Implementation and Evaluation of Advanced Propagation Algorithms for Global Constraints

(Fortgeschrittenenpraktikum)
FoPra thesis

Patrick Pekczynski
Supervisor: **Guido Tack**
Responsible Professor: **Prof. Gert Smolka**

Programming Systems Lab
Department 6.2 - Computer Science
Faculty 6 - Natural Sciences and Technology I
Saarland University, 66041 Saarbrücken, Germany,
`pekczynski@ps.uni-sb.de`,

According to their nature, there are both people who have quick intelligence, and those who must withdraw and take time to think things over.
— Hagakure [40]

Abstract. In this paper we present the implementation and evaluation of advanced propagation algorithms for global constraints. We discuss an $\mathcal{O}(t + n)$ bounds consistent algorithm for the `sortedness` constraint [39] as well as an $\mathcal{O}(n \cdot log(n))$ bounds consistent [24], an $\mathcal{O}\left(n \cdot d \cdot \min\left(\sqrt{n + d}, \frac{n}{2}\right)\right)$ domain consistent algorithm [25] for the `global cardinality` constraint as well as extensions of these constraints, where n is the number of variables, t is the sorting time for the variables and d is the cardinality of the largest variable domain. Furthermore, we show that the advantage of these propagation algorithms is not only theoretically but also practically by using *Gecode* [38], a *generic constraint development environment*, a C++ library for constraint programming as implementation and evaluation platform.

1 Introduction

Constraint programming is a powerful state-of-the-art approach to solve hard combinatorial problems. Imposing a more high-level view on these problems, global constraints yield a stronger pruning and increase the efficiency of constraint programming in order to solve them. This paper presents the implementation and evaluation of advanced propagation algorithms for global constraints in *Gecode* [38] , a *generic constraint development environment*, a C++ library for constraint programming. The paper is organized as follows: section 2 introduces the basic notation and the definitions for *constraint programming* as well as the essential components of *constraint propagation* and *constraint distribution*, whereas the emphasis lies on introducing propagators as the computational analog of constraints and constraint propagation as global mechanism behind the execution of propagators. Subsequently, section 3 studies advanced propagation algorithms for global constraints. It focuses on a bounds consistent algorithm for the `sortedness` constraint [39] and its bounds consistent extension to permutation variables. Furthermore, section 3 discusses a bounds consistent [24] and a domain consistent [25] propagation algorithm for the `global cardinality` constraint as well as its extension to cardinality variables. Section 4 provides an outline of how the *Gecode* architecture improves constraint propagation with elaborate mechanisms and takes care of the communication between propagators and problem variables. The last section 5 presents an empirical evaluation of the implementation of the discussed propagation algorithms by providing practical models of constraint satisfaction problems demonstrating not only the theoretical but also the practical gain of the `sortedness` and `global cardinality` propagators. The remaining sections contain a summarization, the contributions of this thesis and prospects on future work.

2 The Framework of Constraint Programming

In this section we introduce *constraint programming (CP)* as the underlying framework for this paper and explain the basic terminology we use. Apart from this basic introduction to CP , we further recapitulate elementary definitions of graph theory used in the propagation algorithms that we focus on in this paper. The following definitions concerning CP -terminology are taken from lecture notes on *Constraint Programming 05* [18] and from [34].

2.1 The Basics

First of all we restrict the term of constraint programming to finite domain constraint programming because the propagation algorithms we discuss in the remaining sections reason about variables ranging over *finite domains*.

Definition 1 (Finite-domain variable (FDVar)). *Taken from a set $\mathfrak{X}$ of typed variables a* finite-domain variable *x with type τ ($x : \tau$) is associated with a finite domain D (a finite set of values) of type τ,e.g. $\tau = \mathbb{N}$, $\tau = \mathbb{Z}$. Hence, x ranges over the values in D, written $x : D$, which is a shorthand notation for $dom(x) = D$ or $x \in D$. In this context, $x : \{0, 1, 2, 3\}$ denotes that x is an FDVar of type $\tau = \mathbb{N}$ and it can take any value $v \in \{0, 1, 2, 3\}$.*

In order to describe which values an FDVar can take, there is the notion of a valuation and a variable assignment.

Definition 2 (Valuation and Assignment). *Let $\mathcal{X}$ be a finite set of FDVars. A* valuation *for $\mathcal{X}$is a mapping $v : \mathcal{X} \mapsto \tau$, s.t. $\forall x \in \mathcal{X} : \tau = dom(x) \wedge v(x) \in \tau$. An* assignment *for $\mathcal{X}$is a valuation $\alpha : \mathcal{X} \mapsto \tau$, s.t. $\forall x \in \mathcal{X} : \alpha(x) \in dom(x) = \tau$, where $\alpha(x)$ denotes the value assigned to x. The set of all valuations is called $\mathtt{val}(\mathcal{X})$ and the set of all assignments is referred to as $\mathtt{ass}(\mathcal{X})$.*

The central notion in CP is the notion of a **constraint**. For the remainder of this paper a constraint is defined as follows:

Definition 3 (Constraint). *Given a finite set of FDVars $\mathcal{X}$, a constraint for $\mathcal{X}$ is a set $C \subseteq \mathtt{val}(\mathcal{X})$ of valuations for $\mathcal{X}$.*

As a constraint C does not necessarily constrain all the variables in $\mathcal{X}$, there is also the notion of the scope of a constraint:

Definition 4 (Scope of a constraint). *Let $\mathcal{X}$ be a finite set of FDVars and $\mathcal{S} \subseteq \mathcal{X}$. If a constraint C has the scope $\mathcal{S}$,*

$$C := \{v \in \mathtt{val}(\mathcal{X}) | \ \mathbf{if}\ (x : \tau) \in \mathcal{S}\ \mathbf{then}\ v(x) \in (\tau' \subseteq \tau)\ \mathbf{else}\ v(x) \in \tau\}$$

Hence, a variable $x \in \mathcal{S}$ can only be mapped to a value $v(x) \in \mathtt{val}(\mathcal{X}) \supseteq C$, where $v(x)$ is restricted by $\tau' \subseteq \tau$ defined in C, whereas all $x \notin \mathcal{S}$ can be mapped to any value $v(x)$ respecting the type τ of the variable.

2.2 Phases of constraint programming

Solving a problem with CP consists of two essential phases, the *modeling phase* and the *solution phase*. During the first phase the focused problem is encoded into a special constraint formulation, a *constraint satisfaction problem*.

Definition 5 (Constraint Satisfaction Problem (CSP)). *A* constraint satisfaction problem *(*CSP*) is a tuple $\langle \mathcal{X}; \mathfrak{C} \rangle$, where $\mathcal{X}$ denotes a finite set of FDVars x_i, the problem variables of the CSP, with associated domains D_i, written $x_i : D_i$, and $\mathfrak{C}$ denotes a set of constraints on $\mathcal{X}$.*

Provided a CSP encoding of a problem, the second phase of CP tries to find a solution to the CSP.

Definition 6 (Solution to a CSP). *Given a CSP $P := \langle \mathcal{X}; \mathfrak{C} \rangle$ and a constraint $C \in \mathfrak{C}$, a valuation α **satisfies** C if $\alpha \in C$ and α is a **solution** to P if $\alpha \in \bigcap_{C \in \mathfrak{C}}$. The set of all assignments for $\mathcal{X}$ satisfying C is called $\mathtt{sat}(C)$ and the set of all solutions to a given problem P is denoted as $\mathtt{sol}(P)$.*

As solving a CSP is $\mathcal{NP}$-hard in general [4], the second phase of CP is characterized by the hybrid design of the two basic components *constraint propagation* and *constraint distribution* [33], where constraint distribution is again divided into the components of *branching* and *search*. In order to decide, whether a CSP $P = \langle \mathcal{X}; \mathfrak{C} \rangle$ has a solution, a naive *generate-&-test-method* [4] consists in generating all valuations $v \in \mathtt{val}(\mathcal{X})$ and testing for every possible valuation v, whether $v \in \mathtt{sol}(P)$ holds or not. Obviously, this algorithm succeeds in deciding whether there is a solution in P or not, but generating $\mathtt{val}(\mathcal{X})$ easily leads to a combinatorial explosion, for example if P is the CSP-encoding of an $\mathcal{NP}$-complete scheduling problem. Even if P has no solution ($\mathtt{sol}(P) = \emptyset$), this approach has to test all possible variable assignments in the search space of P. In order to avoid unnecessary testing for valuations $v \notin \mathtt{val}(\mathcal{X})$ this naive *generate-&-test-method* is extended by the inference technique of constraint propagation.

2.3 Constraint Propagation

The major contribution of constraint propagation to the *generate-&-test-method* is the pruning of a CSP's search space. Hence constraint propagation reduces the set of all possible valuations in size. This is realized by domain reduction which is defined as follows:

Definition 7 (Domain reduction (p.3 [18])). *Given two CSPs $P := \langle \mathcal{X}; \mathfrak{C} \rangle$ and $P' := \langle \mathcal{X}; \mathfrak{C} \rangle$ only differing in the associated variable domains ($dom(x)$ of P differs from the mapping $dom'(x)$ of P') P is called a **reduction** of P' ($P < P'$) if these criteria hold:*

$$\forall x \in \mathcal{X} : dom(x) \subseteq dom'(x)$$

$$\exists x \in \mathcal{X} : dom(x) \subset dom'(x)$$

Using the concept of domain reduction, constraint propagation tries to narrow the cardinality of the variable domain as much as possible.

Definition 8 (Variable determination). *Given an FDVar $x : D \in \mathcal{X}$, we say that x is* determined *or x is* assigned, *if its associated domain $D = dom(x)$ contains only one value, that is $D = \{d\}$.*

Due to these definitions of domain reduction and variable determination a CSP $\langle \{x_1 : \{d_1\}, \ldots, x_n : \{d_n\}\}; \mathfrak{C} \rangle$ corresponds to a variable assignment $\alpha = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. Using the above definitions, the following inference rule gives a trace how *constraint* *propagation* can reduce the number of valuations the *generate-&-test* method has to take care of:

$$\textsc{Constraint Propagation}\quad \frac{P := \langle \mathcal{X} \cup \{x : D\}; \mathfrak{C} \cup \{C\}\rangle \qquad d \in D \qquad \mathtt{sat}(C) \cap \{\alpha \in \mathtt{ass}(\mathcal{X} \cup \{x : D\}) | \alpha(x) = d\} = \emptyset}{P' := \langle \mathcal{X} \cup \{x : D' := (D - \{d\})\}; \mathfrak{C} \cup \{C\}\rangle}$$

The propagation mechanism checks whether a domain D of an FDVar x contains values d that cannot take part in a solution of the CSP and removes them from the domain. Thus it reduces $D = dom(x)$ and hence P to P' by pruning inconsistent valuations from $\mathtt{val}(\mathcal{X}) \cup \{x : D\})$. Furthermore, the size of the search space decreases and so does the time needed to test whether the remaining valuations are solutions to the problem. During this inference constraint propagation makes use of the information specified in the constraint $C \in \mathfrak{C}$ to decide, whether a value has to be removed or not.

2.3.1 Propagators

As we have seen the inference mechanism of constraint propagation is able to exclude valuations from being tested for feasibility and hence avoids an unnecessary search for solutions in inconsistent valuations for the set of problem variables. The importance of constraint propagation in CP is further underlined by the fact that state-of-the-art constraint solvers implement constraint propagation as a essential technique to solve CSPs efficiently. As the above inference rule is instantiated differently for every constraint $C \in \mathfrak{C}$, there has to be a corresponding propagation rule for every constraint of the CSP. The implementation of this inference rules is realized by the concept of *propagators*. In order to explain this concept, which is essential for the algorithms presented in the following sections, we have to provide some more definitions concerning the architecture that was used for their implementation.

Definition 9 (Constraint Store ([31, 33])). *A constraint store $s \in S$ is a conjunction of logical formulas representing information about values of FDVars. These logical formulas have the form $x_i \in D_i$ and are called* basic constraints. *A store s incorporating these basic constraints can be seen as a mapping s from FDVars to their current domains corresponding to the current variable domain dom [19]. A constraint store s implements the set of FDVars $\mathcal{X} = \{x_1 : D_1, \ldots, x_n : D_n\}$, and we write*

$$\begin{aligned} s &\overset{def.}{:=} \{x_i \in D_i, \ldots, x_n \in D_n\} \\ &\equiv \{x_i : D_i, \ldots, x_n : D_n\} \\ &\equiv \{x_i \in D_i \wedge \ldots \wedge x_n \in D_n\} \end{aligned}$$

Given two stores $s_1, s_2 \in S$ we say that s_1 is stronger *than s_2 if $s_1 \leq s_2 \Leftrightarrow \forall x \in \mathcal{X} : s_1(x) \subseteq s_2(x)$. Further, we call s_1 strictly stronger than s_2 if $s_1 < s_2 \Leftrightarrow \forall x \in \mathcal{X} : s_1(x) \subset s_2(x)$.*

From this definition of a constraint store, a propagator can be defined as follows:

Definition 10 (Propagator). *A propagator $p : S \mapsto S$ is a monotonically decreasing mapping [34] from stores to stores, that is:*

$$\begin{aligned} &\forall s \in S : p(s) \leq s \\ &\forall s_1, s_2 \in S : s_1 \leq s_2 \Rightarrow p(s_1) \leq p(s_2) \end{aligned}$$

Furthermore, a propagator p has additional properties. Let $P = \langle \mathcal{X}; \{C\}\rangle$ be a CSP and let $s \in S$ be the store representing $\mathcal{X}$.

1. *A propagator p implementing C is called* correct for *C if*

$$\texttt{val}(s) \cap C = \texttt{val}(p(s)) \cap C$$

that is, propagation of p maintains solutions.
2. *Given $\alpha \in \texttt{val}(P)$, p is* checking *with respect to α, that is, p can decide whether $\alpha \in \texttt{sol}(P) = \texttt{sat}(\mathcal{C})$ or not and hence whether α satisfies a constraint C or not.*

The combination of a constraint store $s \in S$ and the set of all propagators $p_i \in S \mapsto S$ defined on s form the ***computation space****, an essential component of the* Gecode *architecture. Hence the computation space represents the implementation of a CSP $P := \langle \mathcal{X}; \mathfrak{C} \rangle$ where the store s implements the set of FDVars and the propagators p_i implement the constraints $C \in \mathfrak{C}$ defined on $\mathcal{X}$.*

2.3.2 Propagation Status

In the context of these definitions, a propagator p is the implementation of the inference rule seen above, instantiated with a specific constraint $C \in \mathfrak{C}$. Hence p is the counterpart of C on the implementation level of a constraint solver. As a such, p only modifies a constraint store s with respect to the FDVars in $PS := \{x : D \in s | x \in \mathcal{S}\}$, where PS can be considered as the scope of a propagator. Thus, for every constraint $C \in \mathfrak{C}$ specified in a CSP P we have at least one or more propagators implementing C, where the set of all these propagators performs the constraint propagation on the valuation set of $\mathcal{X}$ given in the CSP P. Therefore a constraint solver has to keep track of this set of propagators $\mathcal{P}$ specified by the constraint set $\mathfrak{C}$ of the CSP-encoding and needs to apply every $p \in \mathcal{P}$ at least once to the first constraint store given by the set of FDVars $\mathcal{X}$ in the CSP. After propagation of p on the first store $s := \{x_1 : D_1 \wedge \ldots \wedge x_n : D_n\}$, propagation yields one of the following status:

[P_1] Failure:
$p(s) = \bot \Leftrightarrow \exists x_i : D_i \in p(s) : D_i = \emptyset$
There is no value $v \in D$, such that $\alpha = (x_1 : d_1, \ldots, x_i : v, \ldots, x_n : d_n) \in \texttt{sat}(\mathcal{C})$ holds.

[P_2] Success and Fixpoint reached :
$p(p(s)) = p(s) \Leftrightarrow p(s') = s' \Leftrightarrow \forall x_i : D_i \in s' : \left(\neg\exists D'_i \subset D_i : x_i : D'_i \in p(s')\right)$
s' has been reached by propagation of p on s, but applying p once more does not narrow the domains any further. If a propagator p has reached its fixpoint after propagation, we say that p is **idempotent** and $p(s) = s'$ is a fixpoint of p.

[P_3] Success, but no Fixpoint reached:
Propagation of p has not reached its fixpoint and p has to be executed again to find out whether after repeated iteration it achieves its fixpoint or detects an inconsistency.

[P_4] Subsumption (entailment):
Let $s \in S$. If $\forall t \in S : t \leq s \wedge p(t) = t$ holds, p is **subsumed** in s.
After propagation p cannot contribute new information on any store that is achieved by further narrowing of the variable domains. Hence p does not need to be respected for stronger spaces and can be removed from $\mathcal{P}$ of scheduled propagators.

As we have seen, the inference mechanism of constraint propagation is a major reason for the efficiency in the second phase for constraint programming. This increase in efficiency is not only due to detection of inconsistencies [P_1] but also due to a monotonic reduction of the search space [P_2] – [P_4] because the deduction mechanism removes possible valuations by discarding domain values that cannot be part of a solution to the CSP. Nevertheless it is possible that *constraint propagation* does not suffice to find a solution to the problem, especially if the propagators have already achieved status [P_2] although at least one problem variable has not been determined so far. Since the general CSP is known to be $\mathcal{NP}$-complete [10] and *constraint propagation* is incomplete [2] with respect to reducing the variable domains further and finding all solutions to the problem we also need a search architecture performing further domain reduction on the domains of the problem variables. This is achieved by applying the following inference rule:

$$\textsc{Branching}\ \frac{\langle \mathcal{X} \cup \{x : D\}; \mathfrak{C}\rangle \qquad |D| > 1 \qquad D = D_1 \uplus \ldots \uplus D_k}{\langle \mathcal{X} \cup \{x : D_1\}; \mathfrak{C}\rangle | \ldots | \langle \mathcal{X} \cup \{x : D_k\}; \mathfrak{C}\rangle}$$

The process of partitioning the variable domains as given in this rule and the search of an index $1 \leq i \leq k$ to be processed further is called *branching*. This process reduces the variable domains until *constraint propagation* is able to perform new inferences on a reduced CSP $P := \langle \mathcal{X} \cup \{x : D_i\}; \mathfrak{C}\rangle$. Thus the combination of constraint propagation and the branch and search process of *constraint distribution* infers whether a CSP is solvable or not and ends the solution phase of CP .

2.3.3 Consistency Level

Apart from the propagation status a further characteristic of a propagator is its *consistency level*. The consistency level of a propagator p implementing a constraint C denotes the strength of the pruning performed on the variables in the scope $\mathcal{S}$ of C. In the subsequent sections concerning the presented algorithms, their implementation and evaluation we will focus on the following basic consistency notions of *bounds consistency* and *domain consistency*. Analogously to the notion of $\mathtt{ass}(\mathcal{X})$ in definition 2 we define $\mathtt{ass}_{\mathcal{X}}(\mathcal{bnd}) := \{\alpha \in \mathtt{val}(\mathcal{X}) | \forall x : D \in \mathcal{X} : \alpha(x) \in [\min(dom(x)) .. \max(dom(x))]\}$

Definition 11 (Bounds consistency). *A propagator p is called* bounds consistent *if it narrows the domains of the variables in its scope such that the following holds:*

$$\forall x \in \mathcal{X} : \forall d \in \{min(dom(x)), max(dom(x))\} : \exists \alpha \in \mathtt{ass}_{\mathtt{bnd}}(\mathcal{X}) : \alpha \in C \wedge \alpha(x) = d$$

If a propagator p is bounds consistent according to the above definition p only performs propagation such that the lower and upper bounds of the variable domains are consistent with the constraint C. Thus bounds consistent propagators do not have to propagate on the entire domain of the problem variables and hence their theoretical complexity is often linear or quasilinear in the problem size $\mathcal{O}(\mathcal{X})$.

