in their domain and still satisfy the LBC.

Pruning the domains consists of finding the edges that cannot be part of a matching. From flow theory, we know that an edge can be part of a matching if and only if it belongs to a strongly connected component or lies on a path starting from or leading to a free node (see Theorem 2.3 in Section 2.2.2).

Régin's algorithm prunes the domains by finding all strongly connected components and flagging all edges that lie on a path starting or finishing at a free node. This can be done in $O(|X||D|)$ using DFS as described in [80]. As shown in Section 5.3, pruning the domains for the UBC and then pruning the domains for the LBC is sufficient to prune the domains for the GCC. Therefore, detecting edges that cannot be part of matching M_u and matching M_l is sufficient to enforce domain consistency on GCC.

5.6.3 Dynamic Case

If during the propagation process another constraint removes a value from a domain, we would like to efficiently reintroduce domain consistency over UBC and LBC. Régin [67] describes how to maintain a maximum matching under edge deletion and maintain domain consistency in $O(\delta|X||D|)$ where δ is the number of deleted edges (see Section 3.3.1). His algorithm can also be used on both graphs M_u and M_l to maintain domain consistency in $O(\delta|X||D|)$ steps.

5.7 The EXT-GCC Constraint

As mentioned in Section 3.5.3, the EXT-GCC constraint is a GCC where lower bounds l_v and upper bounds u_v are replaced by variables. The expression EXT-GCC$([x_i, \ldots, x_n], [C_1, \ldots, C_m], D)$ is satisfied if and only if for every value $v \in D$ we have $|\{i \in [1, n] \mid x_i = v\}| = C_v$. Katriel and Thiel [40] describe how to enforce bounds consistency on all variables. We show in Section 5.7.1 how to enforce domain consistency on variables x_i, $1 \leq i \leq n$ and bounds consistency on cardinality variables C_i for $1 \leq i \leq m$. We then show in Section 5.7.3 that enforcing domain consistency on all variables is NP-Hard.

5.7.1 Mixed Consistency

Pruning the cardinality variables C_v seems like a natural operation to apply to the GCC. To give a simple example, if variable $\text{dom}(C_v) = [0, 100]$ constrains the value v to be assigned to at most 100 variables while there are less than 50 variables involved in the problem, it is clear that $\text{dom}(C_v)$ can be reduced to at least interval $\text{dom}(C_v) = [0, 50]$. We will show in the next two sections how to shrink the cardinality variable domains.

Growing the Lower Bounds

Let G be the value graph where node capacities are set to $c(x_i) = 1$ for variable-nodes and $c(a) = \max(\text{dom}(C_a))$ for value-nodes. For a specific value v, we want to find the smallest value $\min(C_v)$ such that there exists a matching M of cardinality $|X|$ that satisfies the capacity constraints $\deg_M(v) = \min(\text{dom}(C_v))$.

We construct a maximum cardinality matching M_u that satisfies the capacity constraints of G. For each matched value v (i.e. for each value v such that $\deg_{M_u}(v) > 0$), we create a graph G^v and a matching M_u^v that are respectively a copy of the graph G and the matching M_u^v in which we remove all edges adjacent to value-node v. The partial matching M_u^v can be transformed into a maximum cardinality matching by repeatedly finding an augmenting path using a DFS in the residual graph and applying this path to M_u^v. This is done in $O(\deg_{M_u}(v)|X||D|)$ steps. Let $|M_u^v|$ be the cardinality of the maximum matching.

Lemma 5.11 *Let G be the value graph and G^v be the value graph where all edges adjacent to v are removed. Let M_u be a maximum matching of cardinality $|X|$ in G and let M_u^v be maximum matching in G^v. The number of edges in M_u adjacent to v must be at least $|M_u| - |M_u^v|$.*

Proof If by removing all edges connected to value-node v in graph G the cardinality of a maximum matching in G drops from $|M_u|$ to $|M_u^v|$ then at least $|M_u| - |M_u^v|$ edges in M_u were adjacent to value-node v and could not be replaced by other edges. Therefore value-node v is required to be adjacent to $|M_u| - |M_u^v|$ edges in M_u in order to obtain a matching of cardinality $|X|$. □

Since M_u is a maximum matching, we have $\sum_v \deg_{M_u}(v) = |X|$ and therefore the time required to prune all cardinality lower bounds for all values is

$$O\left(\sum_v \deg_{M_u}(v)|X||D|\right) = O(|X|^2|D|).$$

Pruning Upper Bounds

We wish to know what is the maximum number of variables that can be assigned to a value v without violating the LBC; i.e. how many variables can be assigned to value v while other values $w \in D$ are still assigned to at least $\min(\text{dom}(C_w))$ variables. We set the capacity of the value nodes to $\min(\text{dom}(C_v))$ and compute the maximum matching M_l. The cardinality of matching M_l might be less than the number of variables $|X|$. Consider the residual graph G_{M_l}. If there exists a path from a free variable-node to the value-node v then there exists a matching M_l' that has one more variable assigned to v than matching M_l and that still satisfies the LBC.

Lemma 5.12 *Given a value graph G and a maximum cardinality matching M_l, the number of edge-disjoint paths in G_{M_l} from free variable-nodes to value-node v can be computed in $O(|X|^{2.67})$ steps.*

Proof We first observe that a value-node in G_{M_l} that is not adjacent to any edge in M_l cannot reach a variable-node (by definition of a residual graph). These nodes, with the exception of node v, cannot lead to a path from a free variable-node to node v. We therefore create a graph $G^v_{M_l}$ by removing from G_{M_l} all nodes that are not adjacent to an edge in M_l except for node v. To the graph $G^v_{M_l}$, we add a special node s called the source node and we add edges from s to all free-variable nodes. Since there are at most $|X|$ matched variable-nodes, we obtain a graph of at most $2|X|+1$ nodes and $O(|X|^2)$ edges.

The number of edge-disjoint paths from the free variable-nodes to value-node v is equal to the maximum flow between s and v. A maximum flow in a directed bipartite graph where edge capacities are all one can be computed in $O(\min\{n^{\frac{2}{3}}m, m^{\frac{3}{2}}\})$ where n is the number of nodes and m the number of edges (see Theorem 8.1 in [4]). We assume that $|X| \geq |D|$, which is the case when $l_v > 0$ for all value v. Since, in the worst case, we have $n = |X| + |D|$ and $m = |X||D|$, we obtain a complexity of $O(|X|^{2.67})$. □

Observe that the maximum number of variables that can be assigned to value v is equal to the number of edges adjacent to v in M_l plus the number of edge-

disjoint paths between the free-nodes and node v. Indeed, each path p can be used to create a matching $M'_l = M_l \oplus p$ with cardinality $|M_l| + 1$ and where the number of edges adjacent to v is also greater by one. We compute the number of such edge-disjoint paths via a flow problem. This allows us to update the new upper bounds $\max(\text{dom}(C_v))$ of each value. According to Lemma 5.12, this operation can be done in $O(|D||X|^{2.67})$ steps.

5.7.2 Bounding the Cardinality Variables

As shown in previous sections, pruning the cardinality variables can be an expensive task requiring as many as $O(|D||X|^{2.67})$ steps in the worst case. We show in this section a special case that allows us to detect in time $O(|X||D|)$ when the cardinality variables l_v and u_v should be pruned to the same value ($l_v = u_v$).

Consider the residual graph G_{M_u}. If there is no path from value-node n leading to a free node then it is impossible to construct a maximum cardinality matching that has one less edge adjacent to value-node n. Therefore we can set $l'_v = u_v$.

The same applies to the residual graph G_{M_l}: if no free node can reach a value-node n, then all maximum matchings must have l_v edges assigned to value v and therefore we can set $u'_v = l_v$.

In both cases, we can determine values whose cardinality variables l_v and u_v can be bounded to the same value using a simple DFS that only requires $O(|X||D|)$ steps. This technique is equivalent and has the same running time complexity as Régin's technique for the Cardinality-Matrix constraint [72].

5.7.3 Domain Consistency is NP-Hard

Recall that the constraint EXT-GCC($[x_1, \ldots, x_n], [K_1, \ldots, K_m], D$) is satisfied if and only if for each value $v \in D$, the number of variables assigned to v is equal to K_v. We show that this problem is equivalent to finding a generalized matching in a certain bipartite graph G. The variables form the left-nodes and the values the right-nodes in G. There is an edge between node x_i and value v if $v \in \text{dom}(x_i)$. A

generalized matching M is a subset of the edges of G such that each left-node is adjacent to exactly one edge in M and each value-node v is adjacent to k edges in M such that $k \in \text{dom}(K_v)$. Clearly, there is a one-to-one correspondence between finding a solution satisfying the EXT-GCC and finding a generalized matching. We now prove that finding such a generalized matching is NP-Hard by reduction to the SAT problem.

Consider a 3-SAT problem with a list of variables $X = \{X_1, \dots X_n\}$, a list of literals $\mathcal{L} = \{x_i, \neg x_i \mid X_i \in X\}$ and a list of clauses $C = \{C_1, \dots C_m\}$ where $C_i \subseteq \mathcal{L}$ are the set of literals of the clause. We want to assign the value *true* or *false* to the literals in $\mathcal{L}$ such that all clauses have at least one literal assigned to *true*.