Definition 12 (Domain consistency). *In contrast to bounds consistency, a propagator p is called* domain consistent *if it narrows the domains of the variables in its scope such that:*

$$\forall x \in \mathcal{X} : \forall d \in dom(x) : \exists \alpha \in \mathtt{ass}(\mathcal{X}) : \alpha \in C \wedge \alpha(x) = d$$

In comparison to a bounds consistent propagator, a domain consistent propagator p has to guarantee, that not only the bounds but all values in the domain of the variables are consistent with C. Thus bounds consistent propagators often yield a *faster pruning* than domain consistent propagators, but domain consistent propagators perform a *stronger pruning* as they reason about the entire variable domain and not only about the domain bounds. Apart from these two consistency levels the *Gecode* library also allows *value consistent* propagators, where the notion of *value consistency* which is defined as follows:

Definition 13 (Value consistency). *A propagator p is called* value consistent *if it narrows the domains of the variables in its scope such that:*

$$(\exists x : D \in X : |D| = 1) \Rightarrow \forall x \in \mathcal{X} : \forall d \in dom(x) : \exists \alpha \in \mathtt{ass}(\mathcal{X}) : \alpha \in C \wedge \alpha(x) = d$$

It is important that a value consistent propagator p can only perform propagation if there is at least one variable in its scope is assigned. Otherwise it cannot perform any propagation on the variable domains. Thus, value consistency is a naive form of checking, whether $\alpha \in C$ still holds if $\alpha(x) = d$ for a variable $x \in \mathcal{X}$ in the scope of p.

2.3.4 Local versus Global Constraints

An essential approach to increase the efficiency of constraint solvers is the use of *global constraints*. An important property of *global constraints* is their larger scope resulting in a more general view on variables than other constraints have. The most prominent example for a global constraint is the `alldiff` constraint with scope $\mathcal{X} := \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$, where

$$\texttt{alldiff} = \{\alpha \in \texttt{val}(\mathcal{X}) | \forall x_i, x_j, i \neq j \in (X) : \alpha(x_i) \neq \alpha(x_j)\}$$

The naive way of imposing this constraint on the variables in $\mathcal{X}$ is the combination of $\frac{n \cdot (n-1)}{2} \in \mathcal{O}\left(n^2\right)$ binary $\texttt{noteq}_{x \neq y}$ constraints of the form

$$\texttt{noteq}_{x \neq y} = \{\alpha \in \texttt{val}(\mathcal{X}) | x, y \in \mathcal{X} \wedge \alpha(x) \neq \alpha(y)\}$$

Let $\mathcal{X} := \{x, y, z\}$, $s := \{x : [1..2], y : [1..2], z : [1..2]\}$ and consider the CSPs

$$\begin{aligned} C_1 &= \langle \mathcal{X}, \{\texttt{noteq}_{x \neq y}, \texttt{noteq}_{y \neq z}, \texttt{noteq}_{z \neq x}\} \rangle \\ C_2 &= (\mathcal{X}, \{\texttt{alldiff}\}) \end{aligned}$$

with

$$\begin{aligned} \texttt{val}(\mathcal{X}) = \{&\{x \mapsto 1, y \mapsto 1, z \mapsto 1\}, \{x \mapsto 1, y \mapsto 1, z \mapsto 2\}, \\ &\{x \mapsto 1, y \mapsto 2, z \mapsto 1\}, \{x \mapsto 1, y \mapsto 2, z \mapsto 2\}, \\ &\{x \mapsto 2, y \mapsto 1, z \mapsto 1\}, \{x \mapsto 2, y \mapsto 1, z \mapsto 2\}, \\ &\{x \mapsto 2, y \mapsto 2, z \mapsto 1\}, \{x \mapsto 2, y \mapsto 2, z \mapsto 2\}\} \end{aligned}$$

Given domain consistent propagators for `alldiff` and $\texttt{noteq}_{x \neq y}$:

$$\begin{aligned} \texttt{alldiff}(s) = \{&\ldots, \\ &x_i : \{v \in s(x_i) | \forall x_j : s(x_j) \in \mathcal{X}, i \neq j : v \notin s(x_j)\}, \\ &, \ldots, \\ &x_j : \{v \in s(x_j) | \forall x_i : s(x_i) \in \mathcal{X}, j \neq i : v \notin s(x_i)\}, \\ &\ldots\} \end{aligned}$$

$$\begin{aligned} \texttt{noteq}_{x \neq y}(s) = \{&\ldots, \\ &x : \{v \in s(x) | v \notin s(y)\}, \\ &y : \{v \in s(y) | v \notin s(x)\}, \\ &\ldots\} \end{aligned}$$

constraint propagation of $\texttt{noteq}_{x \neq y}$ on s in $C_1$is not able to narrow the domains of the variables any further and the inconsistency of the CSP (there is no solution to the CSP) is only detected through exploration of the whole search space $\texttt{val}(\mathcal{X})$. In C_2 however, propagation of `alldiff` can exploit the global knowledge of the `alldiff` propagator, since $\texttt{alldiff}(s) = \bot$ and hence $\texttt{sol}(C_2) = \emptyset$, which would immediately stop a constraint-solver and not perform any search. Thus the example of `alldiff` emphasizes that a *global constraint* is not only able to reduce the number of posted constraints (one `alldiff` constraint for n variables replaces $\mathcal{O}\left(n^2\right)$ binary constraints) but also to impose a more high-level view on the problem variables due to a larger scope. Hence, global constraints can increase the efficiency of a constraint solver by decreasing the number of constraints and increasing the strength of the propagation that is performed on the domains of the problem variables.

3 Propagation Algorithms

The previous section 2.3.4 gave a trace, why global constraints are able to increase the efficiency of CP -systems. In the course of this section we take a glance at advanced propagation algorithms for such global constraints using the *Gecode* library. The global constraints we look at in this context are the `sortedness` constraint and the `global cardinality` constraint. Furthermore this section provides more detailed insight into the propagation algorithms for these global constraints and explains the theory behind them.

3.1 Definitions and Conventions

The explanations in the subsequent section use the following graph theoretical definitions and further notational conventions.

3.1.1 Graph definitions

For a better understanding of the theoretical framework underneath each of the following constraints, this section recapitulates some basic definitions from graph theory. The most important notions the propagation algorithms in this section make use of are a *matching*, an *alternating path* and a *strongly connected component*.

Definition 14 (Matching). *Given an undirected graph $G := \langle V, E \subseteq V^2 \rangle$ a vertex $v \in V$ is* incident *with an edge $e \in E$, denoted as $\mathtt{inc}(v, e)$, if there is a node $x \in V$ such that $e = (x, v)$. x and v are the end-vertices of e. $M \subseteq E$ is called a* matching *of G if*

$$\forall e, e' \in M : \neg \exists v \in V : \mathtt{inc}(v, e) \wedge \mathtt{inc}(v, e')$$

A vertex $v \in V$ is matched (in M) *if $\exists e \in M : \mathtt{inc}(v, e)$ and we call an edge $e \in E$* matched (in M) *or* matching *if $e \in M$. Edges $e \notin M$ and vertices $v \in V : \neg \exists e \in M : inc(v, e)$ are called* free. *Furthermore a matching M is called:*

$$maximal \Leftrightarrow \neg \exists M' \subseteq E : M \subset M'$$
$$maximum \Leftrightarrow \forall M' \subseteq E : |M| \geq \left|M'\right|$$
$$perfect \Leftrightarrow |M| = \frac{|V|}{2}$$

and we write $\mathtt{max}^{\subset}(M, G)$ if M is a maximal matching on G with respect to set inclusion, $\mathtt{max}^{|\,|}(M, G)$ if M is a maximum cardinality matching on G and $\mathtt{pm}(M, G)$ if M is a perfect matching on G. Thus, given a matching M on G the following implications hold:

$$\mathtt{pm}(M, G) \Rightarrow \mathtt{max}^{|\,|}(M, G)$$
$$\mathtt{max}^{|\,|}(M, G) \Rightarrow \mathtt{max}^{\subset}(M, G)$$

Definition 15 (Path [7, App. B]). *Given an undirected graph $G := \langle V, E \subseteq V^2 \rangle$ a* path *p in G from vertex u to vertex w, denoted as $u \bullet\!\!-\!\!\bullet\ w$, is a sequence of vertices $p = \langle v_0, \ldots, v_k \rangle$ of length $|p| = k$, such that $v_0 = u$ and $v_k = w$. By abuse of notation we also use the notion of a* path *as the set of its edges $p = \{(v_i, v_j) \in E \mid 0 \leq i < j \leq k\}$. If we are given a directed graph $\overrightarrow{G}$ we denote a* path *p from u to w as $u \rightsquigarrow w$.*

Definition 16 (Alternating path). *Given an undirected graph $G := \langle V, E \subseteq V^2 \rangle$ and a matching M on G an* alternating path *$\widehat{p} = v_0 \bullet\!\!-\!\!\bullet\ v_k$ in G is a path p whose edges $e_i = (v_i, v_{i+1}) \in E, 0 \leq i < k$ are alternately matched and free. If $|p|$ is even, p is an* even alternating path *and we write $\widetilde{p}$. If $v_k = v_o$, p is an* alternating cycle *and we write $p^{\circlearrowleft}$. If also $|p|$ is even then we call p an* even alternating cycle.

Definition 17 (Augmenting path). *Given an undirected graph $G := \langle V, E \subseteq V^2 \rangle$ and a matching M on G, an* augmenting path *is a path $p = v_0 \bullet\!\!-\!\!\bullet\ v_k$ in G of length $|p| = k$ whose edges are alternately matched and free such that $e_0 = (v_0, v_1)$ and $e_{k-1} = (v_{k-1}, v_k)$ are free.*

Definition 18 (Strongly connected component). *Given a directed graph $\overrightarrow{G} := \langle V, E \rangle$ two vertices $u, v \in V$ are* strongly connected*, denoted as $u \bowtie v$, if there are two paths $p = u \rightsquigarrow v$ and $p' = v \rightsquigarrow u$ in $\overrightarrow{G}$. Further, a* strongly connected component *S of $\overrightarrow{G}$, denoted as $\mathsf{SCC}(S, \overrightarrow{G})$, is a maximal subset $S \subseteq V$ with $S = \{u, v \in V \mid u \bowtie v\}$. Additionally, we also refer to an SCC S as the set of its edges $\{e \in E \mid u, v \in S \wedge (e \in p = u \rightsquigarrow v \vee e \in p' = v \rightsquigarrow u)\}$ used by two paths p and p' in $\overrightarrow{G}$ to join $u, v \in S$.*

3.1.2 Further Definitions

Apart from the graph theoretical definitions above we also introduce the following definitions making the explanation of the propagation algorithms for `sortedness` and `global cardinality` more convenient.

Definition 19 (Set Definitions). *Given some set S and two sets $A \subset S$ and $B \subset S$ we define the* symmetric difference *$A \oplus B \overset{def.}{=} (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$. If $S = A \cup B$ and $A \cap B = \emptyset$ we write $S = A \uplus B$ stating that A and B are disjoint partitions of S.*

Definition 20 (Notational Conventions).

1. *If we are given a finite set of FDVars $\mathcal{X}$* we denote the cardinality of $\mathcal{X}$ as $n \overset{def.}{=} |\mathcal{X}|$.
2. Given a finite set $\mathcal{X}$ of FDVars we define the union of the domains of all variables in $\mathcal{X}$ as $\mathcal{D}(\mathcal{X}) = \bigcup_{i \in \mathbb{K}_n} D_i$
3. $\mathbb{K}_n \overset{def.}{=} \{0, \ldots, n-1\}$
4. $\Pi_n \overset{def.}{=} \{\mathbb{K}_n \mapsto \mathbb{K}_n\}$
5. Given a range $H \overset{def.}{=} [a..b] \overset{def.}{=} \{a, \ldots, b\} \subset \mathbb{Z}$ we define:
   - $\underline{H} \overset{def.}{=} \min(H) = a$ as the lower bound of H
   - $\overline{H} \overset{def.}{=} \max(H) = b$ as the upper bound of H

   Note that a range is also referred to as an interval.
6. Given a constraint C defined on $\mathcal{X} \overset{def.}{=} \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$ we use $x_i : D_i$ to refer to a single variable in $\mathcal{X}$ and $\mathcal{X}$ to refer to the whole set of variables.
7. Given a finite set $\mathcal{X}$ of FDVars, a variable assignment $\alpha \in \mathtt{ass}(\mathcal{X})$ and an integer value $v \in \mathbb{Z}$ we define the number of times α assigns the value v to a variable $x_i : D_i \in \mathcal{X}$ as

$$\#(\alpha, v) \overset{def.}{=} \left|\{x_i \in \mathcal{X} | \alpha(x_i) = v\}\right|$$

3.2 Sortedness

The implementation of the `sortedness` constraint in *Gecode* presented in this section stems from section 3.1, p. 40-59 in the doctoral dissertation of *Sven Thiel: Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions* [39]. Before we discuss the details of the propagation algorithm, we first take a look at the constraint's denotational semantics:

Definition 21 (Sortedness). *Given two finite sets of FDVars $\mathcal{X} = \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$, and $\mathcal{Y} = \{y_0 : E_0, \ldots, y_{n-1} : E_{n-1}\}$ we define:*

$$\begin{aligned}\texttt{sortedness}\,(\mathcal{X},\mathcal{Y}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{bnd}}(\mathcal{S})|\ &\alpha(y_0) \leq \ldots \leq \alpha(y_{n-1}) \\ \wedge\ &\exists \pi \in \Pi_n : \alpha(x_i) = \alpha(y_{\pi(i)})\}\end{aligned}$$

Hence, `sortedness` *is a bounds consistent constraint with a scope of $2 \cdot n$ variables $x_0 : D_0, \ldots, x_{n-1} : D_{n-1}$ and $y_0 : E_0, \ldots, y_{n-1} : E_{n-1}$ checking whether it is possible to sort the $x_i : D_i$ in non-decreasing order, such that*

$$\texttt{sort}(x_0, \ldots, x_{n-1}) = (y_0, \ldots, y_{n-1})$$

Thus, `sortedness` *checks whether there exists a sorting permutation $\pi \in \Pi_n$ such that $\forall i \in \mathbb{K}_n : x_i = y_{\pi(i)}$ holds. Provided the following finite sets of FDVars the table below gives some examples for the* `sortedness` *constraint:*

$$\begin{aligned}\mathcal{X}_1 &= \{x_0 : \{1\},\ x_1 : \{2\},\ x_2 : \{5\},\ x_3 : \{9\},\ x_4 : \{2\}\} \\ \mathcal{X}_2 &= \{x_0 : \{9\},\ x_1 : \{5\},\ x_2 : \{2\},\ x_3 : \{1\},\ x_4 : \{2\}\} \\ \mathcal{Y}_3 &= \{y_0 : \{9\},\ y_1 : \{5\},\ y_2 : \{2\},\ y_3 : \{2\},\ y_4 : \{1\}\} \\ \mathcal{Y}_1 &= \{y_0 : \{1\},\ y_1 : \{2\},\ y_2 : \{2\},\ y_3 : \{5\},\ y_4 : \{9\}\} \\ \mathcal{Y}_2 &= \{y_0 : \{1\},\ y_1 : \{2\},\ y_2 : \{2\},\ y_3 : \{7\},\ y_4 : \{9\}\}\end{aligned}$$

`sortedness` $(\mathcal{X}_1, \mathcal{Y}_1)$	✓	correct
`sortedness` $(\mathcal{X}_1, \mathcal{Y}_2)$	$\lightning$	values 5 and 7 do not match
`sortedness` $(\mathcal{X}_2, \mathcal{Y}_3)$	$\lightning$	y-variables (9,5,2,2,1) are not sorted in non-decreasing order

Table 1. Examples for `sortedness`

3.2.1 Theoretical Background

After the formal definition of the `sortedness` constraint, we now focus on the theoretical background for the bounds consistent global propagation algorithm implementing `sortedness` . As Sven Thiel stresses in his doctoral dissertation (cf. 3.1.3, p. 55 [39]), the propagation algorithm splits up in 5 steps (footnotes denote the file in the *Gecode* architecture where they are implemented):

[S$_1$] - Normalization

Normalization[1] adjusts the domain bounds $\underline{E}_i$ and $\overline{E}_i$ of the $y_i : E_i$ variables, such that

$$\forall i \in \mathbb{K}_n : \underline{E}_i \leq \underline{E}_{i+1} \wedge \overline{E}_i \leq \overline{E}_{i+1}$$

holds. If so, the domains E_i are called *normalized*. Additionally, we can also infer that

$$\forall i \in \mathbb{K}_n : \underline{D}_i \geq \underline{E}_0 \wedge \overline{D}_i \leq \overline{E}_{n-1}$$

holds. The complexity of normalization is

$$\mathcal{T}(S_1) = \Theta(n)$$

[1] `sortedness/order.icc`

[S₂] - Sorting
The second step[1] of the propagation algorithm creates sorting permutations $\sigma, \tau \in \Pi_n$, such that

$$\underline{D}_{\sigma(0)} \leq \ldots \leq \underline{D}_{\sigma(n-1)} \wedge \overline{D}_{\tau(0)} \leq \ldots \leq \overline{D}_{\tau(n-1)}$$

holds. Thus, we sort the variables $x_i : D_i \in \mathcal{X}$ according to their lower and upper domain bounds. In order to create π and τ we apply an optimized quicksort algorithm[2] using an non-recursive stack version and insertion sort for subsequences with length $l \leq 20$. Hence, the worst case complexity is

$$\mathcal{T}(S_2) = \mathcal{O}(n \cdot log(n))$$

[S₃] - Matching
As this step[3] uses a specific graph structure, we extend the graph definitions given above as follows:

Definition 22 (Bipartite Graph [7, p.1083]). *An undirected graph $G := \langle V, E \rangle$ is called* bipartite *if its node set V can be partitioned into two sets V_1 and V_2 such that $V = V_1 \uplus V_2 \Leftrightarrow V = V_1 \cup V_2 \wedge V_1 \cap V_2 = \emptyset$ and $E \subset V_1 \times V_2$ holds.*

Definition 23 (Intersection graph [39, p. 43]). *The undirected bipartite graph $G := \langle \mathcal{X} \uplus \mathcal{Y}, \mathcal{E} \rangle$, where $\mathcal{E} = \{(x_i, y_j) | D_i \cap E_j \neq \emptyset\}$ is called the* intersection-graph.
Given an intersection-graph *$G := \langle \mathcal{X} \uplus \mathcal{Y}, \mathcal{E} \rangle$ as defined above, $G' := \langle \mathcal{X} \uplus \mathcal{Y}, \mathcal{E}' \rangle$ is a* reduced intersection-graph *if $\mathcal{E}' = \mathcal{E} \setminus \{e \in E | \forall M : \mathtt{pm}(M, G) \Rightarrow e \notin M\}$*

Apart from being bipartite, the intersection graph has another essential property following directly from normality of $\mathcal{Y}$ achieved in $[S_1]$:

Definition 24 (Convexity [13]). *Let $G := \langle \mathcal{X} \cup \mathcal{Y}, \mathcal{E} \rangle$ be an intersection graph as defined in* definition 17. *G is called* convex *if*

$$(x_i, y_l) \in \mathcal{E} \wedge (x_i, y_r) \in \mathcal{E} \Rightarrow \forall s \in [l, r] : (x_i, y_s) \in \mathcal{E}$$

Since by $[S_1]$ the intersection graph is also convex, we use Glover's Algorithm [39, 13] to compute a perfect matching on the intersection graph. Let $G := \langle \mathcal{V}, \mathcal{E} \rangle$ be an intersection graph for $\mathcal{X}$ and $\mathcal{Y}$ as defined above. Glover's algorithm then computes for every y_j-node the set $N := \{x_i | (x_i, y_j) \in \mathcal{E}\}$ of free x_i-nodes reachable from y_j and matches y_j with the x_i for which $\overline{D}_i$ is minimal. The algorithm selects the candidate with minimal $\overline{D}_i$ because any other candidate x_c with $\overline{D}_i < \overline{D}_c$ can match y_i-nodes x_i is not able to. Hence x_c is kept for further processing as shown in figure 1. As Thiel proposes, this matching problem

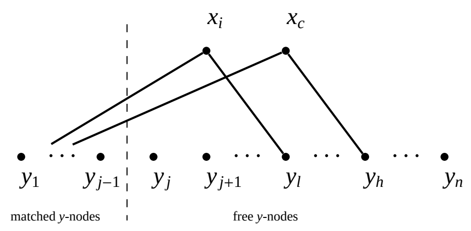

Fig. 1. candidate choice

[2] `support/sort.hh`
[3] `sortedness/matching.icc`

is handled as a *union-find problem* that itself represents an instance of the *Offline-min problem* (cf. p.139-141 in [1]). Mimicking the behavior of a priority queue we solve this problem using a tree structure as described in [1, p.129ff]. Processing $\mathcal{O}(n)$ `union` and `find` operations as it is the case with our problem of computing the perfect matching on G is in $\mathcal{O}\left(n^2\right)$. Using a weighted `union` operation the tree structure is kept balanced and we achieve a complexity of $\mathcal{O}(n \cdot log(n))$. Using a further optimization as presented in [1], so-called path compression yields an amortized running time of $\mathcal{O}(n \cdot G(n))$, relying on the subsequent definitions for the functions F, and G:

$$F(i) = \begin{cases} 1 & \text{if } i = 0 \\ 2^{F(i-1)} & \text{if } i > 0 \end{cases}$$

$$G(n) := \min\{k \in \mathbb{N} | F(k) \geq n\}$$

Thus, this step's complexity results in

$$\mathcal{T}(M_2) = \mathcal{O}(n \cdot G(n))$$

[S$_4$] - Strongly Connected Components

The major insight of this propagation algorithm for `sortedness` is the following correspondence:

$\alpha \Rightarrow M$: From valuations to matchings

$$\begin{aligned} &\forall \alpha \in \mathtt{val}(\mathcal{S}) : \alpha = \{x_i \mapsto d_i, \ldots, y_j \mapsto e_j\} \in \mathtt{sat}(\mathtt{sortedness}) \\ \Rightarrow\ &\exists \pi \in \Pi_n\ \forall i \in \mathbb{K}_n : d_i = e_{\pi(i)} \\ \Rightarrow\ &\exists M \subseteq \mathcal{E} : \mathtt{pm}(M, G) \wedge M := \{(x_i, y_{\pi(i)}) | i \in \mathbb{K}_n\} \end{aligned}$$

$M \Rightarrow \alpha$: From matchings to valuations

$$\begin{aligned} &\forall M \subseteq \mathcal{E} : \mathtt{pm}(M, G) \\ &\exists \pi \in \Pi_n : M := \{(x_i : D_i, y_{\pi(i)} : E_{\pi(i)}) | i \in \mathbb{K}_n\} \\ &\wedge \forall i \in \mathbb{K}_n : d_i \in (D_i \cap E_{\pi(i)}) \\ \overset{E_j' snormalized}{\Rightarrow}\ &\exists \alpha \in \mathtt{sat}(\mathtt{sortedness}) : \alpha = \{x_i \mapsto d_i, \ldots, y_j \mapsto e_j\} \end{aligned}$$

Hence we have $\alpha \Leftrightarrow M$ stating that every solution $\alpha \in \mathtt{sat}(\mathtt{sortedness})$ corresponds to a perfect matching M on the intersection graph. Thus the algorithm has to consider every edge taking part in a possible perfect matching on the graph. Instead of computing all possible matchings the following corollary shows, why the computation of a single matching suffices in order to obtain every edge belonging to some matching in G.

Corollary 1 (Maximum Matching [see 5, 26, chp. 7 chp. 4]) *An edge belongs to some of but not all maximum matchings, iff, 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.*

As $[S_3]$ already computes a perfect matching M on G and since for both partitions $\mathcal{X}$ and $\mathcal{Y}$ of the intersection graph $|\mathcal{X}| = |\mathcal{Y}| = n$ holds, there are no free nodes in the graph after $[S_3]$. Thus, as every perfect matching is also a maximum matching, corollary 1 tells us that deciding whether an edge $e \in E$ belongs to some perfect matching M on G or not is equivalent to the decision whether e belongs to a strongly connected component of G. In order to compute strongly connected components G is transformed into a DAG we refer to as *oriented intersection graph* $\overrightarrow{G}$ that is defined as follows:

Definition 25 (Oriented intersection graph (cf. Lemma 3.3 p.48 [39]). *Given a perfect matching M in the intersection graph G (cf. definition 23), the oriented intersection graph is obtained by directing every edge $e \notin M$ from the $\mathcal{X}$ to the $\mathcal{Y}$ partition and adding the reverse edge for every edge $e \in M$.*

In the case of this bipartite oriented intersection graph $\overrightarrow{G}$ the computation of strongly connected components coincides with the computation of the even alternating cycles as emphasized in corollary 1. The fourth step, the computation of the SCCs [4] in $\overrightarrow{G}$ is implemented according to the algorithm 3.2 on p.50 [39]. Thiel adapted the algorithm by Cheriyan and Mehlhorn to the special structure of the oriented intersection graph. Regarding the implementation, a component C is represented as a list of y-indices $C = [i_0, \ldots, i_{k-1}]$, where y_{i_0} is the leftmost and $y_{i_{k-1}}$ is the rightmost y-node in the component. Due to normalization we have $\underline{E}_{i_0} \leq \ldots \leq \underline{E}_{i_{k-1}}$ and $\overline{E}_{i_0} \leq \ldots \leq \overline{E}_{i_{k.1}}$.

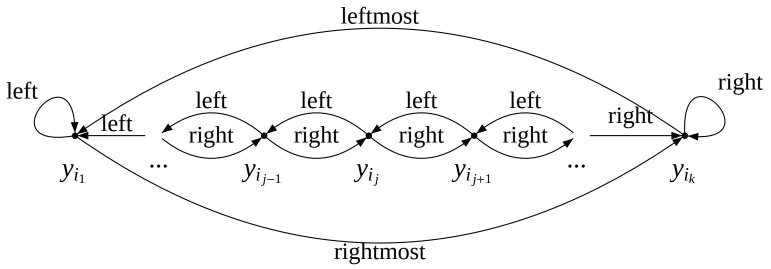

Fig. 2. SCC-representation

Finally, this step not only computes the SCCs of $\overrightarrow{G}$, but also labels each $\mathcal{X}$-variable with the number of the SCC it is contained in. The labeling is performed in $\mathcal{O}(n)$. As not only the labeling is linear but also the SCC computation [39] this step's overall complexity is

$$\mathcal{T}(S_4) = \mathcal{O}(n + n) = \mathcal{O}(n)$$

[S$_5$] - Domain Reduction
In order to explain how the variable domains are narrowed in this step[5], we have to define the *reduced intersection graph* as follows:

Definition 26 (reduced intersection graph (cf. p.44 [39])). *Given an intersection graph G (cf. definition 23) we obtain the reduced intersection graph by removing all edges $e \in E$ that do not belong to any perfect matching M on G, i.e. we remove every edge neither contained in the matching computed in* $[S_3]$ *nor in any SCC computed in* $[S_4]$.

By definition of the oriented intersection graph (cf. definition 25) the smallest SCC s that may occur is $s = (y_j, x_{\phi(j)})$ consisting of the single pair formed by an y_j-variable and its matching mate $x_{\phi(j)}$.

[S$_{5.1}$] - Narrowing $\mathcal{X}$
Let G^- denote :the reduced intersection graph for $\mathcal{X}$and $\mathcal{Y}$and $S_i \subseteq D_i$ the reduced domain of x_i. By construction, G^- only contains edges $\{x_i, y_j\}$ belonging to a perfect matching M

[4] `sortedness/narrowing.icc`
[5] `sortedness/narrowing.icc`

on G corresponding to an assignment α. Thus S_i is computed as $S_i = D_i \cap \bigcup_{\{x_i,y_j\}\in G^-} E_j$ (cf. p. 44 corollary 3.1 [39]) and we infer:

$$\textsc{Narrow-}\mathcal{X}\textsc{-LB}\ \frac{lm_i := \min\{j \in \mathbb{K}_n | \underline{E}_j \leq \underline{D}_i \leq \overline{E}_j\} \qquad y_{lm_i} \bowtie x_i}{\underline{S}_i \geq \underline{E}_{lm_i}}$$

$$\textsc{Narrow-}\mathcal{X}\textsc{-UB}\ \frac{rm_i := \max\{j \in \mathbb{K}_n | \underline{E}_j \leq \overline{D}_i \leq \overline{E}_j\} \qquad y_{rm_i} \bowtie x_i}{\overline{S}_i \leq \overline{E}_{rm_i}}$$

where E_{lm_i} is the domain of the leftmost y_{lm_i}-variable and E_{rm_i} is the domain of the rightmost y_{rm_i}-variable, x_i can reach in its SCC. By step $[S_4]$, the x-variables are labeled with the number of their corresponding SCC. According to algorithm 3.3. on p. 53 in [39] we merge the sequence $\overline{E}_{i_0}, \ldots, \overline{E}_{i_{k-1}}$ with the sequence $\underline{D}_{\sigma(\phi(i_0))}, \ldots, \underline{D}_{\sigma(\phi(i_{k-1}))}$, such that

$$\begin{aligned} \ldots \overline{E}_s \underline{D}_t \ldots &\Leftrightarrow \overline{E}_s \cap \underline{D}_t = \emptyset \\ \ldots \underline{D}_t \overline{E}_s \ldots &\Leftrightarrow \overline{E}_s \cap \underline{D}_t \neq \emptyset \end{aligned}$$

Hence, we can perform domain reduction for the lower bounds of all $\mathcal{X}$-variables in at most $\mathcal{O}(k \cdot (n/k)) = \mathcal{O}(n)$ time, where k is the size of the largest SCC in $\overrightarrow{G}$. We achieve this linear complexity by merging the sequences $\underline{E}_{i_0}, \ldots, \underline{E}_{i_{k-1}}$ with the sequence $\overline{D}_{\sigma(\phi(i_0))}, \ldots, \overline{D}_{\sigma(\phi(i_{k-1}))}$ in order to narrow the upper bounds of the x-variables, such that

$$\begin{aligned} \ldots \overline{D}_t \underline{E}_s \ldots &\Leftrightarrow \underline{E}_s \cap \overline{D}_t = \emptyset \\ \ldots \underline{E}_s \overline{D}_t \ldots &\Leftrightarrow \overline{E}_s \cap \underline{D}_t \neq \emptyset \end{aligned}$$

[S$_{5.2}$] - Narrowing $\mathcal{Y}$

Due to their normalization, domain reduction for $\mathcal{Y}$ is less work. Let $T_j \subseteq E_j$ denote the reduced domain of y_j. Then we obtain the new upper bounds $\overline{T}_j$ for free from the perfect matching ϕ that has already been computed in step $[S_3]$. Since ϕ matches the y_j with those $x_{\phi(j)}$-candidates for which $\overline{D}_{\phi(j)}$ is minimal, we compute the new upper bound of y_j by

$$\textsc{Narrow-}\mathcal{Y}\textsc{-LB}\ \frac{i = \phi(j)}{\overline{T}_j \leq \overline{D}_i}$$

Performing the same domain reduction on the lower bounds requires a new matching ψ, where ψ matches the y_j-variables from n down to 1 and always chooses the neighbor x_i from $N(y_j)$ with maximal $\underline{D}_i$. Thus we can reduce the lower bounds of the domains E_j symmetrically by setting

$$\textsc{Narrow-}\mathcal{Y}\textsc{-UB}\ \frac{i = \psi(j)}{\underline{T}_j \geq \underline{D}_i}$$

Since the matching computation $[S_3]$ is in $\mathcal{O}(n)$ and since the update for every y_j is in $\mathcal{O}(1)$ the domain reduction for the y_j-variables is performed in $\Theta(n)$. Hence, the overall complexity for the narrowing step is

$$\mathcal{T}(S_5) = \Theta(n)$$

Total complexity of `sortedness`

Summing up the complexity for the different steps, we get the following result for the overall complexity of the `sortedness` constraint:

As table 2 shows, the complexity of the `sortedness` constraint depends on the time needed to generate the sorting permutations σ and τ for $\mathcal{X}$ in step $[S_2]$.

$\mathcal{T}(S_1)$	$= \Theta(n)$
$\mathcal{T}(S_2)$	$= \mathcal{O}(n \cdot log(n))$
$\mathcal{T}(S_3)$	$= \mathcal{O}(n \cdot G(n))$
$\mathcal{T}(S_4)$	$= \mathcal{O}(n)$
$\mathcal{T}(S_5)$	$= \Theta(n)$

$\mathcal{T}(\texttt{sortedness}) = \mathcal{O}(n \cdot log(n))$

Table 2. complexity-`sortedness`

3.2.2 Extended Sortedness

In the previous sections we have discussed the theoretical background of a global propagation algorithm for the `sortedness` constraint as described in [39]. As Thiel points out in his doctoral dissertation there is a straightforward approach for extending the `sortedness` constraint to explicit permutation variables $\mathcal{Z}$ as a third argument. Nevertheless, this approach does neither achieve bounds consistency on $\mathcal{X}$ nor on the permutation variables $\mathcal{Z}$. Motivated by Thiel's comparison with related work [39, sec. 3.1] we introduce the extended version of `sortedness` as `sortedness`$^+$ and define it as follows:

Definition 27 (Sortedness$^+$). *Given finite sets of FDVars* $\mathcal{X} \overset{def.}{=} \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$, $\mathcal{Y} \overset{def.}{=} \{y_0 : E_0, \ldots, y_{n-1} : E_{n-1}\}$ *and* $\mathcal{Z} \overset{def.}{=} \{z_o : F_o, \ldots, z_{n-1} : F_{n-1}\}$ *and let* $\mathcal{S} \overset{def.}{=} \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$. *Then*

$$\texttt{sortedness}^+(\mathcal{X}, \mathcal{Y}, \mathcal{Z}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{bnd}}(\mathcal{S}) | \alpha(y_0) \leq \ldots \leq \alpha(y_{n-1}) \wedge \alpha(x_i) = \alpha(y_{\alpha(z_i)})\} \cap \texttt{alldiff}(\mathcal{Z})$$

Hence, `sortedness`$^+$ *now has an extended scope of n extra variables in* $\mathcal{Z}$. These variables are called permutation variables because $\mathcal{Z}$ explicitly implements the sorting permutation $\pi \in \Pi_n$ such that $\forall i \in \mathbb{K}_n : x_i = y_{\pi(i)}$ holds. Thus π does no longer only exist on the denotational level of the propagator but is made accessible through $\mathcal{Z}$ which can be used to perform inferences on $\mathcal{X}$ and $\mathcal{Y}$. Reconsider the finite sets of FDVars from table 1 and the following additional finite sets of FDVars for the permutation variables. Then the following table presents some examples for the `sortedness`$^+$ constraint:

$\mathcal{X}_1 =$	$\{x_0 : \{1\},$	$x_1 : \{2\},$	$x_2 : \{5\},$	$x_3 : \{9\},$	$x_4 : \{2\}\}$
$\mathcal{X}_2 =$	$\{x_0 : \{9\},$	$x_1 : \{5\},$	$x_2 : \{2\},$	$x_3 : \{1\},$	$x_4 : \{2\}\}$
$\mathcal{Y}_3 =$	$\{y_0 : \{9\},$	$y_1 : \{5\},$	$y_2 : \{2\},$	$y_3 : \{2\},$	$y_4 : \{1\}\}$
$\mathcal{Y}_1 =$	$\{y_0 : \{1\},$	$y_1 : \{2\},$	$y_2 : \{2\},$	$y_3 : \{5\},$	$y_4 : \{9\}\}$
$\mathcal{Y}_2 =$	$\{y_0 : \{1\},$	$y_1 : \{2\},$	$y_2 : \{2\},$	$y_3 : \{7\},$	$y_4 : \{9\}\}$
$\mathcal{Z}_1 =$	$\{z_0 : \{1\},$	$z_1 : \{2\},$	$z_2 : \{4\},$	$z_3 : \{5\},$	$z_4 : \{3\}\}$
$\mathcal{Z}_2 =$	$\{z_0 : \{1\},$	$z_1 : \{2\},$	$z_2 : \emptyset,$	$z_3 : \{5\},$	$z_4 : \{3\}\}$
$\mathcal{Z}_3 =$	$\{z_0 : \{1\},$	$z_1 : \{2\},$	$z_2 : \{3\},$	$z_3 : \{5\},$	$z_4 : \{4\}\}$
$\mathcal{Z}_4 =$	$\{z_0 : \{4, 5\},$	$z_1 : \{1\},$	$z_2 : \{4\},$	$z_3 : \{5\},$	$z_4 : \{3\}\}$

`sortedness`$^+(\mathcal{X}_1, \mathcal{Y}_1, \mathcal{Z}_1)$	✓	correct
`sortedness`$^+(\mathcal{X}_1, \mathcal{Y}_2, \mathcal{Z}_2)$	$\lightning$	values 5 and 7 do not match
`sortedness`$^+(\mathcal{X}_2, \mathcal{Y}_3, \mathcal{Z}_3)$	$\lightning$	$\mathcal{Y}_3$ not sorted in non-decreasing order
`sortedness`$^+(\mathcal{X}_1, \mathcal{Y}_1, \mathcal{Z}_4)$	$\lightning$	$\mathcal{Z}_4$ is no sorting permutation

Table 3. Examples for `sortedness`$^+$

In order to achieve bounds consistency on all variables in $\mathcal{S}$ we introduce additional computation steps $[S_{2.1}]$ and $[S_{5.3}]$ to the 5 steps above. Furthermore we extend step $[S_{5.1}]$ by the following two inferences for the reduced domains U_i of the permutation variables $z_i : F_i \supseteq U_i$:

$$\frac{lm_i := \min\{j \in \mathbb{K}_n | \underline{E}_j \leq \underline{D}_i \leq \overline{E}_j\} \qquad y_{lm_i} \bowtie x_i}{\underline{U}_i \geq lm_i}$$

$$\frac{rm_i := \max\{j \in \mathbb{K}_n | \underline{E}_j \leq \overline{D}_i \leq \overline{E}_j\} \qquad y_{rm_i} \bowtie D_i}{\overline{U}_i \leq rm_i}$$

[S2.1] - Connecting $\mathcal{X}$ and $\mathcal{Y}$

The first additional step[6] $[S_{2.1}]$ correlates the permutation variables in $\mathcal{Z}$ with the problem variables in $\mathcal{X}$ and $\mathcal{Y}$. It not only connects $\mathcal{X}$ with $\mathcal{Y}$ but also ensures that the domain bounds of all variables are consistent at the start of the propagation algorithm. Consider a permutation variable $z_i : F_i = [l..r]$ connecting the corresponding problem variable $x_i : D_i$ to possible matching mates from $y_l : E_l$ to $y_r : E_r$. At first this step ensures that the permutation variables only connect the $\mathcal{X}$ variables with feasible $\mathcal{Y}$ variables by ensuring that $\forall i \in \mathbb{K}_n$:

$$\textsc{ValidPerm-LB}\ \frac{l := \min\{j \in \mathbb{K}_n | \underline{E}_j \leq \underline{D}_i \leq \overline{E}_j\}}{\underline{D}_i \geq \underline{E}_l \qquad \underline{F}_i \geq l}$$

$$\textsc{ValidPerm-UB}\ \frac{r := \max\{j \in \mathbb{K}_n | \underline{E}_j \leq \overline{D}_i \leq \overline{E}_j\}}{\overline{D}_i \leq \overline{E}_r \qquad \overline{F}_i \leq r}$$

In case that $z_i : F_i = \{p\}$ is already determined we can perform the following inferences $\forall i \in \mathbb{K}_n$:

$$\frac{D_i = \{v\}}{T_p := \{v\}} \qquad \frac{E_p = \{v\}}{S_i := \{v\}}$$

$$\frac{|D_i| > 1 \qquad nl := \max(\underline{D}_i, \underline{E}_p) \qquad nr := \min(\overline{D}_i, \overline{E}_p)}{\underline{D}_i \geq nl \qquad \overline{D}_i \leq nr \qquad \underline{E}_p \geq nl \qquad \overline{E}_p \leq nr}$$

Otherwise, if $|F_i| > 1$, we cannot do any better than inferring:

$$\frac{}{\underline{D}_i \geq \underline{E}_l \qquad \overline{D}_i \leq \overline{E}_r}$$