From the SAT problem, we construct the bipartite graph $G = \langle L \cup R, E \rangle$ as follows. For each literal l_j in a clause C_i, we create one left-node $S(C_i, l_j) \in L$ and one right-node $d(C_i, l_j) \in R$. For each clause C_i we create a left-node $C_i \in L$ and for each variable X_i we create another left-node $X_i \in L$. Finally, we add to the graph a right-node $l_i \in R$ for each literal l_i.

We connect the left-nodes in L to the right-nodes in R as follows. We start with an empty set of edges $E = \emptyset$. For each clause C_i and each literal $l_j \in C_i$, we add the edges $(C_i, d(C_i, l_j))$, $(S(C_i, l_j), d(C_i, l_j))$ and $(S(C_i, l_j), l_j)$. For each variable $x_i \in X$ we add the edges (X_i, x_i) and $(X_i, \neg x_i)$. Finally, we add two values in the domain of the K variables as follows: $\text{dom}(K_{d(C_i,l_j)}) = \{0, 1\}$ and $\text{dom}(K_{l_i}) = \{0, k_i + 1\}$ where k_i is equal to the number of clauses containing the literal l_i or more formally $k_i = |\{C_j \in C \mid l_i \in C_j\}|$. Figure 5.3 shows the part of graph G that is related to variable X_i.

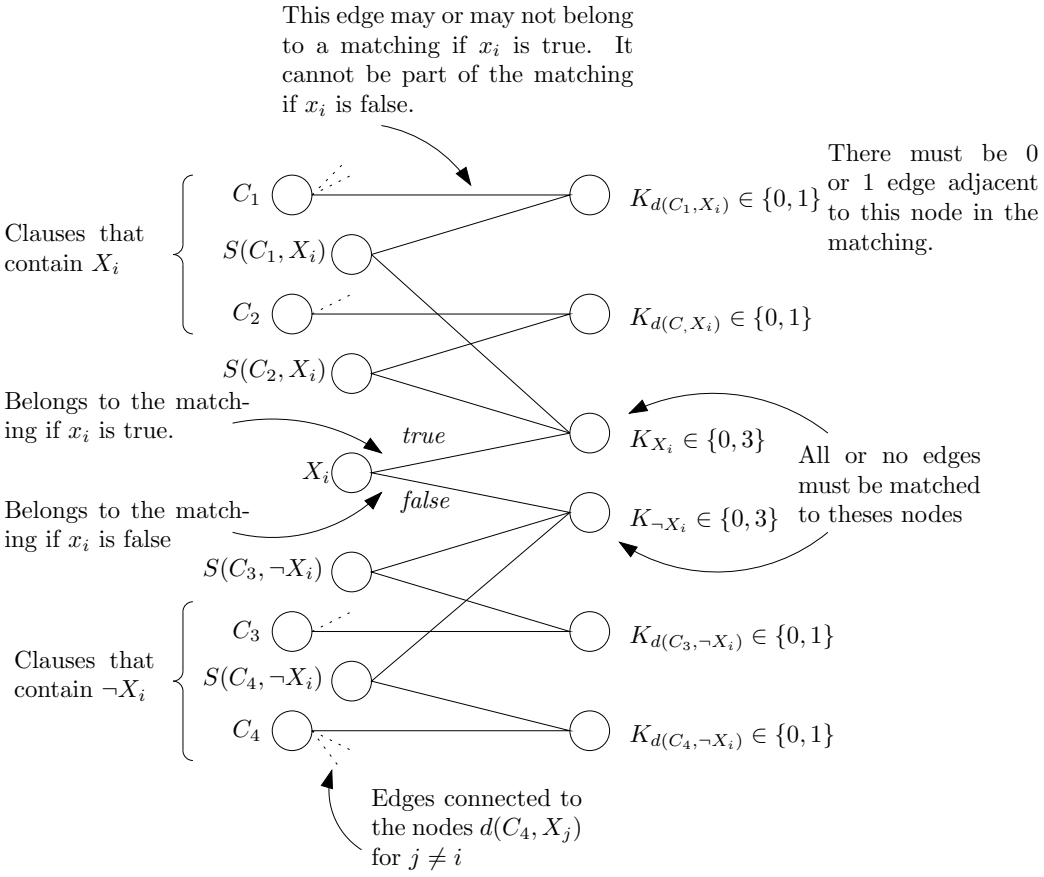

Figure 5.3: Part of graph G related to variable X_i.

The intuition of the reduction is simple. A generalized matching in G corresponds to a solution to the SAT problem. If $(X_i, x_i) \in M$ then $x_i = \mathit{true}$ and if $(X_i, \neg x_i) \in M$ then $x_i = \mathit{false}$. All clause nodes C_i must be matched to another node. They can only be matched with an edge $(C_i, d(C_i, l_j))$ if $l_j = \mathit{true}$.

Lemma 5.13 *Let G be the graph described above and let l_i be a literal in $\{x_i, \neg x_i\}$. The edge (X_i, l_i) belongs to a generalized matching M in graph G if and only if the edge $(S(C_j, l_i), l_i)$ belongs to M for all clauses C_j.*

Proof The nodes $S(C_j, l_i) \in E$ and the node X_i are the only nodes connected to node l_i. Since we have $K(l_i) = \{0, k_i + 1\}$ and $k_i + 1$ is equal to the number of nodes connected to l_i, either all edges adjacent to l_i belong to M or no edges adjacent to l_i belong to M. Therefore for all nodes $S(C_j, l_i)$ we have $(X_i, l_i) \in M \iff S(C_j, l_i) \in M$. □

Lemma 5.14 *Let G be the graph described above and let l_j be a literal in $\{x_j, \neg x_j\}$. If the edge $(C_i, d(C_i, l_j))$ belongs to a generalized matching M of the graph G then (X_j, l_j) also belongs to this generalized matching.*

Proof Suppose the edge $(C_i, d(C_i, l_j))$ belongs to the generalized matching M. Since the cardinality of node $d(C_i, l_j)$ is $\{0, 1\}$ and edge $(C_i, d(C_i, l_j))$ is adjacent to this node, no more edges in M can be adjacent to node $(C_i, d(C_i, l_j))$. Therefore node $S(C_i, l_j)$ has no other choice to be matched with node l_j. By Lemma 5.13 we obtain that (X_j, l_j) belongs to M. □

Lemma 5.15 *The SAT instance is satisfiable if and only if there exists a generalized matching M in the corresponding graph G.*

Proof ($\Rightarrow$) Suppose SAT is satisfiable, we construct a matching by pointing each node C_i to a node $d(C_i, l_i)$ such that literal l_i is true in the SAT solution. Other left-nodes in L are matched according to Lemma 5.14 and Lemma 5.13.

($\Leftarrow$) Consider a generalized matching M. For all variables $X_i \in X$, we have either the edge (X_i, x_i) or $(X_i, \neg x_i)$ in M. We say that literal l_i is true if the edge (X_i, l_i) belongs to M and false if the edge does not belong to M. For all clauses

C_i, we have an edge $(C_i, d(C_i, l_j))$ in M for some $l_j \in \{x_j, \neg x_j\}$. This implies by Lemma 5.14 that l_j is true and therefore clause C_i is satisfied. Therefore all clauses are satisfied by the variable assignments given by the edges (X_i, l_i) □

Lemma 5.15 shows that determining the satisfiability of extended-GCC is NP-complete and therefore enforcing domain consistency on the extended-GCC is NP-hard.

5.8 Universality

At some point in the search process, a constraint can become *universal*. In other words, this constraint no longer constrains the search space.

Definition 5.10 A constraint C is universal for a problem if any tuple $t \in \text{dom}(x_1) \times \ldots \times \text{dom}(x_n)$ satisfies the constraint C.

We study under what conditions a given GCC behaves like the universal constraint. We give an algorithm that tests in constant time if the LBC or the UBC are universal. Recall that the set of solutions satisfying the GCC is the intersection of the set of solutions satisfying the LBC and the set of solutions satisfying the UBC. If both the LBC and the UBC are universal, then the GCC is universal. This implies there is no need to run a propagator on the GCC since we know that all values have a support. In other words, as far as the GCC is concerned, every assignment from now on is valid. The universality test speeds up the propagation by avoiding unnecessary calls to the GCC propagator. Our result holds for domain, range, and bounds consistency.

5.8.1 Universality of the Lower Bound Constraint

Lemma 5.16 *The* LBC *is universal for a problem if and only if for each value $v \in D$ there exists at least l_v variables x such that $dom(x) = \{v\}$.*

Proof ($\Longleftarrow$) If for each value $v \in D$ there are l_v variables x such that $\text{dom}(x) = \{v\}$ then it is clear that any variable assignment satisfies the LBC.

($\Longrightarrow$) Suppose for a LBC problem there is a value $v \in D$ such that there are less than l_v variables whose domain only contains value v. Therefore, an assignment where all variables that are not bounded to v are assigned to a value other than v would not satisfy the LBC. This proves that LBC is not universal under this assumption. □

The following algorithm verifies if the LBC is universal in $O(|X| + |D|)$ steps:

1. Create a vector t such that $t[v] = l_v$ for all $v \in D$.
2. For all domains that contain only one value v, decrement $t[v]$ by one.
3. The LBC is universal if and only if no elements in t are positive.