These inferences are applied such that the complexity for this step is

$$\mathcal{T}(S_{2.1}) = \Theta(n)$$

[S5.3] - Crossing Edges

In contrast to the consistency check above step $[S_{5.3}]$ is based on the following observation: As discussed in $[S_4]$, the major insight into the propagation algorithm for `sortedness` is the link between $\mathcal{X}$ and $\mathcal{Y}$ via a sorting permutation $\pi \in \Pi_n$. Furthermore the SCC computation in $[S_4]$ showed that there is a one-to-one correspondence between permutations π and perfect matchings M on the *intersection-graph* $G := \langle \mathcal{X} \uplus \mathcal{Y}, \mathcal{E} \rangle$. Let U_i denote the reduced domain of a permutation variable $z_i : F_i$, such that $z_i : U_i \subseteq F_i$ holds. Hence we have the following information about the permutation variables in $\mathcal{Z}$:

$$\forall i \in \mathbb{K}_n : U_i = \{k \in \mathbb{K}_n | D_i \cap E_k \neq \emptyset\}$$

6 `sortedness/sortedness.icc`

Thus the reduced domain U_i of a permutation variable $z_i : F_i$ contains exactly the indices of those $y_k : E_k$ being a matching mate for $x_i : D_i$ in a perfect matching M on G. Furthermore $\underline{U}_i := l$ is the index of the leftmost reachable $\mathcal{Y}$-node y_l of x_i in G that belongs to a perfect matching M on G and $\overline{U}_i := r$ is the index of the rightmost reachable $\mathcal{Y}$-node y_r of x_i belonging to such a perfect matching. Thus we can shrink F_i to $U_i := [l..r]$. Now the question arises, why setting $\underline{U}_i := l$ and $\overline{U}_i := r$ does not suffice to achieve bounds consistency on $\mathcal{X}$ and $\mathcal{Z}$?
As permutations π encode all perfect matchings M between $\mathcal{X}$ and $\mathcal{Y}$ on G they also encode those matchings M', such that:

$$M' \Leftrightarrow \beta \in \mathtt{ass}(\mathcal{S}) \wedge \Big(\exists i, j \in \mathbb{K}_n : i < j \wedge \beta(y_i) > \beta(y_j)\Big)$$

which is an obvious contradiction to the definition of $\mathtt{sortedness}^+$ where we require $\forall \alpha \in \mathtt{sortedness}^+ : \alpha(y_0) \leq \ldots \leq \alpha(y_{n-1})$. From step $[S_2]$ we are given sorting permutations σ and τ, such that $\underline{D}_{\sigma(0)} \leq \ldots \leq \underline{D}_{\sigma(n-1)} \wedge \overline{D}_{\tau(0)} \leq \ldots \leq \overline{D}_{\tau(n-1)}$. After the narrowing of the $\mathcal{X}$-domains $[S_{5.1}]$ the bounds of $\mathcal{Z}$ variables possibly encode a matching $M \Leftrightarrow \beta \notin \mathtt{sat}(\mathtt{sortedness}^+)$ containing *crossing edges* violating the definition of $\mathtt{sortedness}^+$. This violation can occur on both, the lower and the upper bounds of variables in $\mathcal{Z}$. Nonetheless, if those crossing edges belong to an SCC that we just computed in step $[S_4]$ the propagation algorithm must not delete them as they take part in some solution to the matching problem. The decision whether to prune such an edge or not is done by checking whether the permutation variable $z_i : U_i = [l..r]$ on a crossing edge coincides with the indices of the leftmost and rightmost reachable variables belonging to the same SCC that $x_i : S_i$ belongs to. In the formula below we denote these indices with lm_i for the index of the leftmost variable and rm_i respectively. If these indices do coincide with the bounds of the permutation variable l and r these bounds do take part in some solution to the constraint. Otherwise, we can exclude them as inconsistent as they neither belong to any SCC nor, by definition of the Glover algorithm (3.2.1), belong to the perfect matching computed in step $[S_3]$. Thus taking possible SCCs into account we can formalize a *crossing edge* on the lower bound of a variable $z_i : U_i \subseteq F_i \in \mathcal{Z}$ as follows (a *crossing edge* on the upper bound is specified symmetrically):

$$p := \sigma(i) \wedge q := \sigma(j) \wedge p < q = p + 1 \wedge \underline{D}_p < \underline{D}_q \tag{1}$$
$$U_p = [l_0..r_0] \wedge U_q = [l_1..r_1] \wedge 0 < l_1 < l_0 \wedge lm_q \neq l_1 < r_1 \tag{2}$$
$$U_i = \{k \in \mathbb{K}_n | D_i \cap E_k \neq \emptyset\} \tag{3}$$
$$(2)\&(3) \Rightarrow \forall k \in \{0, \ldots, l_0 - 1\} : D_p \cap E_k = \emptyset \tag{4}$$
$$\Leftrightarrow \Big(\forall k \in \{0, \ldots, l_0 - 1\} : \overline{E}_k < \underline{D}_p \overset{(1)}{<} \underline{D}_q\Big) \tag{5}$$
$$\Rightarrow \overline{E}_{l_1} < \underline{D}_q \tag{6}$$
$$(2)\&(3) \Rightarrow D_q \cap E_{l_1} \neq \emptyset \Leftrightarrow \underline{D}_q \leq \overline{E}_{l_1} \tag{7}$$
$$(6)\&(7) \Rightarrow \overline{E}_{l_1} < \underline{D}_q \wedge \underline{D}_q \leq \overline{E}_{l_1} \Rightarrow \lightning \tag{8}$$

Further we know that $r_1 \geq l_0$ which is proved by contradiction. Assume $r_1 < l_0$ and let $(H) \overset{def.}{=} r_1 < l_0$. Then we get:

$$(H)\&(2)\&(3) \Rightarrow \overline{D}_q < \underline{D}_p \tag{9}$$
$$(1) \Rightarrow \lightning \tag{10}$$
$$\Rightarrow r_1 \geq l_0 \tag{11}$$

Symmetrically the occurrence of a *crossing edge* on an upper bound can be found by considering cases where $u := \tau(i) \wedge v := \tau(j) \wedge u < v = u + 1$ and $U_u = [l_0..r_0] \wedge U_v =$

$[l_1..r_1] \wedge 0 < r_1 < r_0 \wedge l_0 < r_0 \neq rm_u$. Analogously to the proof for the lower bound we can show by contradiction that for the case of a *crossing edge* on an upper bound $l_0 \leq r_1$ holds. Summing up a *crossing edge* occurs on:

CE$_1$ - [a lower bound]

1. σ is a valid sorting permutation as specified in $[S_2]$
2. $lm_i := \min\{j \in \mathbb{K}_n | \underline{E}_j \leq \underline{D}_i \leq \overline{E}_j\}$, that is lm_i is the minimal index of the variable $y_{lm_i} : E_{lm_i} \in \mathcal{Y}$ belonging to the same SCC as x_i
3. $\exists i, j \in \mathbb{K}_n : i \neq j \wedge \sigma(i) < \sigma(j) \wedge \underline{U}_{\sigma(j)} < \underline{U}_{\sigma(i)} \leq \overline{U}_{\sigma(j)} \wedge lm_{\sigma(j)} \neq \underline{U}_{\sigma(j)} < \overline{U}_{\sigma(j)}$

CE$_2$ - [an upper bound]

1. τ is a valid sorting permutation as specified in $[S_2]$
2. $rm_i := \max\{j \in \mathbb{K}_n | \underline{E}_j \leq \overline{D}_i \leq \overline{E}_j\}$ that is rm_i is the maximal index of the variable $y_{rm_i} : E_{rm_i} \in \mathcal{Y}$ belonging to the same SCC as x_i
3. $\exists i, j \in \mathbb{K}_n : i \neq j \wedge \tau(i) < \tau(j) \wedge \underline{U}_{\tau(i)} \leq \overline{U}_{\tau(j)} < \overline{U}_{\tau(i)} \wedge \underline{U}_{\tau(i)} > \overline{U}_{\tau(i)} \neq rm_{\tau(i)}$

In order to ensure bounds consistency on the lower bounds of the permutation variables after the successful domain reduction of $\mathcal{X}$ and $\mathcal{Y}$ we apply the following algorithm [1](symmetrically for the upper bounds):

Algorithm 1 `perm_bc` on lower bound

```
Require: $\sigma$ with $\underline{D}_{\sigma(0)} \leq \ldots \leq \underline{D}_{\sigma(n-1)}$, $idx_i := \min\{j \in \mathbb{K}_n | x_i \bowtie y_j \ in\ G^-\}$
 1: for $i = 1; i < n; i \leftarrow i + 1$ do
 2:   if $\left(\sigma(i-1) < \sigma(i) \wedge \underline{U}_{\sigma(i-1)} > \underline{U}_{\sigma(i)} \wedge \underline{U}_{\sigma(i)} \neq idx_{\sigma(i)}\right)$ then
 3:     if $\left(\left|U_{\sigma(i)}\right| = 1\right)$ then // Vital matching edge do not remove it
 4:       if $\left(\overline{S}_{\sigma(i-1)} < \underline{S}_{\sigma(i)}\right)$ then
 5:         return $P_1$        // Report failure (sec.2.3.2)
 6:       end if
 7:     else
 8:       $\underline{U}_{\sigma(i)} \leftarrow \underline{U}_{\sigma(i-1)}$        // Update the lower bound
 9:     end if
10:   end if
11: end for
```

The update in line 8 is motivated as follows:

1. If $\left|U_{\sigma(i)}\right| = 1$ holds (line 3) the conflicting value $\underline{U}_{\sigma(i)}$ represents a vital matching edge and must not be removed. If also $\overline{S}_{\sigma(i-1)} < \underline{S}_{\sigma(i)}$ holds the violation cannot be repaired even using the upper bound sorting τ as a matching.
2. If $\left|U_{\sigma(i)}\right| > 1$ and CE_1 holds the conflicting value does neither belong to a matching on the intersection graph G nor to a SCC computed in step $[S_4]$. Hence the corresponding variable $x_{\sigma(i)} : D_{\sigma(i)} \in \mathcal{X}$ can only take values greater than those $x_{\sigma(i-1)} : D_{\sigma(i-1)}$ can take. Hence we achieve bounds consistency on the permutation variables by updating $\underline{U}_{\sigma(i)}$ as described in line 8.

As the algorithm processes all permutation variables it has the same complexity as the normalization step $[S_1]$ for $\mathcal{X}$ and $\mathcal{Y}$. Hence we have $\mathcal{T}(S_{5.3}) = \Theta(n)$. Thus extending the `sortedness` constraint to permutation variables results in the same total complexity as `sortedness` which is $\mathcal{T}(\texttt{sortedness}^+) = \mathcal{T}(\texttt{sortedness}) = \mathcal{O}(n \cdot log(n))$.

[1] `sortedness/order.icc`

3.2.3 Idempotency for sorting propagators

Sortedness

Provide a space $\mathcal{S} = \mathcal{X} \cup Y$ such that $\mathcal{S}$ is solvable(`sortedness` $(\mathcal{S}) \neq [P_1]$) and `sortedness` is not subsumed on $\mathcal{S}$ (`sortedness` $(\mathcal{S}) \neq [P_4]$) and that $\forall i \in \mathbb{K}_n$ D_i and E_i are ranges of the form $[a..b]$ we notice the following behavior: At first the propagation algorithm checks in the normalization step $[S_1]$ whether the variable domains in the initial space are feasible inputs for the constraint and eventually modifies the variable domains in order to restore feasibility. After the initial modification it uses the domain information from the variables in order to compute all perfect matchings on a special graph structure of $\mathcal{X}$ and $\mathcal{Y}$. Having found out what values in the domains form such a matching it finally reduces the variable domains such that there are only those values left needed by any matching computed by the above steps. Thus if the variable domains are not modified externally `sortedness` cannot infer more information than gathered in the above mentioned computation steps and hence achieves its fixpoint $[P_2]$. There are only two possibilities that `sortedness` is not at a fixpoint after one pass of the propagation algorithm:

1. Holes in the domains:
   There is at least one variable domain D such that $D \subset \{a, \dots, b\} \wedge \{a, b\} \subset D$. As the propagation algorithm for `sortedness` is bounds consistent he treats such a domain as range $D = [a..b]$. Hence a modification of the domain bounds could result in the loss of bounds consistency and we need at least another pass of the propagation algorithm to restore bounds consistency again.
2. Shared variables:
   Sharing is a special *Gecode* mechanism that we explain in more detail in section 4. If sharing is detected while propagating the computed domain information back to the variables we also loose bounds consistency and have to perform at least one additional pass of the propagation algorithm in order to restore consistency on the domain bounds.

Sortedness with Permutation Variables As `sortedness`$^+$ is an extension of the above discussed `sortedness` constraint the same two cases as above may occur where the propagator is not at a fixpoint. Nevertheless, the propagation algorithm for `sortedness`$^+$ does not support sharing on the permutation variables. Because of the extension to permutation variables however, there is a third possibility that the propagator is not at a fixpoint:

3. Crossing edge:
   From step $[S_{5.3}]$ we know that crossing edges come along with the loss of bounds consistency on the permutation variables. Although this inconsistency is fixed with the `perm_bc` algorithm presented in the same step the connection of $\mathcal{X}$ and $\mathcal{Y}$ through $\mathcal{Z}$ could have changed in the sense that there might by stronger bounds for the problem variables. Hence, the propagator can possibly infer new domain information about the variables in $\mathcal{X}$ and $\mathcal{Y}$, is not at a fixpoint and we have to perform another pass of the propagator. But why is one additional pass of the propagator sufficient to restore idempotency? Using Thiel's graph view[39] on the work of Bleuzen-Guernalec and Colmerauer the bounds of the permutation variables can be characterized as follows: either the bounds represent edges belonging to an SCC on G^- including matching edges or they represent edges joining smaller SCCs to a larger SCC. The latter ones directly correspond to the *crossing edges* mentioned above. Hence, removing those edges joining strongly connected components in G^- we obtain only those edges that take part in some perfect matching on G and respect the variable sorting. Thus, the second pass of the propagator is able to identify the smaller SCCs and narrow the $\mathcal{X}$ -variables accordingly. As by the removal of the crossing edges there is no need to execute `perm_bc` and the second pass of the propagator does not modify the bounds

of the permutation variables and proceeds according to the propagation algorithm for the `sortedness` constraint achieving its fixpoint again.

If the `perm_bc` algorithm detects no crossing edges on the reduced intersection graph then not only the variables in $\mathcal{X}$ and $\mathcal{Y}$ are bounds consistent but also the permutation variables in $\mathcal{Z}$ since all values in the domains of the permutation variables do take part in some solution to the constraint. Hence `sortedness`$^{+}$ cannot propagate other domain information back to the variables than it already deduced in the computation steps mentioned above.

3.3 Global Cardinality

In this section we focus on propagations algorithms for the `globalcardinality` constraint (`gcc`) that is also referred to as the `generalized cardinality` constraint. The implementation of the domain consistent propagator `gcc`$_{dom}$ in *Gecode* is based on *Improved Algorithms for the Global Cardinality Constraint* [25] by Quimper *et al.* and the implementation of the bounds consistent propagator `gcc`$_{bnd}$ in *Gecode* is taken from *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint* [24] by Quimper *et al.*. The denotational semantics of `gcc` are defined as follows:

Definition 28 (Global Cardinality). *Given a finite set of FDVars $\mathcal{X}$* , the union of all variable domains $\mathcal{D}(\mathcal{X})$, a finite set $V \subseteq \mathcal{D}(\mathcal{X})$ of values for the variables with $|V| = m$ and a finite set $\mathcal{F} \subset \{(l, u) \in \mathbb{K}_{n+1} \times \mathbb{K}_{n+1} \mid l \leq u\}$ with $\left|\mathcal{F}\right| = |V| = m$, we define `global cardinality` as

$$\texttt{gcc}\,(\mathcal{X}, V, \mathcal{F}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{cl}(\mathcal{X}) \mid \forall i \in \mathbb{K}_n : v_i \in V \wedge (l_i, u_i) \in \mathcal{F} \wedge l_i \leq \#(\alpha, v_i) \leq u_i\}$$

where $cl \in \{val, bnd, dom\}$ indicates the consistency level of the `global cardinality` which is $cl = val$ for the value consistent, $cl = bnd$ for the bounds consistent and $cl = dom$ for the domain consistent version of the `gcc` propagation algorithm.

Obviously, `gcc` is a generalization of `alldiff` since the above definition enables us to rewrite `alldiff` as

$$\texttt{alldiff}\,(\mathcal{X}) := gcc(\mathcal{X}, \mathcal{D}(\mathcal{X}), \mathcal{F})\ with \mathcal{F} = \begin{cases} \{(0,1), \ldots, (0,1)\} & \text{if } \left|\mathcal{D}(\mathcal{X})\right| > \left|\mathcal{X}\right| \\ \{(1,1), \ldots, (1,1)\} & \text{if } \left|\mathcal{D}(\mathcal{X})\right| = \left|\mathcal{X}\right| \end{cases}$$

If $\mathcal{X}$, V and $\mathcal{F}$ are specified as below the following table presents some examples for the `global cardinality` constraint:

$\mathcal{X}_1 = \{x_0 : \{1\}, x_1 : \{2\}, x_2 : \{3\}, x_3 : \{2\}, x_4 : \{1\}\}$	$V_1 = \{1, 2, 3\}$	$\mathcal{F}_1 = \{(2, 2), (1, 2), (1, 1)\}$
$\mathcal{X}_2 = \{x_0 : \{1\}, x_1 : \{1\}, x_2 : \{1\}, x_3 : \{3\}, x_4 : \{3\}\}$	$V_2 = \{1, 3\}$	$\mathcal{F}_2 = \{(1, 2), (2, 2)\}$
$\mathcal{X}_3 = \{x_0 : \{1\}, x_1 : \{2\}, x_2 : \{3\}, x_3 : \{4\}, x_4 : \{1\}\}$	$V_3 = \{1, 2, 3, 4\}$	$\mathcal{F}_3 = \{(3, 3), (1, 1), (1, 1), (1, 1)\}$

`gcc` $(\mathcal{X}_1, V_1, \mathcal{F}_1)$	✓	correct
`gcc` $(\mathcal{X}_2, V_2, \mathcal{F}_2)$	$\lightning$	value 1 occurs to often
`gcc` $(\mathcal{X}_3, V_3, \mathcal{F}_3)$	$\lightning$	value 1 occurs to few

Table 4. Examples for `global cardinality`

3.3.1 From `alldiff` to `global cardinality` - Theoretical Results

Before we go into the detail of the propagation algorithms of a bounds and a domain consistent propagator for the `global cardinality` constraint we first want to recapitulate what we just learned from the computation steps S_3 and S_4 of the bounds consistent `sortedness` propagator: Every assignment $\alpha \in$ `sat(sortedness )` corresponds to a perfect matching M on the bipartite convex intersection graph G (see def.23). In contrast to the `sortedness` constraint the `alldiff` constraint requires a different bipartite graph to argue about. Hence we introduce the following graph structure:

Definition 29 (Variable value graph [see 26, p. 4]). *The undirected bipartite graph* $G := \langle \mathcal{X} \uplus \mathcal{D}(\mathcal{X}), E \rangle$*, where* $E = \{(x_i : D_i, v_j) \mid v_j \in D_i\}$ *is called the* variable value graph *of* $\mathcal{X}$ or $VVG_{\mathcal{X}}$.

Thus we can exchange the intersection graph from the `sortedness` constraint by the variable value graph for the `alldiff` constraint with the result that every assignment $\beta \in$ `sat(alldiff )` corresponds to a maximum matching M on $VVG_{\mathcal{X}}$ where either M is a set cover on $\mathcal{X}$ in $VVG_{\mathcal{X}}$ if $\left|\mathcal{D}(\mathcal{X})\right| > \left|\mathcal{X}\right|$ or a perfect matching M on $VVG_{\mathcal{X}}$ if $\left|\mathcal{D}(\mathcal{X})\right| = \left|\mathcal{X}\right|$. In this case we can apply the following theorem from graph theory:

Theorem 1 (König-Hall Theorem [5]). *Let* $G := \langle V = V_1 \uplus V_2, E \rangle$ *be a bipartite graph. Let further* $\Gamma_G(A) \subseteq V_2$ *denote the set of vertices adjacent to set* A *for any subset* $A \subseteq V_1$. *Then* G *has a perfect matching if and only if for every subset* $A \subseteq V_1$ *the inequality*

$$|A| \leq |\Gamma_G(A)|$$

holds.