We can easily make the algorithm dynamic under the modification of variable domains. We keep a counter c that indicates the number of positive elements in vector t. Each time a variable gets bounded to a single value v, we decrement $t[v]$ by one. If $t[v]$ reaches the value zero, we decrement c by one. The LBC becomes universal when c reaches zero. Using this strategy, each time a variable domain is pruned, we can check in constant time if the LBC becomes universal.

5.8.2 Universality of the Upper Bound Constraint

Lemma 5.17 *The* UBC *is universal for a problem if and only if for each value $v \in D$ there exists at most u_v variable domains that contain v.*

Proof ($\Longleftarrow$) Trivially, if for each value $v \in D$ there are u_v or fewer variable domains that contain v, there is no assignment that could violate the UBC and therefore the UBC is universal.

($\Longrightarrow$) Suppose there is a value v such that more than u_v variable domains contain v. If we assign all these variables to the value v, we obtain an assignment that does not satisfy the UBC. □

To test the universality of the UBC, we create a vector a such that $a[v] = I(\{v\}) - u_v$. According to Lemma 5.17, the UBC is universal if and only if no elements of a are positive. Such a vector might be slow to update when variable domains change. In order to perform faster update operations, we represent the vector a by a vector t whose first element is equal to $-u_{\min(D)}$ and whose following elements is given by the difference between u_{v-1} and u_v. More formally, we initialize t as follows: $t[\min(D)] \leftarrow -u_{\min(D)}$ and $t[v] \leftarrow u_{v-1} - u_v$ for $\min(D) < v \leq \max(D)$. We first assume that variable domains are initially intervals. This restriction will later be removed. For each variable $x_i \in X$, we increment the value of $t[\min(\mathrm{dom}(x_i))]$ by one and decrement $t[\max(\mathrm{dom}(x_i)) + 1]$ by one. Let i be an index initialized to value $\min(D)$. At all time, we maintain the following invariant.

$$a[v] = I(\{v\}) - u_v = \sum_{j=i}^{v} t[j] \tag{5.5}$$

Index i divides the domain of values D in two sets: the values v smaller than i which are not contained in more than u_v variable domains and other values which can be contained in any number of variable domains. We maintain the index i to hold the highest possible value that satisfies this invariant. If index i reaches a value greater than $\max(D)$ then all values v in D are contained in less than u_v variable domains and therefore the UBC is universal. Algorithm 11 increases index i to the first value v that is contained in more than u_v domains. The algorithm also updates vector t such that Equation 5.5 is verified for all values greater than or equal to i.

Suppose a variable domain gets pruned in such a way that all values in interval $[a, b]$ are removed. To maintain the invariant given by Equation 5.5 for values greater than or equal to i, we update our vector t by removing 1 from element $t[\max(a, i)]$ and adding one to element $t[\max(b + 1, i)]$. We then run Algorithm 11 to increase index i. If $i > \max(D)$ then the UBC is universal since no value is contained in more domains than its maximal capacity.

Initializing vector t and increasing iterator i until $i > \max(D)$ requires $O(|X| + |D|)$ steps. Updating the data structure when a variable domain changes requires

Algorithm 11: Algorithm used for testing the universality of the UBC. It increases index i to the smallest value $v \in D$ contained in more than u_v domains. The algorithm also modifies vector t to validate Equation 5.5 when $v \geq i$.

```
while (i ≤ max(D)) and (t[i] ≤ 0) do
    i ← i + 1 ;
    if i ≤ max(D) then
        t[i] ← t[i] + t[i − 1];
```

constant time plus the time to move index i. Index i can only be increased until it reaches value $\max(D)$. Therefore, checking universality each time an interval of values is removed from a variable domain is achieved in amortized constant time.

5.9 The Global Cardinality Constraint on Non Integer Variables

In Section 4.4, we show how the ALL-DIFFERENT constraint propagator can be adapted for non integer variables such as set, multiset, or tuple variables. The GCC can be adapted for such variables as well.

When dealing with GCC on integer variables, we express the lower capacity and the upper capacity of a value v with the constants l_v and u_v that are in fact entries in vector l and u. In turn, when working with large domains, these look-up tables could require too much memory. We therefore assume that the lower and upper capacity of each value are given by a function instead of a constraint. For instance, the constant functions $\lfloor v \rfloor = 0$ and $\lceil v \rceil = 1$ define the ALL-DIFFERENT constraint. In order to be feasible, the following restrictions apply: $\sum_v \lfloor v \rfloor \leq n$ and $\sum_v \lceil v \rceil \geq n$. For efficiency reasons, we assume that the values L whose lower capacity is positive are known, i.e. $L = \{v \mid \lfloor v \rfloor > 0\}$ is known.

We need an equivalent to Lemma 4.6 to determine which variables have a *large domain*. The following Lemma provides the required result for the GCC.

Lemma 5.18 *Let F be a set of variables whose domains are not contained in any Hall set and assume $\lceil v \rceil \geq k$ holds for all value v. If $x_i \notin F$ is a variable whose domain contains more than $\lfloor \frac{n-|F|}{k} \rfloor$ values, then dom(x_i) is not contained in any Hall set.*

Proof The largest Hall set can contain the domain of at most $n-|F|$ variables and therefore has at most $\lfloor \frac{n-|F|}{k} \rfloor$ values. If $|\text{dom}(x_i)| > \lfloor \frac{n-|F|}{k} \rfloor$, then dom$(x_i)$ cannot be contained in any Hall set. □

As we did before, we divide the GCC into two constraints: the lower bound constraint (LBC) and the upper bound constraint (UBC).

The upper bound constraint is similar to the ALL-DIFFERENT constraint. Up to $\lceil v \rceil$ variables can be assigned to a value v instead of only 1 with the ALL-DIFFERENT constraint. Lemma 5.18 suggests to modify Line 1 of Algorithm 5 by testing if $|\text{dom}(x_i)| > \frac{|X|-|F|}{k}$ before inserting variable x_i in set F instead of testing $|\text{dom}(x_i)| > n-|F|$. The rest of the propagation for the UBC is the same as the ALL-DIFFERENT constraint.

The lower bound constraint can also be easily handled when variable domains are large. Consider the set L of values whose lower capacity is positive, i.e. $L = \{v \mid \lfloor v \rfloor > 0\}$. In order for the lower bound constraint to be satisfiable over n variables, the cardinality of L must be bounded by n. The values not in L can be assigned to a variable only if all values v in L have been assigned to at least $\lfloor v \rfloor$ variables. Since all values not in L are symmetric, we can replace them by a single value p such that $\lfloor p \rfloor = 0$. We now obtain a problem where each variable domain is bounded by $n+1$ values. We can apply a propagator for the lower bound constraint on this new problem. Notice that if the lower bound constraint propagator removes p from a variable domain, it implies by symmetry that all values not in L should be removed from this variable domain. Therefore, we can enforce domain consistency on the GCC in $O(|X|^{2.5} + E)$ time where E is the time necessary to enumerate the variable domains. Time E depends on the nature of the domains, whether they are sets, tuples, multi-sets, include cardinality constraints, or lexicographical constraints as discussed in Sections 4.4.2, 4.4.3, and 4.4.4.

Chapter 6

The Inter-Distance Constraint

6.1 Introduction

The *cumulative scheduling problem* with one resource of capacity C consists of a set of tasks $T_1, \ldots, T_n$ to which we associate four integer variables: a release time r_i, a deadline d_i, a processing time p_i and a capacity requirement c_i. Each task T_i must start at time t_i such that $r_i \leq t_i \leq d_i - p_i$. Let $\Omega(t)$ be the set of tasks in process at time t, i.e. the tasks T_i such that $t_i \leq t \leq t_i + p_i$. We have the resource capacity constraint $\sum_{T_i \in \Omega(t)} c_i \leq C$. This problem is NP-Hard [22] even in the case where $C = 1$ which we call, in this particular case, the *disjunctive scheduling problem*.

Edge finders [13, 55] have largely been used to solve scheduling problems. This technique reduces the intervals $[r_i, d_i]$ by detecting time zones that must be allocated to a subset of the tasks making these zones unavailable for other tasks. The goal is to increase release times and reduce deadlines without eliminating any feasible solution. The problem is said to be *bounds consistent* when intervals $[r_i, d_i]$ have been fully shrunk, i.e. when there exists at least one feasible schedule in which task T_i starts on time r_i and at least one feasible schedule in which task T_i finishes on time d_i. It is NP-Hard to make a scheduling problem bounds consistent, even in the disjunctive case. For this reason, edge finders try to reduce, in polynomial time, the size of the intervals without necessarily achieving bounds consistency.

A backtracking search assigns starting times to tasks and uses the edge finder to reduce the size of the search tree.

We study the disjunctive scheduling problem when all tasks have the same processing time $p_i = p$. This problem can be solved in polynomial time [23] but traditional algorithms only return one solution that generally does not satisfy the side constraints. These side constraints can even make the problem NP-Hard. Constraint programming can be used to encode such problems. A single Inter-Distance constraint can encode the disjunctive scheduling problem. This constraint ensures that starting times are pairwise distant by at least p units of time.

Artiouchine and Baptiste [5] recently proposed an $O(n^3)$ propagator that enforces bounds consistency on the Inter-Distance constraint. By achieving bounds consistency, their propagator prunes the search space better than edge finding algorithms resulting in smaller choice points in the backtracking search and an improved time performance. We propose in this work a quadratic propagator that is faster both in theory and in practice, even for small instances.

Throughout this chapter, we will consider the set $[a, b]$ as the interval of integer values between a and b inclusively. If $a > b$, then the interval $[a, b]$ is empty. We nevertheless say that the lower bound of the interval is $\min([a, b]) = a$ and the upper bound is $\max([a, b]) = b$ as for non-empty intervals.

We first present some notions about how bounds consistency can be enforced on the Inter-Distance constraint. We then explain how the computation can be simplified. Based on this simplification, we present our algorithm and a data structure that ensures the quadratic behaviour of our propagator. Finally, we present some experiments proving the efficiency of our propagator.

6.2 The Inter-Distance Constraint

Régin [69] first introduced the Inter-Distance constraint. In this constraint, the expression Inter-Distance$([X_1, \ldots, X_n], p)$ holds if and only if $|X_i - X_j| \geq p$ for all $i \neq j$. When $p = 1$, the Inter-Distance specializes into an All-Different

constraint [67, 54, 51]. Régin [67] showed that a single global constraint, in many cases, causes more domain reductions than the $\frac{n(n-1)}{2}$ equivalent binary constraints. This observation also applies to the INTER-DISTANCE constraint which is the general case. Artiouchine and Baptiste provided the first propagator for bounds consistency of the INTER-DISTANCE constraint. The running time complexity of this propagator is $O(n^3)$.

We use the following problem as a running example.

Example 6.1 Consider a problem with $n = 3$ tasks T_1, T_2, and T_3 with processing time $p = 6$ and the following release times and deadlines.

$$\begin{array}{lll} r_1 = 2 & r_2 = 10 & r_3 = 4 \\ d_1 = 12 & d_2 = 20 & d_3 = 21 \end{array}$$

After propagating the constraint INTER-DISTANCE$([T_1, T_2, T_3], p)$, we obtain the following release times and deadlines.

$$\begin{array}{lll} r_1 = 2 & r_2 = 14 & r_3 = 8 \\ d_1 = 8 & d_2 = 20 & d_3 = 14 \end{array}$$

Here, task T_1 must finish before or at time 8 in order to allow tasks T_2 and T_3 to meet their deadlines. Task T_2 cannot start before time 14 since the two other tasks are not completed before this time. Finally, task T_3 must be executed between tasks T_1 and T_2 forcing its release time to be increased and its deadline to be reduced.

Garey et al. [23] designed an algorithm that finds a solution satisfying the INTER-DISTANCE constraint in $O(n \log n)$ steps. Their algorithm proceeds in two phases. In the first phase, the algorithm computes a set of regions F in which no tasks are allowed to start. We call these regions the *forbidden regions*. Their number is bounded by the number of tasks n. Once these forbidden regions are computed, the second phase uses a greedy strategy to schedule the tasks.

Artiouchine and Baptiste as well as Garey et al. use two basic functions as main pillars of their algorithm. Let $ect(F, r, q)$ be the *earliest completion time* of

a schedule of q tasks starting at or after time r with no task starting within a forbidden region in the set of forbidden regions F. Symmetrically, let $lst(F, d, q)$ be the *latest starting time* of a schedule of q tasks finishing at or before time d with no task ever starting in a forbidden region in F. Computing $ect(F, r, q)$ and $lst(F, d, q)$ can be done in $O(q)$ steps using the following recurrences where $ect(F, r, 0) = r$ and $lst(F, d, 0) = d$.

$$ect(F, r, q) = \min\{t \notin F \mid t \geq ect(F, r, q-1)\} + p \quad (6.1)$$

$$lst(F, d, q) = \max\{t \notin F \mid t \leq lst(F, d, q-1) - p\} \quad (6.2)$$

Using these two functions, Artiouchine and Baptiste describe two types of adjustment intervals necessary and sufficient to maintain bounds consistency on the INTER-DISTANCE constraint. We first define $\Delta(r, d)$ to be the set of tasks whose release times and deadlines are contained in the interval $[r, d]$. An *internal adjustment interval* is an interval in which no task is allowed to start. The set of internal adjustment intervals is a superset of the forbidden regions F. Theorem 6.1 formally characterizes the internal adjustment intervals. Intuitively, the more there are tasks that need to be scheduled between time r and time d, the more the internal adjustment intervals will be large. We define the following internal adjustment intervals.

$$I_{r,d,q} = [lst(, d, q+1) + 1, ect(F, r, |\Delta(r, d)| - q) - 1] \quad (6.3)$$

Theorem 6.1 (Artiouchine and Baptiste [5]) *Given two time points r, d, and an integer $0 \leq q < |\Delta(r, d)|$, no task can start in the interval $I_{r,d,q}$ with endpoints $lst(F, d, q+1) + 1$ and $ect(F, r, |\Delta(r, d)| - q) - 1$.*

Proof Suppose that job i starts at time $t < ect(F, r, |\Delta(r, d)| - q)$ in a feasible schedule for some $0 \leq q < |\Delta(r, d)|$. According to the definition of ect, at most $|\Delta(r, d)| - q - 1$ jobs can finish at or before t. Therefore at least $q + 1$ jobs must start after time t. Consequently $t \leq lst(F, d, q+1)$. Therefore, no tasks can start in the interval $[lst(F, d, q+1) + 1, ect(F, r, |\Delta(r, d)| - q) - 1]$. □

The *external adjustment intervals* are intervals in which a subset of the tasks are not allowed to start. They are formally characterized in Equation 6.4.

$$E_{r,d,q} \quad = \quad [lst(, d, q+2)+1, ect(F, r, |\Delta(r,d)|-q)-1] \tag{6.4}$$

Theorem 6.2 (Artiouchine and Baptiste[5]) *Given two time points r, d and an integer $0 \leq q < |\Delta(r,d)|$, a task $i \notin \Delta(r,d)$ cannot start in the interval $E_{r,d,q}$ with endpoints $lst(F,d,q+2)+1$ and $ect(F,r,|\Delta(r,d)|-q)-1$.*

Proof Suppose that job $i \notin \Delta(r,d)$ starts at time $t < ect(F,r,|\Delta(r,d)|-q)$ in a feasible schedule for some $0 \leq q < |\Delta(r,d)|$. According to the definition of *ect*, at most $|\Delta(r,d)|-q-1$ jobs finish before or at time t. Therefore at least $q+1$ jobs, in addition to job i, finish after t which implies $t \leq lst(F,d,q+2)$. Therefore, jobs not in $\Delta(r,d)$ cannot start in the interval $[lst(F,d,q+2)+1, ect(F,r,|\Delta(r,d)|-q)-1]$. □

Notice that the external adjustment intervals form a superset of the internal adjustment intervals. This forces the tasks that can be scheduled outside the time interval $[r,d]$ to leave some time zones free for the tasks that must be scheduled inside the interval $[r,d]$.

Table 6.1 shows the internal and external adjustment intervals from Example 6.1.

Artiouchine and Baptiste formally proved that the internal and external adjustment intervals are necessary and sufficient to enforce bounds consistency on the INTER-DISTANCE constraint.

6.3 Towards a Quadratic Propagator

Internal and external adjustment intervals in the worst case may be computed with up to n possible release times r, n possible deadlines d and produce $O(n)$ adjustment intervals each. Therefore, $O(n^3)$ adjustment intervals could be checked in the worst case, hence the cubic time complexity of the Artiouchine-Baptiste propagator.

Internal Adjustment Intervals			
$r_i \backslash d_j$	12	20	21
2	$\{[7,7]\}$	$\{[9,7],[15,13]\}$	$\{[3,7],[9,13],[16,19]\}$
4	$\emptyset$	$\{[15,9]\}$	$\{[9,9],[16,15]\}$
10	$\emptyset$	$\{[15,15]\}$	$\{[16,15]\}$
External Adjustment Intervals			
$r_i \backslash d_j$	12	20	21
2	$\{[-3,7]\}$	$\{[3,7],[9,13]\}$	$\{[-3,7],[3,13],[9,19]\}$
4	$\emptyset$	$\{[9,9]\}$	$\{[3,9],[9,15]\}$
10	$\emptyset$	$\{[9,15]\}$	$\{[9,15]\}$

Table 6.1: Internal and external adjustment intervals generated by a pair of time points (r_i, d_j) from Example 6.1. Intervals are written in decreasing order with respect to parameter q. The forbidden regions are $F = \{[-3,1],[3,3],[9,9]\}$.

In reality, the union of all internal and external adjustment intervals consists of a maximum of $O(n^2)$ disjoint intervals. It is therefore possible to ignore intervals that are subsets of already discovered intervals in order to achieve a quadratic complexity. To avoid computing redundant adjustment intervals, we introduce the notion of dominance between two pairs of time points. When a pair of time points dominates another pair, the adjustment regions of the dominant pair contain some adjustment regions of the other pair.