In the case of the `alldiff` constraint we instantiate this theorem by definition 29 to $A = \mathcal{X}$ and $\Gamma_G(A) = \Gamma_{VVG_{\mathcal{X}}}(\mathcal{X}) = \mathcal{D}(\mathcal{X})$ deducing the following equivalence:

$$\texttt{sat(alldiff)} \neq \emptyset \Leftrightarrow |X| \leq \left|\mathcal{D}(\mathcal{X})\right| \tag{12}$$

stating that the `alldiff` constraint on $\mathcal{X}$ is satisfiable if and only if the union of all variable domains $\mathcal{D}(\mathcal{X})$ contains at least as many values as there are variables in $\mathcal{X}$. Nevertheless, this equivalence is not sufficient for the `gcc` constraint as the definition of the `gcc` allows a value $v \in \mathcal{D}(\mathcal{X})$ to occur more than once in a variable assignment α permitting that the node representing v in $VVG_{\mathcal{X}}$ can have more than only one incident matching edge. Hence, we extend the above definition of a variable value graph such that it fits the requirements of the `global cardinality` constraint as:

Definition 30 (Variable value graph for `global cardinality`). *The undirected bipartite graph* $G := \langle \mathcal{X} \uplus (\mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1}), E \rangle$ $E = \{(x_i : D_i, (v_j, l_j, u_j)) \mid v_j \in D_i \wedge \forall \alpha \in \texttt{ass}_{dom}(\mathcal{X}) : l_j \leq \#(\alpha, v_j) \leq u_j\}$ *is called the* variable value graph *of* $\mathcal{X}$ or $VVG_{\mathcal{X}}$.

The main idea required for extending the equivalence from equation (12) to the `gcc` is the following (see [24]): Since the structure of the `gcc` imposes a lower bound l_i and an upper bound u_i on the value occurrences $\#(\alpha, v_i)$ in α where $l_i \leq \#(\alpha, v_i) \leq u_i$ is required for all values $v_i \in V \subseteq \mathcal{D}(\mathcal{X})$ the constraint is decomposed into an *upper bound constraint (*`ubc` *)* and a *lower bound constraint (*`lbc` *)* such that the following equivalence holds:

$$\texttt{gcc}\,(\mathcal{X}, V, \mathcal{F}) \Leftrightarrow (\texttt{lbc}\,(\mathcal{X}, V, \mathcal{F}) \cap \texttt{ubc}\,(\mathcal{X}, V, \mathcal{F}))$$

In this case `lbc` and `ubc` are defined analogously to the `gcc` as:

$$\begin{aligned} \texttt{lbc}\,(\mathcal{X}, V, \mathcal{F}) &\overset{def.}{=} \{\alpha \in \texttt{ass}_{cl}(\mathcal{X}) \mid \\ &\quad \forall i \in \mathbb{K}_m : v_i \in V \wedge (l_i, u_i) \in \mathcal{F} \wedge l_i \leq \#(\alpha, v_i)\} \end{aligned} \tag{13}$$

$$\begin{aligned} \texttt{ubc}\,(\mathcal{X}, V, \mathcal{F}) &\overset{def.}{=} \{\alpha \in \texttt{ass}_{cl}(\mathcal{X}) \mid \\ &\quad \forall i \in \mathbb{K}_m : v_i \in V \wedge (l_i, u_i) \in \mathcal{F} \wedge \#(\alpha, v_i) \leq u_i\} \end{aligned} \tag{14}$$

In addition to this decomposition of the `gcc` we define

Definition 31 (Capacities). *Given a finite set $S \subseteq \mathcal{D}(\mathcal{X})$ we define the* capacity *of S as*

$$C(S) \overset{def.}{=} \left|\{x_i : D_i \in \mathcal{X} \mid D_i \subseteq S\}\right|$$

the intersection capacity *of S as*

$$I(S) \overset{def.}{=} \left|\{x_i : D_i \in \mathcal{X} \mid D_i \cap S \neq \emptyset\}\right|$$

the minimal capacity *of S as*

$$\lfloor S \rfloor \overset{def.}{=} \sum_{v_i \in S} l_i$$

and the maximal capacity *of S as*

$$\lceil S \rceil \overset{def.}{=} \sum_{v_i \in S} u_i$$

where l_i is the lower and u_i the upper bound of $\#(\alpha, v_i)$ as mentioned in the above `gcc` *definition.*

Provided this decomposition we instantiate the value c_v of the capacity function c such that for the `ubc` constraint $\forall i \in \mathbb{K}_m : c(v_i) = u_i$ and for the `lbc` constraint $\forall i \in \mathbb{K}_m : c(v_i) = l_i$ holds. This finally leads to the following extension of equation (12):

$$\mathtt{sat}(\mathtt{ubc}) \neq \emptyset \Leftrightarrow |X| \leq \lceil \mathcal{D}(\mathcal{X}) \rceil \tag{15}$$

$$\mathtt{sat}(\mathtt{lbc}) \neq \emptyset \Leftrightarrow \lfloor \mathcal{D}(\mathcal{X}) \rfloor \leq |X| \tag{16}$$

which, taken as a conjunction, result in the following satisfiability criterion for the `global cardinality` constraint:

$$\mathtt{sat}(\mathtt{gcc}) \neq \emptyset \Leftrightarrow \lfloor \mathcal{D}(\mathcal{X}) \rfloor \leq |X| \leq \lceil \mathcal{D}(\mathcal{X}) \rceil \tag{17}$$

As a result of these equivalences we conclude that the `global cardinality` constraint is satisfiable if and only if there is a maximum matching M_u on $VVG_{\mathcal{X}}$ with $|M_u| = \left|\mathcal{X}\right|$ and a maximum matching M_l on $VVG_{\mathcal{X}}$ such that $|M_l| = \lfloor \mathcal{D}(\mathcal{X}) \rfloor$ [see 25, sec. 3.2]. Having this extension from `alldiff` to `gcc` at hand the next sections present two propagation algorithms for the `global cardinality` constraint which despite their different consistency levels are both based on the equivalence described in equation (17).

3.3.2 A Bounds Consistent View

During this section we focus on a propagation algorithm for the `global cardinality` that achieves bounds consistency on the problem variables in $\mathcal{X}$ and whose implementation is taken from *An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint* [24] by Quimper *et al.*. Due to the close relation of `gcc` to `alldiff` (see sec. 3.3.1) the essence of this implementation consists in reusing an existing implementation of the `alldiff` constraint as provided in [20] and extending this implementation to the needs of the `global cardinality` constraint. The basic theory behind the bounds consistent `alldiff` propagator presented in [20] is motivated by the following definition:

Definition 32 (Hall interval (cf. section 3.1 [24])). *If there is a set $H \subseteq \mathcal{D}(\mathcal{X})$ such that*

$$C(H) = |H|$$

holds, H is called a Hall interval *and we write $Hall(H)$. Thus $Hall(H)$ indicates that the number of variables $x_i : D_i \in \mathcal{X}$ whose domains are contained in H equals the cardinality of H.*

By this definition and equation (12) it follows that:

$$\texttt{sat}(\texttt{alldiff}) \neq \emptyset \Leftrightarrow |X| \leq \left|\mathcal{D}(\mathcal{X})\right| \tag{18}$$
$$\Leftrightarrow \forall H \subseteq \mathcal{D}(\mathcal{X}) : C(H) \leq |H| \tag{19}$$

Thus the bounds consistent propagation algorithm for the `alldiff` constraint processes the problem variables in $\mathcal{X}$ checking whether there are Hall intervals in $\mathcal{D}(\mathcal{X})$ and performing domain reduction according to the detected Hall intervals. Conversely, the definition of the `ubc` constraint permits matching the values in $\mathcal{D}(\mathcal{X})$ more than once and we have to extend the above definition of Hall intervals to the notion of Hall sets as follows:

Definition 33 (Hall Sets). *If there is a set $H \subseteq \mathcal{D}(\mathcal{X})$ such that*

$$C(H) = \lceil H \rceil$$

holds, H is called a Hall set *and we write $Hall_S(H)$. Thus we write $Hall_S(H)$ if the number of variables $x_i : D_i \in \mathcal{X}$ whose domains are contained H equals the maximum capacity of H.*

Analogously to equation (19) it follows from this definition of Hall sets that:

$$\texttt{sat}(\texttt{ubc}) \neq \emptyset \Leftrightarrow |X| \leq \lceil \mathcal{D}(\mathcal{X}) \rceil \tag{20}$$
$$\Leftrightarrow \forall H \subseteq \mathcal{D}(\mathcal{X}) : C(H) \leq \lceil H \rceil \tag{21}$$

Moreover the propagation algorithm presented in [24] uses the following notation in order to achieve a similar result for the `lbc` constraint:

Definition 34 (Failure, Unstable and Stable Sets (cf. section 3.2 [24])). *Given a set $T \subseteq \mathcal{D}(\mathcal{X})$ we call T*

1. *a* failure set $F(T)$ *if*
$$I(T) < \lfloor T \rfloor$$
2. *an* unstable set $U(T)$ *if*
$$I(T) = \lfloor T \rfloor$$
3. *a* stable set $S(T)$ *if*
$$C(T) > \lfloor T \rfloor \wedge \forall R \subseteq \mathcal{D} : (U(R) \vee F(R) \Rightarrow T \cap R = \emptyset)$$

Since from this definition it follows that:

$$\texttt{sat}(\texttt{lbc}) \neq \emptyset \Leftrightarrow \lfloor \mathcal{D}(\mathcal{X}) \rfloor \leq |X| \tag{22}$$
$$\Leftrightarrow \neg\exists H \subseteq \mathcal{D}(\mathcal{X}) : F(H) \tag{23}$$

Obviously, for any interval $H \subseteq \mathcal{D}(\mathcal{X})$ the computation of $|H| = \overline{H} - \underline{H} + 1$ takes only constant time. Though, this computation only reasons about the bounds $\underline{H}$ and $\overline{H}$ and it is impossible to compute $\lceil H \rceil$ or $\lfloor H \rfloor$ likewise. As computation step $[S_2]$, which is orthogonal to $[S_1]$, solves this deficiency the computation steps for the $\texttt{gcc}_{bnd}$ propagator can be summarized as follows:

[S₁] - Sorting $\mathcal{X}$

The first step[7] of the algorithm for $\texttt{gcc}_{bnd}$ creates sorting permutations ν and μ such that

$$\underline{D}_{\nu(0)} \leq \ldots \leq \underline{D}_{\nu(n-1)} \wedge \overline{D}_{\mu(0)} \leq \ldots \leq \overline{D}_{\mu(n-1)}$$

holds. As for $[S_1]$ of the `sortedness` propagator we also apply an optimized quicksort algorithm[8] in order to achieve this sorting. Hence the creation of μ has a worst case complexity of $\mathcal{T}(S_1) = \mathcal{O}(n \cdot log(n))$.

[7] `gcc/bnd.icc`
[8] `support/sort.hh`

[S$_2$] - Additional Structures
In addition to the sorting of the domain bounds the propagation algorithm creates supporting structures that are required to reason about the satisfiability of the `lbc` and the `ubc` . At first this step creates the union of all domain bounds $\mathcal{B}(\mathcal{X})$ such that

$$\mathcal{B}(\mathcal{X}) \overset{def.}{=} \left(\bigcup_{i \in \mathbb{K}_n} \underline{D}_i \right) \cup \left(\bigcup_{i \in \mathbb{K}_n} \left(\overline{D}_i + 1 \right) \right) \cup \{\min(\mathcal{D}(\mathcal{X})) - 2, \max(\mathcal{D}(\mathcal{X})) + 2\}$$

Further $[S_2]$ also creates a mapping $r \in \mathcal{B}(\mathcal{X}) \mapsto \{0, \ldots, q-1\}$ with $\forall i \in \{0, \ldots, q-1\} : r(v) = i \Leftrightarrow b_i = v$ such that $\mathcal{B}(\mathcal{X})$ can be accessed via the set $\mathcal{B}(\mathcal{X}) = \{b_0, \ldots, b_{q-1}\} \wedge \left|\mathcal{B}(\mathcal{X})\right| = q$ where q is the number of unique domain bounds. The first and the last element in this set are used as sentinels where $b_0 = \min(\mathcal{D}(\mathcal{X})) - 2$ and $b_{q-1} = \max(\mathcal{D}(\mathcal{X})) + 2$. Apart from $\mathcal{B}(\mathcal{X})$ and the mapping r step also introduces a structure that allows the computation of $\lceil H \rceil$ and $\lfloor H \rfloor$ in constant time. This is achieved by creating partial sum structures Φ for the `ubc` and Ψ for the `lbc` which are defined as:

$$\Phi_i = \begin{cases} i & \text{if } 0 \le i \le 2 \\ \sum_{j=0}^{i-3} u_j & \text{if } 3 \le i < m+3 \\ \Phi_{i-1} + 1 & \text{if } m+3 \le i < m+5 \end{cases} \qquad \Psi_i = \begin{cases} i & \text{if } 0 \le i \le 2 \\ \sum_{j=0}^{i-3} l_j & \text{if } 3 \le i < m+3 \\ \Psi_{i-1} + 1 & \text{if } m+3 \le i < m+5 \end{cases}$$

This creation of Φ and Ψ which is in $\Theta\left(\left|\mathcal{B}(\mathcal{X})\right|\right)$ results in a constant time computation of $\lceil H \rceil = \Phi_{\overline{H}+3-\min(\mathcal{D}(\mathcal{X}))} - \Phi_{\underline{H}+3-\min(\mathcal{D}(\mathcal{X}))-1}$ and $\lfloor H \rfloor = \Psi_{\overline{H}+3-\min(\mathcal{D}(\mathcal{X}))} - \Psi_{\underline{H}+3-\min(\mathcal{D}(\mathcal{X}))-1}$. Since the setup of $\mathcal{B}(\mathcal{X})$, r and the partial sums each take $\Theta(n)$ time the complexity of this step is $\mathcal{T}(S_2) = \Theta(n)$.

[S$_3$] - Satisfying the upper bounds constraint (`ubc`)
Eq. (21) implies that the `ubc` is only satisfiable if there is no subset $H \subseteq \mathcal{D}(\mathcal{X})$ whose capicity exceeds its maximal capacity. Hence, a subset H either is a Hall set or its capacity is strictly smaller than its maximal capacity. Thus, the main task of this computation step is detecting and marking Hall sets and updating the variable bounds according to detected Hall sets.

[S$_{3.1}$] - Updating the lower bounds
In order to update the lower domain bounds the $\mathcal{X}$ variables are processed in increasing order of their upper domain bounds according to the sorting permutation μ computed in step $[S_1]$. Moreover, the detection of Hall sets and the update of the domain bounds are based on the central principle of *domination* whose explanation requires the following definitions:

Definition 35 (Cardinality counter [see 20, 21, sec. 3.1]). *Given a value $s \in \mathcal{B}(\mathcal{X})$ and $i, j \in \mathbb{K}_n$ we define the* cardinality counter *for value s as $c_s^i \overset{def.}{=} \left|\{j \le i \mid \underline{D}_i \ge s\}\right|$ and the* capacity *of s as $v_s^i \overset{def.}{=} \left\lceil [s, \overline{D}_i] \right\rceil - c_s^i$ with respect to the corresponding variable $x_i : D_i \in \mathcal{X}$ in process.*

By μ sorting of the variables and the above definition index $m = \mu(i)$ marks the variable in process according to μ. With respect to this sorting it follows immediately that $c_s^m = C^m([s, \overline{D}_m])$ where $C^m(S)$ denotes the value of $C([s, \overline{D}_m])$ up to the iteration when variable x_m is processed. Hence v_s^m is the number of values $v \in [s, \overline{D}_m]$ that can still be assigned to unprocessed variables. Considering the above definitions the principle of *domination* can be defined as:

Definition 36 (Domination (Lemma 1, section 3.3, [21])).

$$\forall i \in \mathbb{K}_n, \forall j, z \in \mathbb{K}_q : j < z : \qquad v_{b_j}^m \ge v_{b_z}^m \wedge m = \mu(i) \tag{24}$$

$$S_1 = [b_j, \overline{D}_m] \supset S_2 = [b_z, \overline{D}_m] \tag{25}$$

$$j \text{ dominates } z \overset{Lemma\ 1}{\Leftrightarrow} v^m_{b_j} \leq v^m_{b_z} \overset{(24)-(26)}{\Leftrightarrow} v^m_{b_j} = v^m_{b_z} \tag{26}$$

With this definition in mind updating the lower domain bounds is performed as follows: In order to keep track of capacity and cardinality counters we define

$$p_m(z) = \begin{cases} 0 & \text{if } z = 0 \\ \max\{j \in \mathbb{K}_q \mid j < z \wedge v^m_{b_j} - v^m_{b_z} > 0\} & \text{if } z > 0 \end{cases}$$

as the undominated preceeding index of z while x_m is in process. Corresponding to the predecessor $p_i(z)$ of an index z we also define

$$s_m(z) = \begin{cases} q-1 & \text{if } z = q-1 \\ \min\{j \in \mathbb{K}_q \mid j > z \wedge v^m_{b_j} - v^m_{b_z} > 0\} & \text{if } z < q-1 \end{cases}$$

as the undominated succeeding index of z while processing x_m. In addition to this, the propagation algorithm maintains a balanced binary tree datastructure where every value b_s in $\mathcal{B}(\mathcal{X})$, but the left sentinel b_0, points to $b_{p_i(s)}$, its undominated predecessor. Further, we define the shorthand notation

$$d(z) = v^i_{b_{p_i(z)}} - v^i_{b_z}$$

as the *difference of capacities* between a value b_z and its predecessor in $\mathcal{B}(\mathcal{X})$, $b_{p_i(z)}$. Hence, if no variable has been processed so far $\forall z \in \{1, \ldots, q-1\} : b_{p_0(z)} = b_{z-1}$ holds that is all values b_z in $\mathcal{B}(\mathcal{X})$ but the left sentinel point initially to their direct predecessor in $\mathcal{B}(\mathcal{X})$ which is $b_{p_0(z)} = b_{z-1}$. Having initialized the pointer structure in this way the domain update for the lower bound processes the next variable $x_m : D_m \in \mathcal{X}$ with $m = \mu(i)$ in ascending order and computes indices z, y and j such that:

- $z = \min\{j \in \mathbb{K}_q \mid b_j \geq \underline{D}_m \wedge d(j) > 0\}$
- $y = r(\overline{D}_m)$
- $j = p_m(z)$ such that b_j is the undominated predecessor of b_z

Given these indices the algorithm checks whether one of the succeding cases holds:

1. *Domination*
   Let $S_1 = [b_j, \overline{D}_m] \supset S_2 = [b_z, \overline{D}_m]$. If $d(z) = 0$ holds after processing x_m, j dominates z and the propagator infers that a possible Hall set $H = [b_z, b_h] \subseteq S_2$ is not left-maximal and extends H to $H := [b_j, b_h] \subseteq S_1$. Thus we only have to check whether S_1 contains a Hall set and can safely ignore z and S_2. In order to keep the undominated capacity pointer structure consistent b_z points to its undominated successor $b_{s_m(z)}$.
2. *Negative Capacity*
   A negative capacity is detected if there is an interval $H \subseteq \mathcal{D}(\mathcal{X})$ such that $C^m(H) > \lceil H \rceil \Rightarrow C(H) > \lceil H \rceil$. If so, from equation (21) it follows that the ubc is not satisfiable and neither is the gcc . In case that $y \leq z$ failure is detected by testing whether $d(z) < \left\lceil [b_y, b_z - 1] \right\rceil$ for $y \leq z$. Hence we get:

$$\begin{aligned} & d(z) < \left\lceil [b_y, b_z - 1] \right\rceil \qquad (27) \\ \overset{def\ d(z)}{\Leftrightarrow}\ & v^m_{b_j} - v^m_{b_z} < \left\lceil [b_y, b_z - 1] \right\rceil \\ \overset{def\ v^i_k}{\Leftrightarrow}\ & \left\lceil [b_j, b_z - 1] \right\rceil - \left\lceil [b_y, b_z - 1] \right\rceil < C^m([b_j, b_z - 1]) \\ \overset{def\ C^m}{\Leftrightarrow}\ & \left\lceil [b_j, b_y - 1] \right\rceil < C^m([b_j, \overline{D}_m]) \\ \overset{def\ \mathcal{B}(\mathcal{X})}{\Leftrightarrow}\ & \left\lceil [b_j, \overline{D}_m] \right\rceil < C^m([b_j, \overline{D}_m]) \end{aligned}$$

In case that $z < y$ the algorithm checks whether $d(z) < \lceil [b_z - 1, b_y] \rceil$ holds since:

$$\begin{aligned} & d(z) < \lceil [b_z - 1, b_y] \rceil \qquad (28) \\ \overset{def\ d(z)}{\Leftrightarrow}\ & v^m_{b_j} - v^m_{b_z} < \lceil [b_z - 1, b_y] \rceil \\ \overset{def\ v^i_k}{\Leftrightarrow}\ & \lceil [b_j, b_z - 1] \rceil < C^m([b_j, b_z - 1]) + \lceil [b_z - 1, b_y] \rceil \\ \Leftrightarrow\ & \lceil [b_j, b_z - 1] \rceil < C^m([b_j, b_z - 1]) \end{aligned}$$

3. *Narrowing the lower bound*
   Let S_m denote the reduced variable domain of the processed variable $x_m : D_m$. Then the proper update of its lower domain bound is performed by applying the following inference rule:

$$\text{UBC-Lower Bound}\ \frac{H = [b_l, b_r] \wedge Hall_S(H) \qquad x_m : D_m \in \mathcal{X} \wedge b_l \leq \underline{D}_m \leq b_r}{\underline{S}_m \geq b_k \qquad k = s_m(r)}$$

   If a Hall set $H = [b_l, b_r]$ is detected such that the currently processed variable $x_m : D_m$intersects H such that $b_l \leq \underline{D}_m \leq b_r$ (cf section 4.1 in [24]) the new lower bound $\underline{S}_m$ for x_m is computed as $\underline{S}_i := b_k$ where $b_k \in \mathcal{B}(\mathcal{X})$ is the undominated successor of b_r.
4. *Zero-Test* (Lemma 2, section 3.3, [21])
   This step checks, whether $d(z) = \lceil [b_y, b_z - 1] \rceil \Leftrightarrow Hall_S([b_j, \overline{D}_m])$ and marks the detected Hall set $[b_j, \overline{D}_m]$. The equivalence results from exchanging < with = in (27) *Negative Capacity*. As $Hall_S([b_j, \overline{D}_m]) \Leftrightarrow v^m_{b_j} = 0$ this test is also refered to as *Zero-Test*.