Definition 6.1 (Dominance) A pair of time points (r_i, d_j) dominates a pair of time points (r_k, d_l) if we have $\min(I_{r_i,d_j,q}) \leq \min(I_{r_k,d_l,q})$ and $\max(I_{r_i,d_j,q}) \geq \max(I_{r_k,d_l,q})$ for all $0 \leq q < \min(|\Delta(r_i,d_j)|, |\Delta(r_k,d_l)|)$. We write $(r_i,d_j) \succ (r_k,d_l)$.

Notice that we usually have $|\Delta(r_i,d_j)| \neq |\Delta(r_k,d_l)|$. The definition of dominance only applies for q below $\min(|\Delta(r_i,d_j)|, |\Delta(r_k,d_l)|)$. Also, for a fixed deadline d, the dominance operator $(\prec)$ is transitive, i.e. if $(r_i,d) \prec (r_j,d)$ and $(r_j,d) \prec (r_k,d)$ hold, then $(r_i,d) \prec (r_k,d)$ holds. In Example 6.1 we have $(2,21) \succ (4,21)$.

The following lemmas describe a property of the *ect* and *lst* functions that will allow us to efficiently decide if a pair of time points dominates another one.

Lemma 6.1 *If* $ect(F, r_i, q_1) \leq ect(F, r_j, q_2)$ *then* $ect(F, r_i, q_1+k) \leq ect(F, r_j, q_2+k)$ *for any* k, q_1, $q_2 \geq 0$.

Proof The proof is by induction on k. The base case $k = 0$ is trivial. We prove for $k = 1$. We know that $ect(F, r_i, q_1) \leq ect(F, r_j, q_2)$. We have $ect(F, r_i, q_1 + 1) = ect(F, r_i, q_1) + p + s_i$ where s_i is a (potentially null) shift caused by the (potentially empty) forbidden region $F_i = [ect(F, r_i, q_1), ect(F, r_i, q_1) + s_i] \subseteq F$. Similarly we have $ect(F, r_j, q_2 + 1) = ect(F, r_j, q_2) + p + s_j$ where s_j is the shift caused by the forbidden region $F_j = [ect(F, r_j, q_2), ect(F, r_j, q_2) + s_j] \subseteq F$. If s_i is large enough to obtain $ect(F, r_i, q_1 + 1) \geq ect(F, r_j, q_2 + 1)$, then we have $F_j \subseteq F_i$. Since both forbidden regions intersect, both functions are shifted to the same value and we obtain $ect(F, r_i, q_1 + 1) = ect(F, r_j, q_2 + 1)$ which completes the case for $k = 1$.

For the induction step, suppose that the lemma holds for $k - 1$. We have $ect(F, r_i, q_1 + k) = ect(F, r_i, q_1 + k - 1) + p + s_i$ where s_i is a (potentially null) shift caused by the (potentially empty) forbidden region $F_i = [ect(F, r_i, q_1 + k - 1), ect(F, r_i, q_1+k-1)+s_i] \subseteq F$. Similarly we have $ect(F, r_j, q_2+k) = ect(F, r_j, q_2+k-1)+p+s_j$ where s_j is the shift caused by the forbidden region $F_j = [ect(F, r_j, q_2+k-1), ect(F, r_j, q_2+k-1)+s_j] \subseteq F$. If s_i is large enough to obtain $ect(F, r_i, q_1+k) \geq ect(F, r_j, q_2 + k)$, then we have $F_j \subseteq F_i$. Since both forbidden regions intersect, both functions are shifted to the same value and we obtain $ect(F, r_i, q_1 + k) = ect(F, r_j, q_2 + k)$ which completes the induction step. □

Lemma 6.2 *If* $lst(F, d_i, q_1) \leq lst(F, d_j, q_2)$ *then* $lst(F, d_i, q_1+k) \leq lst(F, d_j, q_2+k)$ *for any* k, q_1, $q_2 \geq 0$.

Proof Symmetric to the proof of Lemma 6.1. □

We now describe three different situations in which a pair of time points dominates another one. The first case is described in Lemma 6.3.

Lemma 6.3 *Let* (r, d_i) *and* (r, d_j) *be the two pairs of time points such that* $d_i < d_j$ *and* $k = |\Delta(r, d_i)| = |\Delta(r, d_j)|$. *Then* $(r, d_i) \succ (r, d_j)$.

Proof We have $lst(F, d_i, 0) < lst(F, d_j, 0)$ and by Lemma 6.2, $lst(F, d_i, q+1) \leq lst(F, d_j, q+1)$. This implies $\min(I_{r,d_i,q}) \leq \min(I_{r,d_j,q})$ for any $0 \leq q < k$ and since we have $\max(I_{r,d_i,q}) = \max(I_{r,d_j,q})$ we have $(r, d_i) \succ (r, d_j)$. □

From Lemma 6.3 we conclude that $(10, 20) \succ (10, 21)$ in Example 6.1. Similarly, we have the following Lemma.

Lemma 6.4 *Let (r_i, d) and (r_j, d) be two the pairs of time points such that $r_i < r_j$ and $k = |\Delta(r_i, d)| = |\Delta(r_j, d)|$. Then $(r_i, d) \prec (r_j, d)$.*

Proof We have $ect(F, r_i, 0) < ect(F, r_j, 0)$ and by Lemma 6.1 $ect(F, r_i, k-q) \leq lst(F, r_j, k-q)$. This implies $\max(I_{r_i,d,q}) \leq \max(I_{r_j,d,q})$ for any $0 \leq q < k$ and since we have $\min(I_{r_i,d,q}) = \min(I_{r_j,d,q})$ we have $(r_i, d) \prec (r_j, d)$. □

In Example 6.1, Lemma 6.4 detects $(4, 20) \prec (10, 20)$. We show a last case where a pair of time points dominates another one.

Lemma 6.5 *Let (r_i, d) and (r_j, d) be two pairs of time points such that $|\Delta(r_i, d)| = |\Delta(r_j, d)| + k$ and $ect(F, r_i, k) \leq ect(F, r_j, 0)$. Then $(r_j, d) \succ (r_i, d)$.*

Proof Clearly, for $0 \leq q < |\Delta(r_j, d)|$, the internal adjustment intervals $I_{r_i,d,q}$ and $I_{r_j,d,q}$ share the same lower bound. For the upper bounds, we have the following:

$$\begin{aligned}
\max(I_{r_i,d,q}) &= ect(F, r_i, |\Delta(r_i, d)| - q) - 1 \\
&= ect(F, r_i, |\Delta(r_j, d)| + k - q)) - 1 \text{ and by Lemma 6.1} \\
&\leq ect(F, r_j, |\Delta(r_j, d)| - q) - 1 \\
&\leq \max(I_{r_j,d,q})
\end{aligned}$$

Therefore we have $(r_j, d) \succ (r_i, d)$. □

In Example 6.1, we have $(10, 20) \succ (2, 20)$ from Lemma 6.5. There might be other conditions under which we can conclude that a pair of time points is tighter than another one but this remains an open question. The cases stated in Lemma 6.3, 6.4, and 6.5 are sufficient for our purposes.

Lemma 6.5 is crucial to obtaining a quadratic algorithm. Consider a deadline d and a sequence of release times $r_1 < r_2 < \ldots < r_k$ such that $(r_1, d) \prec (r_2, d) \prec \ldots \prec (r_k, d)$. There can be up to $O(n^2)$ internal adjustment intervals associated to these pairs of time points. Nevertheless, the union of all $O(n^2)$ intervals can be given by the union of only $O(n)$ intervals. We first notice that the following intervals all share the same lower bound. The union of the intervals is therefore equal to the interval whose upper bound is the greatest.

$$\begin{aligned}\bigcup_{i=1}^{j} I_{r_i,d,q} &= [\min(I_{r_j,d,q}), \max_{1\leq i\leq j} \max(I_{r_i,d,q})] \\ &= [\min(I_{r_j,d,q}), \max(I_{r_j,d,q})] \\ &= I_{r_j,d,q}\end{aligned}$$

Using this observation, we compute the union of all adjustment intervals formed by the pairs $(r_1, d), \ldots, (r_k, d)$ using the following equation (to simplify notation, we let $|\Delta(r_{k+1}, d)| = 0$ since r_{k+1} is undefined).

$$\bigcup_{i=1}^{k} \bigcup_{q=0}^{|\Delta(r_i,d)|-1} I_{r_i,d,q} = \bigcup_{i=1}^{k} \bigcup_{q=|\Delta(r_{i+1},d)|}^{|\Delta(r_i,d)|-1} I_{r_i,d,q} \tag{6.5}$$

Notice that the left hand side of Equation 6.5 has $O(n^2)$ intervals while the right hand side has only $O(n)$ intervals. Indeed, the number of intervals to be united is given by $\sum_{i=1}^{k}(|\Delta(r_i, d)| - |\Delta(r_{i+1}, d)|)$. The telescopic series simplifies to $|\Delta(r_1, d)| - |\Delta(r_{k+1}, d)| = |\Delta(r_1, d)|$.