[S$_{3.2}$] - Updating the upper bounds
Narrowing the upper domain bounds is done in a symmetric way by processing the variables in decreasing order of the lower domain bounds starting with the last variable. As the decreasing order of the lower domain bounds is given by the sorting permutation ν created in step $[S_1]$ the index of the variable in process x_m is given by $m = \nu(i)$. Further the index structure in the balanced binary tree is inverted such that a value b_z initially points towards b_j with $j = s_m(z)$ instead of pointing to its predecessor $b_{p_m(z)}$ as in step $[S_{3.1}]$. Moreover, we redefine the *cardinality counter* of a value s as $c^i_s \overset{def.}{=} |\{j \geq i \mid \overline{D}_i \leq s\}|$ and the capacity of s as $v^i_s = [\underline{D}_i, s] - c^i_s$, but $\forall i \in \mathbb{K}_n, \forall j, z \in \mathbb{K}_q : z < j : v^i_{b_j} \geq v^i_{b_z}$ still holds. The inversion of the pointer direction causes this step of the propagation algorithm to compute:

- $z = \min_j \{j \in \mathbb{K}_q \mid b_j \leq \overline{D}_m \wedge d(j) > 0\}$
- $y = r(\underline{D}_m)$
- $j = s_m(z)$ such that b_j is the undominated succesor of b_z

and yields the following changes in the four different cases checked by the propagator:

1. *Domination* still checks whether $d(z) = 0$, but semantically it checks whether $[\underline{D}_m, b_z]$ is *right-maximal* or whether the interval can be extended to $[\underline{D}_m, b_j]$. In order to keep the undominated capacity pointer structure consistent b_z points to its undominated predecessor b_s with $s = p_m(z)$.
2. *Negative Capacity* detects a failure if $d(z) < \lceil [b_z, b_y - 1] \rceil$ or $d(z) < \lceil [b_y, b_z - 1] \rceil$
3. *Narrowing the upper bound* applies

$$\text{UBC-Upper Bound}\ \frac{H = [b_l, b_r] \wedge Hall_S(H) \qquad x_m : D_m \in \mathcal{X} \wedge b_l \leq \overline{D}_m \leq b_r}{\overline{S}_m \leq b_k \qquad k = p_m(l)}$$

Thus this step updates $\overline{D}_m$ whenever there is a Hall set $H = [b_l, b_r]$, such that $b_l \leq \overline{D}_i \leq b_r$ holds (cf. section 4.1 in [24]). If so, the new upper bound $\overline{S}_m$ of x_m is computed as $\overline{S}_m := b_k$ where b_k is the undominated predecessor of b_l.

4. *Zero-Test* checks whether $d(z) = \left\lceil [b_z, b_y - 1] \right\rceil$ and marks $[\underline{D}_m, j]$ as a Hall set.

The overall complexity of $[S_3]$ is bounded by the operations for maintenance and update of the pointers in the underlying balanced binary tree data structure for the values in $\mathcal{B}(\mathcal{X})$ and the Hall sets which is achieved in $\mathcal{O}(n \cdot log(n))$ as explained in Lemma 3, section 3.3 of [21, 20].

[S$_4$] - Satisfying the lower bounds constraint (`lbc`)
From equation (23) we know that the `lbc` constraint is only satisfiable if there is no subset $H \subseteq \mathcal{D}(\mathcal{X})$ such that H is a failure set (cf. Lemma 2, section 3.2 in [24]). Hence this step analyzes the matching problem between $\mathcal{X}$ and $\mathcal{D}(\mathcal{X})$ in terms of failure, stable and unstable sets as explained in definition 34. In order to keep track of those different kind of sets the balanced binary tree data structure used in S_3 is extended by pointers for potentially stable and stable sets.

[S$_{4.1}$] - Updating the lower bounds
Updating the lower domain bounds of the problem variables requires processing them according to the sorting permutation μ created in step $[S_1]$ where $m = \mu(i)$ is the index of the variable in process. Using the same definitions as described in $[S_3]$ this step computes indices j, y, z such that:

– $j = p_m(z)$ such that b_j is the undominated predecessor of b_z
– $y = r(\overline{D}_m)$
– $z = \min\{j \in \mathbb{K}_q \mid b_j \geq \underline{D}_m \wedge d(j) > 0\}$

After the computation of these indices this step checks whether one of the following cases matches:

1. *Potentially stable set*
If the propagator detects that $\underline{D}_m < b_z \Leftrightarrow z > r(\underline{D}_m) + 1$ the set $S = [\underline{D}_m, \min(b_y, b_z)]$ contains more variables $x_k : D_k \ni \underline{D}_m$ than required by the corresponding lower bound $l_{\underline{D}_m - \min(\mathcal{D}(\mathcal{X}))}$. Hence it is possible that S becomes a stable set and S is marked as a potentially stable set by setting $ps(\min(y, z)) = \underline{D}_m$.
2. *Stable set*
First the propagator checks whether $U([b_j, \overline{D}_m])$ holds before x_m is processed. Analogously to the negative capacity step in $[S_{3.1}]$ we check for a stable this check is performed by testing whether $d(z) \leq \left\lfloor [b_y, b_z - 1] \right\rfloor$ since:

$$\begin{aligned} d(z) \leq \left\lfloor [b_y, b_z - 1] \right\rfloor &\Leftrightarrow C^m([b_j, \overline{D}_m]) \geq \left\lfloor [b_j, \overline{D}_m] \right\rfloor \\ &\Rightarrow I([b_j, \overline{D}_m]) = \left\lfloor [b_j, \overline{D}_m] \right\rfloor \Leftrightarrow U([b_j, \overline{D}_m]) \end{aligned}$$

If $d(z) \leq \left\lfloor [b_y, b_z - 1] \right\rfloor$ applies, we have $z > y$ and $z > j$. Thus, processing x_m results in increasing $C^m([b_j, \overline{D}_m])$ by one and we obtain $C^m([b_j, \overline{D}_m]) > \left\lfloor [b_j, \overline{D}_m] \right\rfloor$. Nevertheless, it is possible that $\exists R \subseteq \mathcal{D}(\mathcal{X}) : U(R) \wedge S \cap R \neq \emptyset$ contradicting to the definition of a stable set. However, $z > y$ implies that $z > r(\underline{D}_m) + 1$ stating that there is a potentially stable set that has become a stable set. In order to obtain the maximal stable subset of $[b_j, \overline{D}_m]$ the algorithm then marks $[ps(y), \overline{D}_m]$ as a stable set where $ps(y)$ is the lower bound of the maximal stable set $S \subseteq [b_j, \overline{D}_m]$ not intersecting an unstable or a failure set such that the former unstable set $[b_j, \overline{D}_m]$ is now partitioned into the unstable set $[b_j, ps(y) - 1]$ and the stable set $[ps(y), \overline{D}_m]$.

3. *Failure Set*
Let $S_1 = [b_j, \overline{D}_m] \supset S_2 = [b_z, \overline{D}_m]$. If $d(z) = 0$ holds after processing x_m, j dominates z and the propagator infers that a possible stable or unstable set $S = [b_z, b_h] \subseteq S_2$ is not left-maximal and extends H to $S := [b_j, b_h] \subseteq S_1$. Thus we only have to check whether S_1 is stable or unstable and can safely ignore z and S_2. In order to keep the pointer structure consistent b_z points to b_s with $s = s_m(z)$ such that $b_{p_m(s)} = b_j$.
4. *Unstable Set*
Corresponding to the *zero-test* case in the ubc step $[S_{3.1}]$ this step tests whether $d(z) = \left\lfloor [b_y, b_z - 1] \right\rfloor$ since
$$\begin{aligned} & d(z) = \left\lfloor [b_y, b_z - 1] \right\rfloor \\ \Leftrightarrow & \left\lfloor [b_j, \overline{D}_m] \right\rfloor = C^m([b_j, \overline{D}_m]) \\ \Rightarrow & \left\lfloor [b_j, \overline{D}_m] \right\rfloor = I([b_j, \overline{D}_m]) \Leftrightarrow U([b_j, \overline{D}_m]) \end{aligned}$$
If the above condition holds, we know that there are sufficiently many variables in $[b_j, \overline{D}_m]$ such that $\exists \alpha \in \mathtt{ass}_{bnd}(\mathcal{X}) : \forall v_i \in [b_j, \overline{D}_m] : \#(\alpha, v_i) \geq l_i$.
5. *Narrowing the lower bound*
If at iteration m we are already given a set $H \subseteq [b_l, b_r]$ that is no failure set and that intersects the domain D_m of x_m we can apply the following inference rule:
$$\text{LBC-LOWER BOUND}\ \frac{H = [b_l, b_r] \qquad \neg F(H) \qquad x_m : D_m \in \mathcal{X} \wedge b_l \leq \underline{D}_m < b_r}{\underline{S}_m \geq b_k \qquad k > r \wedge d(k) > 0}$$
since the variable $x_m : D_m$ is needed to cover the required minimum number of value occurrences in $D_m \setminus H$. Hence the lower bound $\underline{D}_m$ is set to the next value in $\mathcal{B}(\mathcal{X})$ not in H in order to cover its minimum occurrence. As at iteration m the partition of $\mathcal{D}(\mathcal{X})$ into stable and unstable sets is still unknown we do not know whether $S(H)$ holds. Hence $\underline{S}_m$ is stored as *newBound$_m$* in the data structure for later update. Thus, the proper reduction step of $\underline{D}_m$ to $\underline{S}_m$ is only performed if after processing all variables x_i the following holds:
$$\text{LBC-LOWER BOUND}\ \frac{\begin{array}{c} H = [b_l, b_r] \\ \neg F(H) \qquad x_m : D_m \in \mathcal{X} \wedge b_l \leq \underline{D}_m < b_r \qquad \forall R \subseteq \mathcal{D}(\mathcal{X}) : S(R) \wedge D_m \not\subset R \end{array}}{\underline{S}_m \geq b_k \qquad k = s_m(r)}$$

[S$_{4.2}$] - Updating the upper bounds
Analogously to the ubc we narrow the variables' upper bounds in a symmetric way by processing the variables in order of the lower bounds which is given by the sorting permutation ν such that the variable in process is x_m where $m = \nu(i)$. Further, the underlying pointer structure of the balanced binary tree is inverted such that a value b_z initially points to its undominated successor $b_{s_m(z)}$ in $\mathcal{B}(\mathcal{X})$ instead of pointing to its undominated predecessor. Further we redefine the *cardinality counter* and the *capacity counter* such that $c_s^m \overset{def.}{=} \left|\{j \geq m \mid \overline{D}_m \leq s\}\right|$ and $v_s^m = [\underline{D}_m, s] - c_s^m$ holds for value s after variable x_m has been processed. These structural changes result in the following modifications to the above steps:

1. *Potentially stable and stable sets*
The maximal stable subset $S \subseteq \mathcal{D}(\mathcal{X})$ has already been determined in step $[S_{4.1}]$ and hence we can omit the steps for *potentially stable* and *stable* sets for the upper domain bounds since the stable set information can be reused in this step without recomputing the set again.

2. *Failure Set*
   This step still checks whether $d(z) = 0$ holds, but now tests whether the current the unstable set $S := [\underline{D}_m, b_z]$ is right-maximal or whether it can be extended to $[\underline{D}_m, b_j]$.
3. *Unstable Set*
   This case tests whether $d(z) = \lfloor [b_z, b_y - 1] \rfloor$ and marks $([\underline{D}_m, b_j])$ as an unstable set if the former condition holds.
4. *Narrowing the upper bound*
   If at iteration m there is a set $H \subseteq [b_l, b_r]$ that is no failure set and that intersects the domain D_m of x_m we can apply the following inference rule:

$$\textsc{LBC-Upper Bound}\ \frac{H = [b_l, b_r] \qquad \neg F(H) \qquad x_m : D_m \in \mathcal{X} \wedge \underline{D}_m < b_l \leq \overline{D}_m \leq b_r}{\overline{S}_m \leq b_k \qquad k = p_m(l)}$$

Although we already now the maximal stable subset of $\mathcal{D}(\mathcal{X})$ the proper update of the upper bounds is performed after all variables have been processed where the propagator performs the following inference:

$$\textsc{LBC-Upper Bound}\ \frac{\begin{matrix} H = [b_l, b_r] \\ \neg F(H) \qquad x_m : D_m \in \mathcal{X} \wedge \underline{D}_m < b_l \leq \overline{D}_m \leq b_r \qquad \forall R \subseteq \mathcal{D}(\mathcal{X}) : S(R) \wedge D_m \not\subset R \end{matrix}}{\overline{S}_m \leq b_k \qquad k = p_m(l)}$$

As the computation step for the `lbc` part uses the same balanced binary tree data structure as the `ubc` step $[S_3]$ its complexity depends likewise on the complexity of the most expensive tree operations

[Find]
: Find undominated capacity pointers in the balanced binary tree

[Path compression]
: Given a path p in the tree and a node v make every node on p a direct successor of v.

having each a worst case complexity of $\mathcal{O}(n \cdot log(n))$. Hence, the worst case complexity of the `lbc` step is the same as for the `ubc` step that also depends on those operations on the underlying data structure. Thus, the time needed for checking satisfiability of the `lbc` is at least $\omega(n)$ and at most $\mathcal{O}(n \cdot log(n))$.

Total complexity of `gcc` A synopsis of the complexity for all computation steps for `gcc`$_{bnd}$ is shown in table 5 underlining

$\mathcal{T}(S_1)$	$\mathcal{O}(T)$
$\mathcal{T}(S_2)$	$\Theta(n)$
$\mathcal{T}(S_3)$	$\mathcal{O}(n \cdot log(n))$
$\mathcal{T}(S_4)$	$\mathcal{O}(n \cdot log(n))$
$\mathcal{T}($`gcc`$)$	$\mathcal{O}(n \cdot log(n))$

Table 5. complexity-bounds consistent `gcc`

that the complexity of the bounds consistent propagation algorithm for the `gcc` constraint is bounded by the complexity of the `lbc` and the `ubc` part which themselves are bounded by a $\mathcal{O}(n \cdot log(n))$ worst case complexity of the find and compression operations in the balanced binary tree structure as described in [24].

3.3.3 A Domain Consistent View

In the previous section we discussed a bounds consistent propagation algorithm for the `gcc` based on the theory of *Hall intervals* and an extension thereof. The essence of this bounds consistent algorithm is to project the result from the König-Hall Theorem (see theorem 1) on the domain bounds. Hence, the bounds consistent propagation algorithm does not need to construct the full $VVG_{\mathcal{X}}$ in order to determine the satisfiability of the `gcc` on the given problem variables in $\mathcal{X}$. Conversely, this section gives a glance at the underlying theory of a domain consistent propagation algorithm `gcc` $_{dom}$ explicitly constructing the full $VVG_{\mathcal{X}}$ and relying on graph algorithms as used in the propagation algorithm for the `sortedness` constraint in section 3.2. The implementation of `gcc` $_{dom}$ described in this section is based on *Improved Algorithms for the Global Cardinality Constraint* [25] by Claude-Guy Quimper et al. Analogously to `gcc` $_{bnd}$ (see section 3.3.2) the propagation algorithm for `gcc` $_{dom}$ determines the satisfiability of the constraint by decomposing it into two smaller constraints, namely the `ubc` and the `lbc` .

[S₁] - Construction of the variable value (VVG)
Since all further computation steps reason about the variable value graph for the `gcc` as explained in definition 30 the propagator's first step constructs $VVG_{\mathcal{X}}$ for the problem variables in $\mathcal{X}$. Let $\mathcal{X} = \{x_0 : \{2\},\ x_1 : \{1,2\},\ x_2 : \{2,3\},\ x_3 : \{2,3\},\ x_4 : [1..4], x_5 : \{3,4\}\}$, $V = \{1,2,3,4\}$ and $\mathcal{F} = \{(1,3),(1,3),(1,3),(2,3)\}$ then the picture below shows a variable value graph for `gcc` $(\mathcal{X}, V, \mathcal{F})$:

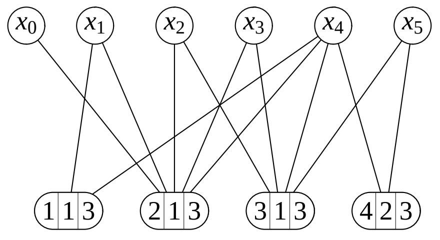

Fig. 3. Example of a variable value graph

In this figure a vertex in the upper $\mathcal{X}$ - partition of the graph represents a variable $x_i : D_i \in \mathcal{X}$ and a vertex in the lower $\mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1}$ - partition represents a triple (v_j, l_j, u_j) such that the first component of the vertex denotes a value $v_j \in \mathcal{D}(\mathcal{X})$ associated with the respective lower bound l_j and upper bound u_j on its occurrence in a variable assignment $\alpha \in \mathtt{ass}_{dom}(\mathtt{gcc})$. Given such a variable value graph we redefine the notion of a free vertex for a vertex $v_j \in \mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1}$ as: v_j *free with respect to the* `lbc` if $\left|\{e \in E \mid \mathtt{inc}(v_j, e)\}\right| < l_j$ and *free with respect to the* `ubc` if $\left|\{e \in E \mid \mathtt{inc}(v_j, e)\}\right| < u_j$. From the above example and by definition of the $VVG_{\mathcal{X}}$ it becomes obvious that it takes $\Theta(|D_i|)$ time to construct a variable vertex in the graph representing a variable $x_i : D_i$ with all outgoing edges to the value partition. Let further $d \overset{def.}{=} \max\left(\bigcup_{i \in \mathbb{K}_n} |D_i|\right)$ denote the cardinality of the largest variable domain. As the domain size $|D_i|$ is bounded by the size of the largest variable domain, d, the setup of the complete $VVG_{\mathcal{X}}$ is done in $\mathcal{T}(S_1) = \mathcal{O}\left(\left|\mathcal{X}\right| \cdot d\right)$ time. For the remainder of this section we refer to d as the size of the largest variable domain D_i, to $m_{\mathcal{X}} \overset{def.}{=} \left|d \cdot \mathcal{X}\right|$ as the maximal number of edges in the $VVG_{\mathcal{X}}$ and to $n_{\mathcal{X}} \overset{def.}{=} \left|\mathcal{X}\right| + \left|\mathcal{D}(\mathcal{X})\right|$ as the number of vertices in the $VVG_{\mathcal{X}}$.

[S$_2$] - Maximum matching
As depicted in sec. 3.3.1 the major insight into a propagation algorithm for the `global cardinality` constraint is the correspondence between a maximum matching on $VVG_{\mathcal{X}}$ and a variable assignment $\alpha \in$ `sat(gcc )`. From corollary 1 we know that an initial maximum matching M on $VVG_{\mathcal{X}}$ is sufficient to determine whether edges of the variable value graph belong to some maximum matching on $VVG_{\mathcal{X}}$ or not. Due to the decomposition of the `gcc` into the `ubc` and the `lbc` (see equation (13)) this step of the propagator checks whether there are two maximum matchings M_u and M_l on $VVG_{\mathcal{X}}$ such that $|M_u| = \left|\mathcal{X}\right| \wedge |M_l| = \lfloor \mathcal{D}(\mathcal{X}) \rfloor$. At first we describe the maximum matching step for the `ubc` and compare it subsequently to the maximum matching step for the `lbc` . Instead of the advanced Hopcroft-Karp bipartite maximum matching algorithm as proposed in [15, 26, 25] we use a hybrid algorithm H_M for testing whether there exists maximum matchings M_u and M_l on $VVG_{\mathcal{X}}$. The first step of H_M is a greedy algorithm G_M to compute a maximal matching N_u on $VVG_{\mathcal{X}}$ where $\gamma_u \overset{def.}{=} |N_u|$. If G_M results in a matching N_u with $\texttt{max}^{|\,|}(N_u, VVG_{\mathcal{X}})$ we set $M_u = N_u$ and we are done. If N_u is not maximum we use the following corollary:

Corollary 2 (Extending a maximal matching [see 5, chp. 7]) *Given a graph G and a matching M on G, M is maximum if, and only if, there exists no augmenting path between any two free vertices in G.*

In case that $\gamma_u \neq \left|\mathcal{X}\right|$ we know that $\texttt{max}^{\subset}(N_u, VVG_{\mathcal{X}})$ and the second step of H_M tests whether there exist k vertex-disjoint augmenting paths p_i such that N_u can be extended to a maximum matching $M_u \overset{def.}{=} N_u \oplus \left(\bigoplus_{i \in \mathbb{K}_k} p_i\right)$, where $k \in \mathcal{O}\left(\left|\mathcal{X}\right| - \gamma_u\right)$. If this augmentation step fails and N_u cannot be extended to a maximum matching M_u the propagator detects that the `gcc` constraint is unsatisfiable and fails. As G_M is a *factor-2-approximation-algorithm* for a maximum cardinality matching [9, chp. 4] and $\mathcal{T}(G_M) = \mathcal{O}(m_{\mathcal{X}})$ the resulting matching N_u has minimum cardinality $|N_u| \geq \frac{1}{2} \cdot |M_u|$. Hence, the augmentation phase of H_m performs at most $\left(\left|\mathcal{X}\right| - \gamma_u\right) \leq \frac{|\mathcal{X}|}{2}$ augmentation steps. Given a free vertex v_f in the variable value graph $VVG_{\mathcal{X}}$ the search for an augmenting path p_i is a depth-first search (`dfs`) on $VVG_{\mathcal{X}}$ starting from v_f and inverting all edges along p_i. As the complexity of `dfs` on $VVG_{\mathcal{X}}$ is $\mathcal{T}(\texttt{dfs}) = \mathcal{O}(m_{\mathcal{X}})$ and the edge inversion along p_i is in $\mathcal{O}(n_{\mathcal{X}})$ the overall complexity for one augmentation phase is at most $\mathcal{O}(m_{\mathcal{X}} + n_{\mathcal{X}})$. Hence, the computation of an initial maximum matching M_u is bounded by $\mathcal{O}\left(\frac{1}{2} \cdot \left|\mathcal{X}\right| \cdot (m_{\mathcal{X}} + n_{\mathcal{X}})\right) = \mathcal{O}\left(d \cdot \left|\mathcal{X}\right|^2\right)$, since $\mathcal{T}(G_m) = \mathcal{O}(m_{\mathcal{X}})$. At first sight this complexity is worse than the complexity of the above mentioned Hopcroft-Karp algorithm, $\mathcal{O}\left(m_{\mathcal{X}} \cdot \sqrt{\left|\mathcal{X}\right|}\right) = \mathcal{O}\left(\left|\mathcal{X}\right|^{\frac{3}{2}} \cdot \left|\mathcal{D}(\mathcal{X})\right|\right)$. However, Shapira showed in [36] that due to a theorem of Erdös and Gallai [5]

$$\mathcal{T}(H_M) = \mathcal{O}\left(m_{\mathcal{X}} \cdot max\left(\sqrt{n_{\mathcal{X}}^2 - n_{\mathcal{X}} - 2m_{\mathcal{X}}} - \frac{n_{\mathcal{X}}}{2}, \frac{n_{\mathcal{X}}}{2} - \sqrt{\frac{m_{\mathcal{X}}}{2}}\right)\right)$$

is an exact bound for the hybrid algorithm H_M. Taking a closer look at the number of edges it turns out that due to a CSP's problem structure the initial variable domains are rather large than small resulting in a $VVG_{\mathcal{X}}$ where $m_{\mathcal{X}} \in \Theta\left(\binom{n_{\mathcal{X}}}{2}\right)$. For this case the complexity $\mathcal{T}(M_u)$ for computing a maximum matching M_u is:

$$\mathcal{T}(M_u) \quad = \quad \mathcal{O}\left(m_{\mathcal{X}} \cdot max\left(\sqrt{n_{\mathcal{X}}^2 - n_{\mathcal{X}} - 2m_{\mathcal{X}}} - \frac{n_{\mathcal{X}}}{2}, \frac{n_{\mathcal{X}}}{2} - \sqrt{\frac{m_{\mathcal{X}}}{2}}\right)\right)$$

$$\begin{aligned}
&\overset{m_{\mathcal{X}}\in\Theta\left(\binom{n_{\mathcal{X}}}{2}\right)}{=} && \mathcal{O}\left(m_{\mathcal{X}} \cdot \frac{n_{\mathcal{X}}}{2} - \sqrt{\frac{m_{\mathcal{X}}}{2}}\right) \\
&= && \mathcal{O}\left(m_{\mathcal{X}} \cdot \left(n_{\mathcal{X}} - \sqrt{n_{\mathcal{X}}^2 - n_{\mathcal{X}}}\right)\right) \\
&\overset{L'H\hat{o}pital}{=} && o\left(m_{\mathcal{X}} \cdot \sqrt{n_{\mathcal{X}}}\right) = o\left(m_{\mathcal{X}} \cdot \sqrt{\left|\mathcal{X}\right| + \left|\mathcal{D}(\mathcal{X})\right|}\right)
\end{aligned}$$

In order to compute a maximum matching M_l for the `lbc` we only exchange N_u by N_l setting $\gamma_l \overset{def.}{=} \lfloor \mathcal{D}(\mathcal{X}) \rfloor$. Given this changes we perform the same hybrid algorithm as for the `ubc` part resulting in a complexity of $\mathcal{T}(M_l) = o\left(m_{\mathcal{X}} \cdot \sqrt{\left|\mathcal{X}\right|}\right)$, since $\lfloor \mathcal{D}(\mathcal{X}) \rfloor \leq \left|\mathcal{X}\right|$. Thus, the $\mathcal{T}(S_2)$ is upper bounded by $\mathcal{T}(M_u)$. Keeping in mind that $\left|\mathcal{D}(\mathcal{X})\right| \leq \left|\mathcal{X}\right|^k, k > 0$ we have to distinguish between two cases. If $\left|\mathcal{D}(\mathcal{X})\right| \in \mathcal{O}(\mathcal{X})$ holds it follows from $\mathcal{T}(M_u) = \left|\mathcal{D}(\mathcal{X})\right| \in o\left(m_{\mathcal{X}} \cdot \sqrt{\left|\mathcal{X}\right| + \left|\mathcal{D}(\mathcal{X})\right|}\right) = o\left(m_{\mathcal{X}} \cdot \sqrt{\mathcal{X}}\right)$ that the H_M has the same upper bound than a more elaborate maximum matching algorithm like the Hopcroft-Karp algorithm. For all other cases where $\mathcal{O}\left(\left|\mathcal{X}\right|^k\right)$ for $k > 1$ we know that H_m performs at most $\frac{|\mathcal{X}|}{2}$ augmentation steps. Hence, we obtain an overall complexity for the maximum matching step of $\mathcal{T}(S_2) = \mathcal{O}\left(d \cdot \left|\mathcal{X}\right| \cdot \min\left(\sqrt{\left|\mathcal{X}\right| + \left|\mathcal{D}(\mathcal{X})\right|}, \frac{1}{2}\left|\mathcal{X}\right|\right)\right)$.

[S₃] - Free alternating paths

As the previous step $[S_1]$ only computes initial maximum matchings M_u and M_l it is still possible that some vertices of the graph remain unmatched. By corollary 1 it suffices to check the existence of alternating paths starting in a free vertex and of strongly connected components in $VVG_{\mathcal{X}}$ to decide whether an edge of the variable value graph belongs to some maximum matching M or not. In context of the bipartite variable value graph Régin showed in [26, sec. 4] that an alternating path starting in a free vertex corresponds to a directed simple path starting in a free vertex in an oriented version of $VVG_{\mathcal{X}}$. Hence we introduce the following two graph definitions:

Definition 37 (Upper (Lower) oriented variable value graph). *Given a variable value graph $VVG_{\mathcal{X}} := \langle \mathcal{X} \uplus (\mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1}), E\rangle$ as defined in definition 30 and a maximum matching M_u on $VVG_{\mathcal{X}}$ with $|M_u| = \left|\mathcal{X}\right|$ we define the* upper oriented variable value graph *by directing every edge $e \in E \setminus M_u$ from the variable to the value partition and adding the reverse edge for every edge $e \in M_u$. Likewise, we define the* lower oriented variable value graph *for a maximum matching M_l with $|M_l| = \lfloor \mathcal{D}(\mathcal{X}) \rfloor$ by directing every edge $e \in E \setminus M_l$ from the value to the variable partition and adding the reverse edge for every edge $e \in M_l$. As shorthand notation we write $\overrightarrow{G_U}$ for the upper- and $\overrightarrow{G_L}$ for the lower oriented variable value graph.*

Obviously, after the computation of matching M_u in $[S_1]$ there can only be free vertices in the value partition of $\overrightarrow{G_U}$ since $|M_u| = \left|\mathcal{X}\right|$ states that M_u corresponds to a vertex cover on $\mathcal{X}$. As a value vertex in a variable value graph can match more than one variable vertex the number of free vertices in the value partition is bounded by $\left|\mathcal{D}(\mathcal{X})\right|$. Contrarily, after the computation of the M_l matching for the `lbc` constraint there can not only be free variables vertices but also free value vertices, where the number of free value vertices is bounded by $\mathcal{O}\left(\left|\mathcal{D}(\mathcal{X})\right|\right)$ and the number of free variable vertices is bounded by $\mathcal{O}\left(\left|\mathcal{X}\right|\right)$. Performing breadth-first search (`bfs`) on the respective partition possibly containing free vertices we compute all simple directed paths in the $\overrightarrow{G_U}$ in $\mathcal{O}\left(\left|\mathcal{D}(\mathcal{X})\right| + m_{\mathcal{X}}\right)$ and all simple directed paths in $\overrightarrow{G_L}$ in $\mathcal{O}\left(\left|\mathcal{X}\right| + m_{\mathcal{X}}\right)$. Hence, the complexity of step $[S_3]$ is $\mathcal{T}(S_3) = \mathcal{O}\left(\max\left(\left|\mathcal{X}\right|, \left|\mathcal{D}(\mathcal{X})\right|\right) + \left|\mathcal{X}\right| \cdot d\right)$.

[S₄] - Strongly Connected Components of $VVG_{\mathcal{X}}$
Given an initial maximum matching M from $[S_1]$ and having computed all existing free alternating paths this step of $\texttt{gcc}_{dom}$ has to check whether there exist even alternating cycles in $\overrightarrow{G_U}$ and $\overrightarrow{G_L}$. As described by Régin [26] the even alternating cycles in $\overrightarrow{G_U}$ and $\overrightarrow{G_L}$ directly correspond to the strongly connected components in those graphs. Hence, we use the `dfs`- based algorithm Mehlhorn presents in [22, chp. 4] to compute the strongly connected components of $\overrightarrow{G_U}$ and $\overrightarrow{G_L}$ in at most $\mathcal{T}(S_4) = \mathcal{O}(n_{\mathcal{X}} + m_{\mathcal{X}}) = \mathcal{O}\left(|\mathcal{X}| + |\mathcal{D}(\mathcal{X})| + |d \cdot \mathcal{X}|\right)$ time.

[S₅] - Narrowing $\mathcal{X}$
Concluding the above computation steps for $\texttt{gcc}_{dom}$ we now define the domain reduction step the propagator performs. Let $G := \langle \mathcal{X} \uplus (\mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1}), E\rangle$ denote the variable value graph for $\mathcal{X}$ as defined above and let $M_u \subseteq E$ be a maximum matching for the `ubc` and $\overrightarrow{G_U}$ the resulting upper oriented variable graph. Let further $S^*(\overrightarrow{G_U}) \overset{def.}{=} \bigcup_{\mathrm{SCC}(S,\overrightarrow{G_U})} S \subseteq E$ denote the union of all SCCs in $\overrightarrow{G_U}$ and let $A^*(\overrightarrow{G_U}) \overset{def.}{=} \bigcup_{\widetilde{p}} p \subseteq E$ denote the union of all even alternating paths in $\overrightarrow{G_U}$ starting in a free vertex. Then we compute the narrowed domain S_i of a variable $x_i : D_i$ by applying the following inference rule:

$$\textsc{Remove Edges-UBC}\ \frac{\begin{array}{c}\texttt{max}^{|\,|}(M_u \subset E, G) \wedge |M_u| = |\mathcal{X}| \\ E' = \{e = (x_i : D_i, w_j = (v_j, l_j, u_j)) \in E \mid e \in M_u \vee e \in A^*(\overrightarrow{G_U}) \vee e \in S^*(\overrightarrow{G_U})\}\end{array}}{S_i = \{v_j \in D_i \mid e = (x_i, (v_j, l_j, u_j)) \in E'\}}$$

Similarly, for the `lbc` case we exchange M_u with M_l and $\overrightarrow{G_U}$ with $\overrightarrow{G_L}$ resulting in the following reduction rule for the `lbc` :

$$\textsc{Remove Edges-LBC}\ \frac{\begin{array}{c}\texttt{max}^{|\,|}(M_l \subset E, G) \wedge |M_l| = \lfloor \mathcal{D}(\mathcal{X}) \rfloor \\ E' = \{e = (x_i : D_i, w_j = (v_j, l_j, u_j)) \in E \mid e \in M_l \vee e \in A^*(\overrightarrow{G_L}) \vee e \in S^*(\overrightarrow{G_L})\}\end{array}}{S_i = \{v_j \in D_i \mid e = (x_i, (v_j, l_j, u_j)) \in E'\}}$$

As the propagation algorithm has to check all values in all variable domains the narrowing step of $\texttt{gcc}_{dom}$ takes at most $\mathcal{T}(S_5) = \Theta(m_{\mathcal{X}}) = \Theta\left(|d \cdot \mathcal{X}|\right)$ time.

Total complexity of `gcc` A brief summary of the complexities for all computation steps in the domain consistent propagation algorithm $\texttt{gcc}_{dom}$ is given by table 6 highlighting

$\mathcal{T}(S_1)$	$\mathcal{O}\left(\lvert d \cdot \mathcal{X}\rvert\right)$
$\mathcal{T}(S_2)$	$\mathcal{O}\left(\lvert d \cdot \mathcal{X}\rvert \cdot \min\left(\sqrt{\lvert\mathcal{X}\rvert + \lvert\mathcal{D}(\mathcal{X})\rvert}, \frac{1}{2}\lvert\mathcal{X}\rvert\right)\right)$
$\mathcal{T}(S_3)$	$\mathcal{O}\left(\max\left(\lvert\mathcal{X}\rvert, \lvert\mathcal{D}(\mathcal{X})\rvert\right) + \lvert d \cdot \mathcal{X}\rvert\right)$
$\mathcal{T}(S_4)$	$\mathcal{O}\left(\lvert\mathcal{X}\rvert + \lvert\mathcal{D}(\mathcal{X})\rvert + \lvert d \cdot \mathcal{X}\rvert\right)$
$\mathcal{T}(S_5)$	$\Theta\left(\lvert d \cdot \mathcal{X}\rvert\right)$
$\mathcal{T}(\texttt{gcc})$	$\mathcal{T}(S_2)$

Table 6. complexity - $\texttt{gcc}_{dom}$

that the complexity of the `gcc` propagation algorithm is bounded by the complexity of the initial maximum matching computation in $[S_2]$.

Edge deletion Nevertheless, a variable $x_i : D_i$ in $\mathcal{X}$ may be constrained by more propagators than just the `gcc` we studied in this section and it is possible that during *constraint propagation* another propagator p, different from the `gcc` , reduces a variable domain D_i. Consequently, the $\mathcal{X}$-variables and the union of their respective domains, $\mathcal{D}(\mathcal{X})$, now differ from the previously constructed $VVG_{\mathcal{X}}$. Therefore Régin proposes in [26] to restore the consistency of the $VVG_{\mathcal{X}}$ by deleting the corresponding edge $e = (x_i : D_i, (v_j, l_j, u_j)) \in E$ from $VVG_{\mathcal{X}}$ instead of reconstructing the whole graph. Let M be a maximum matching on $VVG_{\mathcal{X}}$. If $e \notin M$ the algorithm restores the consistency of $VVG_{\mathcal{X}}$ by removing e from the graph. Otherwise, if $e \in M$, M is no longer maximum and has to be augmented again. Let R denote the set of edges that have to be removed from the graph such that $|R| = \delta$. Assume further that $m_{\mathcal{X}} - \delta \geq |M|$ since otherwise there exist no maximum matching on $VVG_{\mathcal{X}}$. In case that $\delta < |M|$ the algorithm has to perform at most δ augmentation steps resulting in a worst case complexity of $\mathcal{O}\left(d \cdot \left|\mathcal{X}\right|^2\right)$ if $M \subseteq R$. Again, taking a closer look at the number of edges and the correlation of $\left|\mathcal{X}\right|$ and $\left|\mathcal{D}(\mathcal{X})\right|$ we can do better than this upper bound: Let $m_{\mathcal{X}} = \left|d \cdot \mathcal{X}\right|$ and $|M| = \alpha$. The probability of deleting δ matching edges from the $VVG_{\mathcal{X}}$ is $Pr(|R \cap M| = \delta) = \left(\frac{\alpha! \cdot (m_{\mathcal{X}} - \delta)!}{(\alpha - \delta)! \cdot m_{\mathcal{X}}!}\right) \leq \left(\frac{\alpha}{m_{\mathcal{X}}}\right)^{\delta} \leq \left(\frac{\left|\mathcal{X}\right|}{\left|d \cdot \mathcal{X}\right|}\right)^{\delta} = \frac{1}{\left|\mathcal{D}(\mathcal{X})\right|^{\delta}} \leq \frac{1}{\left|\mathcal{X}\right|^{k \cdot \delta}}$ with $k > 0$. As the probability of deleting a matching edge decreases with increasing domain size and increasing size of $\mathcal{X}$ the worst case complexity for edge deletion including a possible augmentation of the defective matching shrinks to $\mathcal{O}(m_{\mathcal{X}})$. Therefore, respecting value removal by other propagators p in this way the propagator's complexity can only improve from $\mathcal{T}(S_2)$ to $\mathcal{T}(S_5)$ but does not exceed the upper bound of $\mathcal{T}(S_2)$.