In Example 6.1 since we have $(2, 20) \prec (10, 20)$ we obtain the following:

$$\begin{aligned}(I_{2,20,0} \cup I_{2,20,1}) \cup (I_{10,20,0}) &= I_{2,20,1} \cup I_{10,20,0} \\ &= [9, 7] \cup [15, 15] \\ &= [15, 15]\end{aligned}$$

6.4 A Quadratic Propagator

6.4.1 General Scheme

The idea behind the algorithm is the following. We process each deadline in increasing order. If two deadlines d_i and d_j are equal and their associated release times satisfy $r_j \leq r_i$, we process both deadlines at the same time but use i as a reference. For every deadline d_i, we compute the longest sequence of release times $r_{x_1} < r_{x_2} < \ldots < r_{x_k}$ such that $(r_{x_1}, d_i) \prec (r_{x_2}, d_i) \prec \ldots \prec (r_{x_k}, d_i)$. Using this sequence and Equation 6.5, we compute the union of all internal adjustment intervals generated by the pairs of time points whose deadline is d_i. To build the sequence, we iterate through all release times in non-decreasing order. Two sub-cases can occur where we can safely skip a release time r_j.

Case 1 ($d_j > d_i$): Suppose that the deadline d_j associated to r_j has not been processed yet, i.e. $d_j > d_i$. For such a release time r_j, we choose the smallest release time r_k whose deadline has already been processed and such that $r_k > r_j$. Two cases can occur: if such a r_k exists, then $|\Delta(r_j, d_i)| = |\Delta(r_k, d_i)|$ since both intervals $[r_j, d_i]$ and $[r_k, d_i]$ contain the same processed variables and no unprocessed variables. Lemma 6.4 ensures that $(r_k, d_i) \succ (r_j, d_i)$. All adjustment intervals from (r_j, d_i) will be taken into account when iterating through r_k. If no such r_k exists, there are no processed variables whose domain is contained in the interval $[r_j, d_i]$ and no unprocessed variables either. Therefore we have $\Delta(r_j, d_i) = \emptyset$ and no adjustment intervals are associated to the pair (r_j, d_i). In either case, the pair (r_j, d_i) can be ignored.

Case 2 ($r_j > r_i$): A release time r_j greater than r_i can also be safely ignored. Let d_l be the deadline processed before d_i. Since $|\Delta(r_j, d_i)| = |\Delta(r_j, d_l)|$ Lemma 6.3 insures that we have $(r_j, d_l) \succ (r_j, d_i)$ and adjustment intervals from (r_j, d_i) have already been taken into account when processing d_l.

Suppose while processing d_i we find the subsequence $(r_j, d_i) \prec (r_k, d_i)$, then we add to the set U_i the tuples (j, q) for $|\Delta(r_j, d_i)| \leq q < |\Delta(r_k, d_i)|$. Each tuple

$(j, q) \in U_i$ will be later used to create the adjustment intervals $I_{r_j,d_i,q}$ and $E_{r_j,d_i,q}$. These intervals are the ones appearing on the right hand side of Equation 6.5.

Algorithm 12: Enforcing bounds consistency on the INTER-DISTANCE constraint. We assume that the forbidden regions F have already been computed and that release times are sorted such that $r_i \leq r_{i+1}$ and $r_i = r_{i+1} \Rightarrow d_i \leq d_{i+1}$.

Let D be the set of deadlines sorted in increasing order. If two deadlines are equal, exclude from D the one whose associated release time is the smallest.
$P \leftarrow \emptyset$, $A \leftarrow \emptyset$, $U_i \leftarrow \emptyset$, $\forall\ 1 \leq i \leq n$
for $d_i \in D$ **do**
  $P \leftarrow P \cup \{j \mid d_j = d_i\}$
  $l \leftarrow \min(P)$
  for $j \in P \cap [l, i]$ **do**
    $a \leftarrow |\Delta(r_j, d_i)|$
    $b \leftarrow |\Delta(r_l, d_i)|$
    if $ect(F, r_l, b - a) < r_j$ **then**
      $U_i \leftarrow U_i \cup \{(l, q) \mid a \leq q < b\}$
      $l \leftarrow j$;
  $U_i \leftarrow U_i \cup \{(l, q) \mid 0 \leq q < |\Delta(r_l, d_i)|\}$
1 **for** $(j, q) \in U_i$ **do** $A \leftarrow A \cup I_{r_j,d_i,q}$
for *all deadlines* d_i *in non-decreasing order* **do**
2 $r'_i \leftarrow \min\{t \notin A \mid t \geq r_i\}$
  if $d_i \in D$ **then**
3 **for** $(j, q) \in U_i$ **do** $A \leftarrow A \cup E_{r_j,d_i,q}$
$\forall i,\ r_i \leftarrow r'_i$

Algorithm 12 prunes the release times. Notice that variables are indexed in non-decreasing order of release times. Should two tasks share the same release time, the task with the smallest deadline has the smallest index. Following [63], one can prune the upper bounds by creating the symmetric problem where task T'_i has release time $r'_i = -d_i$ and deadline $d'_i = -r_i$. Algorithm 12 can then prune the lower bounds in the symmetric problem, which prunes the upper bounds in the

original problem.

We assume that the forbidden regions F have already been computed in $O(n \log n)$ time (see [23]) and that release times are sorted such that $r_i \leq r_{i+1}$ and $r_i = r_{i+1} \Rightarrow d_i \leq d_{i+1}$. The function $lst(F, d_i, q)$ can be implemented with a table L where $lst(F, r_i, q) = L[i][q]$. Such a table requires $O(n^2)$ steps to initialize and supports function evaluation in constant time. We use a similar table to evaluate $ect(F, r, q)$. The function $|\Delta(r_j, d_i)|$ can trivially be computed in $O(n)$ steps. The function $|\Delta(r_{j+1}, d_i)|$ can later be computed in $O(1)$. The running time complexity of Algorithm 12 is $O(n^2)$ provided that Lines 1, 2, and 3 have time complexity $O(n)$. The next section describes how the adjustment data structure A can meet these requirements.

6.4.2 Keeping Track of Adjustment Intervals

To guarantee a quadratic running time, we must carefully design the data structure A that contains the adjustment intervals. We use a doubly-linked list containing all adjustment intervals sorted by lower bounds, including empty intervals. Each interval $I_{r_i,d_j,q}$ has a pointer $next(I_{r_i,d_j,q})$ and $previous(I_{r_i,d_j,q})$ pointing to the next and previous intervals in the list. The first interval has its *previous* pointer undefined as the last interval has its *next* pointer undefined. Each interval has also a pointer $nextQ(I_{r_i,d_j,q})$ pointing to $I_{r_k,d_j,q+1}$ where r_k and r_i might be equal. If the interval $I_{r_k,d_j,q+1}$ does not exist, the pointer is undefined. The data structure initially contains an empty interval with lower bound $-\infty$ used as a sentinel.

We implement Line 1 of Algorithm 12 as follows. We insert the intervals in decreasing order of lower bounds. Since we processes variables by increasing deadlines, the lower bound of $I_{r_j,d_i,0}$ is larger or equal to any lower bound inserted in A and is therefore inserted at the end of the linked list.

Suppose we have inserted the interval $I_1 = I_{r_j,d_i,q}$ and we now want to insert the interval $I_2 = I_{r_k,d_i,q+1}$. Algorithm 13 computes the insertion point in the linked list. The algorithm follows the *previous* pointers starting from I_1 until it either finds the insertion point or finds an interval whose *nextQ* pointer is assigned. In the later case, the algorithm follows the *nextQ* pointer to finally follow the *next* pointers until

the insertion point is reached. When following the $nextQ(I)$ pointer, the algorithm necessarily goes to or beyond the insertion point since we have $\min(I) < \min(I_1)$ and by Lemma 6.2 we have $\min(nextQ(I)) \leq \min(I_2)$.

Algorithm 13: Computing the insertion point of $I_{r_j,d_i,q+1}$ provided that $I_{r_j,d_i,q}$ has already been inserted.

```
I ← previous(I_{r_j,d_i,q})
I_2 ← I_{r_j,d_i,q+1}
while nextQ(I) is undefined ∧ min(I) > min(I_2) do
    I = previous(I)
if min(I) > min(I_2) then
    I ← nextQ(I)
    while min(next(I)) < min(I_2) do
        I ← next(I)
Insert I_2 after I
```

We show that Algorithm 13 inserts a sequence of $O(n)$ intervals in the linked list A in $O(n)$ steps. There is a maximum of n intervals in A whose $nextQ$ pointer is undefined, therefore the first while loop runs in $O(n)$ time. Let I_4 be an interval explored by the second while loop. The interval I_4 lies between $nextQ(I)$ and the insertion point. By Lemma 6.2, if an interval I_3 was pointing to I_4 with its $nextQ$ pointer, the interval I_3 would lie between I and I_1. Since $I_3 \neq I$, we conclude that no intervals point to I_4 with its $nextQ$ pointer. There is a maximum of n such intervals. The second while loop runs in $O(n)$. We therefore showed that Line 1 can be implemented in $O(n)$ steps. Since $\min(E_{r_i,d_j,q}) = \min(I_{r_k,d_j,q+1})$, Line 3 can be implemented by simply changing the upper bounds of internal adjustment intervals that were already inserted in A.

Line 2 of Algorithm 12 can be implemented in $O(\alpha(n))$ steps where α is the inverse of Ackermann's function. We create a union-find data structure S with elements from 1 to n. For each element i, we associate a time t_i initially set to r_i and a pointer p_i initially unassigned. When inserting adjustment intervals in A in decreasing order of lower bounds, we simultaneously iterate in decreasing order the sets in S. If an interval I is inserted such that $t_i \in I$, we set t_i to $\max(I) + 1$.

We then follow the *next* pointers from I to check if other intervals intersect t_i. If t_i becomes greater or equal to t_{i+1}, we merge the set in S containing i with the set containing $i+1$. The pointer p_i is used to keep track of the last interval I tested with t_i in order to not check twice a same interval. When executing Line 2 of Algorithm 12, we simply retrieve from S the set s containing i and return t_j where $j = \max(s)$.

6.5 Experiments

We implemented our algorithm using the ILog Solver C++ library, Version 4.2 [3]. The library already provides a propagator for the INTER-DISTANCE constraint called *IlcAllMinDistance* and offers two levels of consistency, namely *IlcBasic* and *IlcExtended*. We also implemented the Artiouchine-Baptiste propagator [5]. All experiments were run on a Pentium III 900 MHz with 256 Mb of memory. All reported times are averaged over 10 runs.

6.5.1 Scalability Test

In order to test the scalability of our propagator, we first consider a scheduling problem with a single INTER-DISTANCE constraint over n tasks whose release times are $r_i = 0$ and deadlines are $d_i = np$ for all tasks. This problem has a trivial solution and is solved without backtracking. We clearly see on Figure 6.1 that our propagator has a quadratic behaviour while the Artiouchine-Baptiste propagator has a cubic behaviour. This observation is supported by the study of the third and second derivative. The first and second derivative of the time function associated to the Artiouchine-Baptiste algorithm increases with n while the third derivative remains constant. This suggests that our implementation runs in $O(n^3)$ steps. The first derivative of the time function associated to our algorithm increases with n while the second derivative is constant. This suggests that our implementation of our algorithm runs in $O(n^2)$ steps.

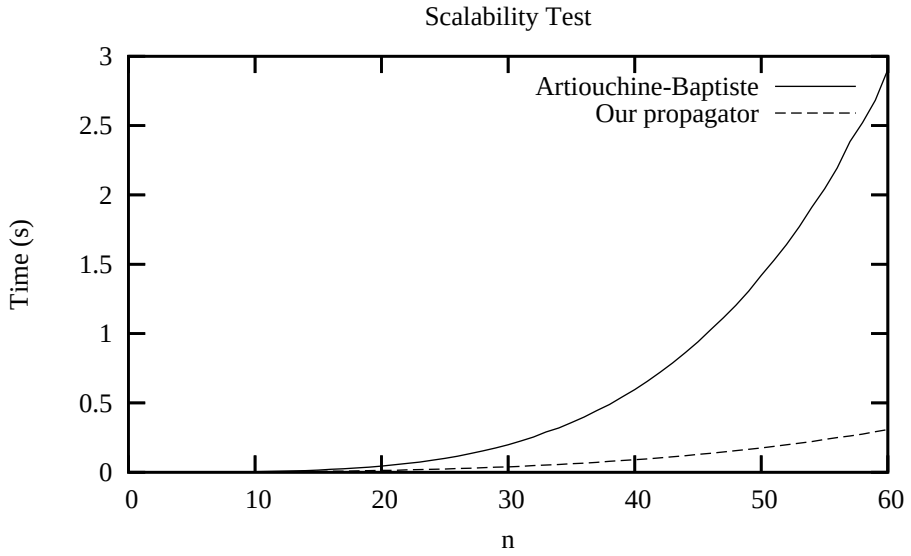

Figure 6.1: Running time of the Artiouchine-Baptiste propagator ($O(n^3)$) and our propagator ($O(n^2)$) in function of the number of tasks. For this scalability test, we set all release times to $r_i = 0$ and deadlines to $d_i = 6n$.

6.5.2 Runway Scheduling Problem

We then study the runway scheduling problem [6]. In this problem, n airplanes have certain time intervals in which they can land. Airplane number i has s_i time intervals $[r_i^1, d_i^1], \ldots, [r_i^{s_i}, d_i^{s_i}]$. Following [5], we create for each airplane a variable t_i with domain $[r_i^1, d_i^{s_i}]$ representing the landing time and a variable c_i with domain $[1, s_i]$ representing the landing time interval. We have the constraints $c_i \geq k \iff t_i \geq r_i^k$ and $c_i \leq k \iff t_i \leq d_i^k$. Finally, we have the constraint INTER-DISTANCE$([t_1, \ldots, t_n], p)$ that ensures a gap of p between each landing. For security reasons, we want to maximize the time p between each landing. We first solve the problem with $p = 1$ and double the value of p until the problem becomes unfeasible. Suppose the problem becomes unfeasible for the first time at $p = 2x$. We perform a binary search in $[x, 2x)$ to find the greatest p satisfying the problem.

Using the same benchmark as [5], we obtain the running times listed in Table 6.2 on random runway problems. For problems with 90 airplanes, we obtain savings

n	#	Our solution	A.-B.	IlcBasic	IlcExt.
15	1	0.09	0.16		
15	40	0.07	0.12	19.27	64.32
30	54	0.42	1.65		
45	20	0.59	3.40		
60	40	2.38	18.72		
75	30	4.04	37.67		
90	10	5.64	60.84		

Table 6.2: Time in seconds to solve some representative problems from the benchmark. n is the number of variables. # is the problem number in the benchmark. Blank entries represent problems that remained unsolved after 2 minutes of computation.

of an order of magnitude. Some propagators could not solve all problems within 2 minutes. All problems that could be solved with the propagators provided in ILog could also be solved by the Artiouchine-Baptiste propagator. All problems solved by the Artiouchine-Baptiste propagator could be solved at greater speed by our propagator.

We then consider the runway problem where all intervals have the same length (see Table 6.3). In these problems, ILog propagators were unable to solve problems in less than two minutes. We obtain an improvement over the Artiouchine-Baptiste propagator proportional to n. This observation is compatible with the running time complexities of the algorithms.

6.6 Conclusion

We presented a new propagator achieving bound consistency for the Inter-Distance constraint. The running time complexity of $O(n^2)$ improves by a factor of n the previous best known propagator. This theoretical improvement gives practical savings in scheduling problems.

It is still an open problem whether there exists an $O(n \log n)$ propagator for the Inter-Distance constraint achieving bound consistency. It would also be interesting to study how the constraint could be generalized for the cumulative

n	a	b	c	d	Our solution	A.-B.	Fails
20	10	10	5	6	0.11	0.25	28
20	10	10	5	6	0.09	0.17	4
30	8	15	3	6	3.40	14.26	2111
40	7	10	5	6	0.95	4.74	183
50	10	10	5	6	0.72	4.23	5
50	10	10	5	6	0.63	3.81	27
55	7	10	5	6	1.03	6.68	16
60	8	15	3	6	1.27	9.46	11
20	10	10	5	6	0.09	0.29	34
20	8	6	3	6	0.09	0.19	13
40	10	20	3	6	0.70	3.54	67
40	7	10	5	6	0.47	2.14	19
50	10	10	5	6	3.99	26.66	435
50	8	6	3	6	0.83	5.26	35
55	8	15	3	6	0.89	6.19	16
60	8	15	3	6	1.18	8.95	25
60	8	15	3	6	1.64	12.54	44

Table 6.3: Time in seconds to solve the runway problems where landing time intervals have size a, the gap between landing time intervals is of size b, and where $c \leq s_i \leq d$ holds.

scheduling problem. For some optimization problems, it would be convenient to consider p as a constrained variable on which we could enforce bound consistency.

Chapter 7

Conclusion

The design of new constraint propagators constitutes the main contribution of this thesis. We presented a new propagator for the bounds consistency of the All-Different constraint with linear worst case complexity. This propagator outperforms previous propagators achieving the same consistency. We also presented the first propagator with amortized linear time complexity achieving range consistency.

Our research about the GCC offers a better understanding of the mathematical theory behind this constraint. We showed how the GCC can be divided into two simpler constraints without hindering propagation. Based on this, we provided a propagator for the bounds consistency of the GCC with linear time complexity. We showed that range consistency can be enforced in amortized linear complexity as well. We improved the time complexity for enforcing domain consistency from $O(|X|^2|D|)$ to $O(|X|^{1.5}|D|)$. We showed how the universality of GCC can be efficiently tested. Finally, we proved that it is NP-Hard to enforce domain consistency on EXT-GCC.

We explored a generalization of the All-Different constraint called the Inter-Distance constraint. We designed a new propagator with quadratic running time complexity that outperforms both in theory and in practice previous propagators.

Many problems concerning the constraints studied in this thesis remain open.

Generally, combining two constraints together often results in a better filtering of the variable domains. Régin [71] study the combination of a GCC with an AMONG constraint. Beldiceanu and Carlsson [7] study an ALL-DIFFERENT constraint where variables are also subject to a MIN and a MAX constraint. It would be interesting to determine which constraints often occur with the ALL-DIFFERENT constraint or the GCC and how the combination of these two constraints can strengthen the propagation.

Three main problems concerning the INTER-DISTANCE constraint still remain to be solved. Is there an $O(n \log n)$ propagator that enforces bounds consistency for this constraint? How can we generalize the INTER-DISTANCE constraint to solve the cumulative scheduling problem. How can we make p a constrained variable whose domain is pruned by the propagator?

The propagators presented in this thesis can be used on a large variety of combinatorial problems. Several of these propagators have already been integrated in open-source and commercial constraint programming libraries.

Bibliography

[1] *ECLiPSe User Manual Release 5.3*, 2002.

[2] *Gecode Reference Documentation*. http://www.gecode.org/, version 0.9.0 edition, November 2005.

[3] ILOG S. A. ILOG Solver 4.2 user's manual, 1998.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Networks Flows, Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[5] K. Artiouchine and P. Baptiste. Inter-distance constraint: An extension of the all-different constraint for scheduling equal length jobs. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, pages 62–76, 2005.

[6] K. Artiouchine, P. Baptiste, and C. Dürr. Runway scheduling with holding loop. In *Proceedings of Second International Workshop on Discrete Optimization Methods in Production and Logistics*, pages 96–101, 2004.

[7] N. Beldiceanu and M. Carlsson. A cumulative model for a pattern sequence problem. In *Proceedings of Constraint Modelling Challenge 2005*, pages 14–20, 2005.

[8] N. Beldiceanu, M. Carlsson, and J.-A. Rampon. Global constraint catalog. Technical Report T2005:08, Swedish Institute of Computer Science, August 2005.

[9] N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Mathematical Computer Modelling*, 20(12):97–123, 1994.

[10] C. Berge. Two theorems in graph theory. In *Proceedings of the National Academy of Sciences of the United States of America*, number 43, pages 842–844, 1957.

[11] C. Berge. *Graphes and Hypergraphes*. Dunod, Paris, 1970.

[12] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. *The Ciao Prolog System*. The Computational logic, Languages, Implementation, and Parallelism (CLIP) Group, August 2004.

[13] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operation Rsearch*, 78:146–161, 1994.

[14] C. W. Choi, W. Harvey, J. Ho-Man Lee, and P. J. Stuckey. Finite domain bounds consistency revisited. Technical Report cs.AI/0412021, arXiv.org, 2004.

[15] A. Colmerauer and P. Roussel. *History of Programming Languages*, chapter The Birth of Prolog. ACM Press/Addison-Wesley, 1996.

[16] R. Debruyne and C. Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 412–417, 1997. Nagoya, Japan.

[17] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, 1988.

[18] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4:507–518, 1975.

[19] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, 1962.

[20] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, 1983.

[21] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991.

[22] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. New York : W. H. Freeman, 1979.

[23] M.R. Garey, D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal on Computing*, 10(2):256–269, 1981.

[24] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 179–193, 1996.

[25] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. Technical report, APES-09-1999, 1999. Available from http://www.csplib.org/.

[26] C. Gervet. *Set Intervals in Constraint Logic Programming: Definition and Implementation of a Language*. PhD thesis, Université de Franche-Comté, France, 1995.

[27] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints Journal*, 1(3):191–244, 1997.

[28] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, pages 26–30, 1935.

[29] Y. Hamadi. *Disolver: the Distributed Constraint Solver*. Microsoft Research, version 2.0 edition, March 2005.

[30] E. Hebrard. Mistral, 2006. http://www.cse.unsw.edu.au/~ehebrard/ mistral/doxygen/html/index.html.

[31] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, January 1999.

[32] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in OPL. In *Proceedings of the First International Conference on Principles and Practice of Declarative Programming*, pages 98–116, 1999.

[33] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1–3):139–164, Oct-Dec 1998.

[34] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.

[35] C. E. Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the 1rst International Conference on Artificiel Intelligence (IJCAI-69)*, pages 265–301, 1969.

[36] J. Hopcroft and R. Karp. A $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, (2):225–231, 1973.

[37] J. Hopcroft and J. Ullman. Set merging algorithms. *In SIAM Journal on Computing*, 2(4):294–303, 1973.

[38] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, release 3.12.3 edition, October 2005.

[39] I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, pages 437–451, 2003. LNCS 2833.

[40] I. Katriel and S. Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10(3):191–217, 2005.

[41] D. Knuth. Volume 4 of the art of computer programming, pre-fascicle 3a: Generating all combinations. http://www-cs-faculty.stanford.edu/~knuth/, 2005.

[42] D. Knuth. Volume 4 of the art of computer programming, pre-fascicle 3a: Generating all n-tuples. http://www-cs-faculty.stanford.edu/~knuth/, 2005.

[43] Koalog. *An overview of Koalog Constraint Solver*, 2005. http://koalog.com/.

[44] F. Laburthe and N. Jussien. *Choco*, 2004. http://choco.sourceforge.net/.

[45] C. F. Laywine and G. L. Mullen. *Discrete mathematics using Latin squares*. Wiley-IEEE, 1998.

[46] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning*, pages 19–28, 1996.

[47] C. Lee, M. Potkonjak, and W. Manginoe-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications. In *Proceedings of International Symposium on Microarchitecture*, pages 330–335, 1997.

[48] H. Levesque. Planning with loops. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 509–515, 2005.

[49] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 232–238, 1993.

[50] W. Lipski and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.

[51] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. 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.

[52] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 165(2):165–185, 1977.

[53] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. Technical Report CS-2005-19, School of Computr Science, University of Waterloo, 2005.

[54] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. *Sixth International Conference on Principles and Practice of Constraint Programming*, 2000.

[55] L. Mercier and P. Van Hentenryck. Edge finding for cumulative scheduling. Submitted for publication, 2005.

[56] R. E. Miller and J. W. Thatcher, editors. *Complexity of Computer Computations*. Plenum Press, New York - London, 1972.

[57] T. Müller and M. Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *Workshop Logische Programmierung*, pages 104–115. Technische Universität München, September 1997.

[58] M. Paterson. Unpublished report. University of Warwick, Coventry, Great Britain.

[59] J. Petersen. Die theorie der regulären graphen. *Acta Mathematica*, 15:193–220, 1891.

[60] T. Petit, J.-C. Régin, and C. Bessière. Specific filtering algorithms for over-constrained problems. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 451–463, 2001.

[61] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 1992.

[62] J.-F. Puget. Finite set intervals. In *Proceedings of Workshop on Set Constraints*, 1996.

[63] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificiel Intelligence (AAAI-98) and the 10th Conference on Innovation Applications of Artificial Intelligence (IAAI-98)*, pages 359–366, 1998.

[64] C.-G. Quimper, A. Golynski, A. López-Ortiz, and P. van Beek. An efficient bounds consistency algorithm for the global cardinality constraint. *Constraint Journal*, 10:115–135, 2005.

[65] C.-G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. Improved algorithms for the global cardinality constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 542–556, September 2004.

[66] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, pages 600–614, 2003. LNCS 2833.

[67] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 11th National Conference on Artificiel Intelligence (AAAI-94)*, pages 362–367, 1994.

[68] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Eighth Annual Conference on Innovative Applications of Artificial Intelligence*, pages 209–215, 1996.

[69] J.-C. Régin. The global minimum distance constraint. Technical report, ILOG, 1997.

[70] J.-C. Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3–4):387–405, 2002.

[71] J.-C. Régin. Combination of among and cardinality constraints. In *CP-AI-OR'05*, pages 288–303, 2005.

[72] J.-C. Régin and C. P. Gomes. The cardinality matrix constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 572–587, 2004.

[73] P. Van Roy. Logic programming in Oz with Mozart. In Danny De Schreye, editor, *International Conference on Logic Programming*, pages 38–51, Las Cruces, NM, USA, November 1999. The MIT Press.

[74] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *PPCP '94: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 10–20, 1994.

[75] A. Sadler and C. Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 604–618, 2004.

[76] C. Schulte and P. J. Stuckey. When do bounds and domain propagation lead to the same search space. In *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pages 115–126, 2001.

[77] B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 182–187, 2000.

[78] K. Stergiou and T. Walsh. The difference all-difference makes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 414–419, 1999.

[79] M. E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. In *Processing of the 8th International Conference on Automated Deduction*, pages 573–587. Springer-Verlag New York, Inc., 1986.

[80] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[81] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[82] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 625–639, 2001.

[83] W. J. van Hoeve. A hyper-arc consistency algorithm for the soft alldifferent constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 679–689, 2004.

[84] W. J. van Hoeve. The alldifferent constraint: a systematic overview. Submitted manuscript. Available from http://www.cs.cornell.edu/ vanhoeve/papers/alldiff.pdf, 2005.

[85] W. J. van Hoeve, G. Pesant, and L.-M. Rousseau. On global warming (softening global constraints). In *6th International Workshop on Preferences and Soft Constraints (held in conjunction with CP 2004)*, 2004.

[86] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 1388–1395, 1997.

[87] T. Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, pages 724–738, 2003.

[88] N.-F. Zhou. *B-Prolog User's Manual (Version 6.7), Prolog, Agent, and Constraint Programming.* Afany Software, CUNY, Kyutech, March 1995.

Index