3.3.4 Extended Global Cardinality

In the previous section we focused on propagation algorithms for the `gcc` with static lower and upper bounds l_j and u_j on the occurence of a value v_j in some variable assignment α. In the remainder of this section we discuss how to apply the above studied propagation algorithms in order to extend the `gcc` to an extended version of the `global cardinality` constraint we refer to as `global cardinality`$^+$ or `gcc`$^+$ that is defined as follows:

Definition 38 (Extended Global Cardinality). *Given a finite set of FDVars* $\mathcal{X} := \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$*, the union of all variable domains* $\mathcal{D}(\mathcal{X})$*, a finite set* $V \subseteq \mathcal{D}(\mathcal{X})$ *of values for the variables with* $|V| = m$*, a finite set of FDVars* $\mathcal{K} := \{k_0 : C_0, \ldots, k_{m-1} : C_{m-1}\}$ *such that* $\forall i \in \mathbb{K}_m : C_i = [\underline{C}_i..\overline{C}_i] \subseteq \{0, \ldots, \left|\mathcal{X}\right|\}$ *and* $\mathcal{S} \overset{def.}{=} \mathcal{X} \cup \mathcal{K}$*, we define* `global cardinality`$^+$ *as*

$$\texttt{gcc}^+(\mathcal{X}, V, \mathcal{K}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{cl}(\mathcal{S}) \mid \forall i \in \mathbb{K}_m : v_i \in V \wedge \#(\alpha, v_i) = \alpha(k_i)\}$$

where $cl \in \{val, bnd, dom\}$ *denotes the consistency level of the* `global cardinality` *which is* $cl = val$ *for the value consistent version of the* `gcc` $cl = bnd$ *for the bounds consistent version and* $cl = dom$ *for the domain consistent version.*

The above restriction of the variable domains C_i to be intervals is necessary in order to obtain domain consistency for $cl = dom$. Otherwise, the problem of enforcing domain consistency on $\mathcal{X}$ and $\mathcal{K}$ is $\mathcal{NP}$-complete [25, 27]. Given $\mathcal{X}$, $\mathcal{K}$ as specified below the table 4 presents some examples for the `global cardinality`$^+$ constraint: Provided the restriction on the cardinality variables as explained in definition 38 we extended the above propagation algorithms by the following computation steps:

$\mathcal{X}_1 = \{x_0 : \{1\}, x_1 : \{2\}, x_2 : \{3\}, x_3 : \{2\}, x_4 : \{1\}\}\ V_1 = \{1, 2, 3\}$
$\mathcal{K}_1 = \{k_0 : \{2\}, k_1 : [1..2], k_2 : \{1\}\}$
$\mathcal{X}_2 = \{x_0 : \{1\}, x_1 : \{1\}, x_2 : \{1\}, x_3 : \{3\}, x_4 : \{3\}\}\ V_2 = \{1, 3\}$
$\mathcal{K}_2 = \{k_0 : [1..2], k_1 : \{2\}\}$
$\mathcal{X}_3 = \{x_0 : \{1\}, x_1 : \{2\}, x_2 : \{3\}, x_3 : \{4\}, x_4 : \{1\}\}\ V_3 = \{1, 2, 3, 4\}$
$\mathcal{K}_3 = \{k_0 : \{3\}, k_1 : \{1\}, k_2 : \{1\}, k_3 : \{1\}\}$

`gcc`$^+$ $(\mathcal{X}_1, V_1, \mathcal{K}_1)$	✓	correct
`gcc`$^+$ $(\mathcal{X}_2, V_2, \mathcal{K}_2)$	↯	value 1 occurs to often
`gcc`$^+$ $(\mathcal{X}_3, V_3, \mathcal{K}_3)$	↯	value 1 occurs to few

Table 7. Examples for `global cardinality`$^+$

[S$_1$] - Consistency
At first we ensure that all C_i contain only feasible values such that

$$\forall i \in \mathbb{K}_k : 0 \leq \underline{C}_i \wedge \overline{C}_i \leq \left|\mathcal{X}\right|$$

holds as required in the definition of the `gcc`$^+$. Checking, whether the cardinality variables are consistent is in $\Theta\left(\left|\mathcal{K}\right|\right)$. Because of the restriction that a domain C_i of a cardinality variable k_i is an interval $C_i = [l_i..u_i]$ we can apply the propagation algorithms as discussed above where the domain bounds $\underline{C}_i$ and $\overline{C}_i$ are used as bounds on the value occurrence for $v_i \in V$ such that $\underline{C}_i \leq \#(\alpha, vi) \leq \overline{C}_i$.

[S$_2$] - Additional Constraints
Analogously to the filtering algorithm Régin applies for the cardinality variables in [27] the definition of `gcc`$^+$ constrains the domain bounds of the C_i's in a stronger way than the consistency check in $[S_1]$ by applying an additional bounds consistent propagator `linear` such that:

$$\texttt{linear}\,(\mathcal{K}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{bnd}(\mathcal{K}) \mid \sum_{i=0}^{m-1} \alpha(k_i) \sim \left|\mathcal{X}\right|\}$$

where we set $\sim \overset{def.}{=} =$ if $V = \mathcal{D}(\mathcal{X})$ and $\sim \overset{def.}{=} \leq$ if $V \subset \mathcal{D}(\mathcal{X})$.

[S$_3$] - Narrowing $\mathcal{K}$
After having applied the above `gcc` propagators with $l_j = \underline{C}_j$ and $u_j = \overline{C}_j$, $0 \leq j \leq |V| = \left|\mathcal{K}\right|$ we can perform the following reduction steps: Taking into account that the C_i's are restricted to be ranges we cannot do any better for the reduced lower bound $\underline{U}_j$ than applying the following inference rule before and after the application of the `gcc` propagator:

$$\textsc{Card-LB}\ \frac{c = \left|\{x_i : D_i \in \mathcal{X} \mid |D_i| = 1 \wedge D_i = \{v_j\}\}\right|}{\underline{U}_i \geq c}$$

From this rule immediately follows a trivial inference rule for the reduced upper bound $\overline{U}_j$, namely:

$$\textsc{Card-UB}\ \frac{a = \left|\{x_i : D_i \in \mathcal{X} \mid |D_i| = 1\}\right| \qquad c = \left|\{x_i : D_i \in \mathcal{X} \mid |D_i| = 1 \wedge D_i = \{v_j\}\}\right|}{\overline{U}_j \leq \left|\mathcal{X}\right| - (c - a)}$$

As the lookup operation for finding the corresponding cardinality variable k_j for a value v_j is bounded by $\mathcal{O}\left(log\left(\left|\mathcal{D}(\mathcal{X})\right|\right)\right) = \mathcal{O}\left(log\left(\left|\mathcal{D}(\mathcal{X})\right|\right)\right)$ the worst case complexity for the maintenance of the above rules is given by $\mathcal{O}\left(\left|\mathcal{X}\right| \cdot log\left(\left|\mathcal{D}(\mathcal{X})\right|\right) + \left|\mathcal{D}(\mathcal{X})\right|\right)$, where a is defined as defined in the above inference rules.

For the bounds consistent case of gcc$^+$ we add the following inference rule after the application of gcc $_{bnd}$. If S_i denotes the reduced variable domain of a problem variable $x_i : D_i \in \mathcal{X}$ it follows that:

$$\text{Card-UB-Bnd}\ \frac{l = \left|\{x_i : S_i \in \mathcal{X} \mid \overline{S}_i < v_j\}\right| \qquad u = \left|\{x_i : S_i \in \mathcal{X} \mid \underline{S}_i > v_j\}\right|}{\overline{U}_j \leq \left|\mathcal{X}\right| - l - u}$$

As the lookup operation for finding the corresponding cardinality variable k_j for a value v_j is bounded by $\mathcal{O}\left(log\left(\left|\mathcal{D}(\mathcal{X})\right|\right)\right) = \mathcal{O}\left(log\left(\left|\mathcal{D}(\mathcal{X})\right|\right)\right)$ the above narrowing steps is bounded by $\mathcal{O}\left(\left|\mathcal{X}\right| \cdot log\left(\left|\mathcal{D}(\mathcal{X})\right|\right) + \left|\mathcal{D}(\mathcal{X})\right|\right)$. In case of the domain consistent gcc we know that after one pass of the propagator the $VVG_{\mathcal{X}}$ contains only those edges taking part in a solution to the constraint Let $VVG^-_{\mathcal{X}} := \langle \mathcal{X} \uplus (\mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1}), E\rangle$ denote the variable value graph for the gcc after the removal of all infeasible edges. Hence, the domain C_j of the corresponding cardinality variable k_j for a value v_j in the variable domains of $\mathcal{X}$ can be reduced to the domain U_j such that:

$$\text{Card-UB-Dom}\ \frac{w_j = (v_i, \underline{C}_i, \overline{C}_i) \in \mathcal{D}(\mathcal{X}) \times \mathbb{K}_{n+1} \times \mathbb{K}_{n+1} \qquad u = \left|\{e \in E \mid \texttt{inc}(w_j, e)\}\right|}{\overline{U}_i \leq u}$$

Obviously, this reduction step is in $\Theta\left(\left|\mathcal{D}(\mathcal{X})\right|\right)$ and can be incorporated by the narrowing step $[S_5]$ of the gcc $_{dom}$ propagator.

3.3.5 Idempotency for cardinality propagators

Provided a space $\mathcal{S} = \mathcal{X}$ such that $\mathcal{S}$ is solvable, gcc $(\mathcal{S}) \neq [P_1]$, and gcc is not subsumed on $\mathcal{S}$, gcc $(\mathcal{S}) \neq [P_4]$,the gcc propagation algorithm follows the same global propagation scheme as the propagator for the sortedness constraint: At first the propagation algorithms collects the domain information as provided by the variable domains D_i. Given this domain information the algorithms proceed by applying their respective computation steps as specified in 3.3.2 for gcc $_{bnd}$ and 3.3.3 for gcc $_{dom}$. Unless the satisfiability tests for lbc or ubc report a failed space the gcc $_{bnd}$ and gcc $_{dom}$ propagate the new domain information from their computation steps back to the variables and achieve a fixpoint $[P_2]$ as they cannot infer more information than provided by one pass of the propagation algorithms. Clearly, we require for the bounds consistent case that the variable domains D_i contained in $\mathcal{S}$ are ranges of the form $D_i = [a..b]$. Analogously to the idempotency conditions of a sortedness propagator in section 3.2.3 both propagators, gcc $_{bnd}$ and gcc $_{dom}$, can be hindered from achieving a fixpoint by the presence of shared variables. Additionally, gcc $_{bnd}$ is not at a fixpoint, if it detects domains with holes the propagator was not aware of which is also the case for the bounds consistent sortedness propagator. As gcc$^+$ is an extension of gcc the same two cases as for global cardinality may occur where the propagator is not at a fixpoint.

4 The Gecode framework

In section 2 we looked at basic definitions of CP and introduced propagators as important concept to increase the efficiency of a constraint solver pruning variable domains with different strength and returning status information about inconsistencies or their idempotency. The subsequent section 3 discussed advanced propagation algorithms for global constraints and their extensions. In this section we give a short outline of important cornerstones in the *Gecode* framework that directly relate to the aspect of constraint propagation and the implementation of the above propagators in *Gecode* . The following definitions and introduced notions are mainly taken from the *Gecode* reference documentation [32], from *Speeding Up Constraint Propagation*[34] by Christian Schulte and Peter Stuckey.

Computation Space

The essential component of the *Gecode* architecture is a *computation space* $\mathcal{CS}$ as defined in [31] forming the computational equivalent of a CSP $P := \langle \mathcal{X}; \mathfrak{C} \rangle$ by encapsulating propagators $p \in \mathcal{P}$ implementing constraints $C \in \mathfrak{C}$ and a constraint store s that implements the set of problem variables $\mathcal{X}$ the constraints in $\mathfrak{C}$ are defined on and that the propagators $p \in \mathcal{P}$ are connected to. The connection of propagators to variables in the constraint store is realized by the mechanism of *variable subscription* [35]. On the one hand variable subscription associates propagators to those variables in the store s being in the scope of the constraint C implementing p. Moreover, variable subscription stores on the other hand the condition under that a propagator p is scheduled for execution. Hence, a form of communication between two independent propagators p_1, p_2 becomes only possible if they both depend at least on one common variable in the constraint store s. In addition to the encapsulation of propagators and variables, $\mathcal{CS}$ also manages the constraint propagation of propagators $p \in \mathcal{P}$ by implementing the set $\mathcal{P}$ of propagators as a queue $\mathcal{Q}$ scheduling the propagators that have to be applied next [32, 34].

Scheduling Propagators

An essential criterion which propagator to perform next in $\mathcal{Q}$ is the idempotency of a propagator as mentioned in 2.3.2. Obviously, an idempotent propagator p cannot infer new domain information for the variables in store s it is connected to directly after being run. Due to the design that a propagator p always returns is current propagation status with respect to its input store s as discussed in 2.3.2 the computation space is dynamically aware of the propagator's idempotency. Thus, constraint propagation inside $\mathcal{CS}$ is optimized by dynamically removing idempotent propagators p from $\mathcal{Q}$ [34]. Along with the idempotency criterion *Gecode* takes also care of the changes imposed to the variable domains. This is realized by introducing *modification events* . The following definition summarizes the modification events used in *Gecode* for constraint propagation on FDVars:

Definition 39 (Modification Event). *Given a FDVar $x_i : D_i$ and $S_i \subseteq D_i$ a reduced variable domain a* modification event *describes the result of an inference applied to D_i where we distinguish:*

[M_1] ***Failure****:*
The domain update resulted in a failure such that $S_i = \bot$

[M_2] ***Unmodified****:*
The variable domain has not been modified and $S_i = D_i$

[M_3] ***Assigned****:*
D_i has been reduced to the set of a single value $S_i = \{v\}$

[M_4] ***Bounds Modification****:*
Applies if $D_i = [a..b]$ has been narrowed to $S_i = [c..d]$ such that either $a < c \leq d = b$ denotes the change of the lower domain bound or $a = c \leq d < b$ denotes the change of the upper domain bound

[M_5] ***Domain Modification****:*
$\exists R \subset D_i : |R| \geq 1 \wedge S_i = D_i \setminus R$. Note that this modification event overlaps with [M_3] and [M_4].

With the help of these modification events indicating what change in the variable domain recently occurred during constraint propagation the computation space is able to define *propagation conditions* for propagators as follows:

Definition 40 (Propagation Condition). *In case that a variable domain D_i of a FDVar $x_i : D_i$ has been changed we define a* propagation condition *as an indicator what propagators p depending on $x_i : D_i$ have to be scheduled in $\mathcal{Q}$ for execution. For propagation on FDVars* Gecode *distinguishes the following propagation conditions:*

[PC_1] ***Value change***
Execute propagators $p \in \mathcal{P}$ connected to x_i under the condition that [M_3] occurred on x_i.

[PC_2] ***Bounds change***
Execute propagators $p \in \mathcal{P}$ connected to x_i under the condition that [M_4] occurred on x_i.

[PC_3] ***Domain change***
Execute propagators $p \in \mathcal{P}$ connected to x_i under the condition that [M_5] occurred on x_i.

Let us assume that a propagator p has already reached its fixpoint before the domain of a variable $x_i : D_i$ has been changed that p is connected with in $\mathcal{CS}$. Obviously, the propagator p can only infer new domain information for D_i if the propagation condition after a modification event on x_i coincides with p's consistency level. Otherwise repeated application of p to the store s containing x_i would not lead to new domain information. Hence, the above modification events are used to define propagation conditions under which the computation space detects whether the fixpoint of a propagator p has been affected by a modification event or not and consequently whether p has to be rescheduled in $\mathcal{Q}$ or not resulting in a further improvement [34] of the constraint propagation inside the computation space. In addition to the prioritization of $\mathcal{Q}$ according to a propagator's complexity [34] *Gecode* further improves the constraint propagation inside a computation space by introducing the mechanism of *staged propagation*. Consider the two propagation algorithms `gcc`$_{bnd}$ and `gcc`$_{dom}$ we discussed in sections 3.3.2 and 3.3.3, where `gcc`$_{dom}$ has an asymptotically greater theoretical complexity than `gcc`$_{bnd}$. From the definitions of modification events and propagation conditions we know that once at its fixpoint the `gcc`$_{bnd}$ is only enqueued for execution if [M_4] is detected on the variables it depends on whereas `gcc`$_{dom}$ is rescheduled in $\mathcal{Q}$ if [M_5] occurs on the variables it has subscribed to. In this context, the idea of staged propagation is to combine different propagators as `gcc`$_{bnd}$ and `gcc`$_{dom}$ implementing the same constraint to one single propagator `gcc` that decides depending on the current modification event which propagation algorithm to execute. Thus, staged propagation not only decreases the number of enqueued propagators but also combines multiple propagators to an efficient and effective propagator [34].

5 Experimental Results

In the last section of the paper we present an empirical evaluation of the implementations of the propagation algorithms for the `sortedness` and the `global cardinality` constraint we discussed in section 3. The main goals of this empirical analysis are as follows:

Efficiency We want to demonstrate that advanced propagation algorithms for global constraints as presented in section 3 are more efficient with respect to constraint propagation than their respective decomposition into local constraints.

Competitiveness on the same platform Second we want to show that the implementations for `sortedness` and `global cardinality` are competitive with an `alldiff` implementation in the above mentioned cases.

Competitiveness with other platforms Last, we want to highlight that these global constraints are competitive with their respective counterparts in other constraint solvers like *SICStus* and *ILOG*. For cross-platform comparison we tried hard to literally implement the same model on different platforms by using the same or constraints or equivalent constraints with respect to denotational semantics and consistency level and assured the same number of backtracks [41] for the respective problem on both platforms in focus.

The experiments with the following models use the *Gecode* C++ constraint programming library [38] and are either part of the *Gecode* library or can be obtained by request from the author of this paper. Due to license restrictions and server capacities the benchmarks were carried out on two different platforms:

[B_1] is a Fujitsu-Siemens with a `Pentium IV` 2,8 Ghz hyper-threading processor with 1024 MB of working memory running RedHat Linux using GNU GCC Compiler 3.4.3. This platform was used for the competitiveness benchmarks against *ILOG* using *ILOG* Solver 6.0.

[B_2] is a S260 MSI Megabook Laptop with a `Pentium M` 2,0 Ghz processor with 1024 MB of working memory running Cygwin on Windows XP using the Microsoft Visual C++ Toolkit 2003. This platform was used for the competitiveness benchmarks against *SICStus* using *SICStus* 3.12.3.

The runtimes for the benchmarks have been measured as the arithmetic mean of 20 runs for each problem, with a coefficient of deviation below 2% .

5.1 Sortedness

In order to evaluate whether the discussed propagator for the `sortedness` constraint is able to improve constraint propagation we compare it to a propagation algorithm for a naive `sortedness` constraint we define as follows:

Definition 41 (D-Sort). *Given two finite sets of FDVars $\mathcal{X} = \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$, and $\mathcal{Y} = \{y_0 : E_0, \ldots, y_{n-1} : E_{n-1}\}$ and a space $\mathcal{S} = \mathcal{X} \cup \mathcal{Y}$ we define:*

$$\texttt{d-sort}(\mathcal{X},\mathcal{Y}) \overset{def.}{=} \bigcap_{0\le i<j\le|\mathcal{Y}|} \texttt{leq}_{y_i \le y_j} \cap \bigcap_{0\le i\le|\mathcal{X}|} \texttt{element}(x_i,\mathcal{Y}) \cap \bigcap_{0\le i\le|\mathcal{X}|} \texttt{element}(y_i,\mathcal{X})$$

with

$$\texttt{leq}_{x\le y} \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{bnd}}(\mathcal{S}) | \alpha(x) \le \alpha(y)\}$$

$$\texttt{element}(x,\mathcal{Y}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{bnd}}(\mathcal{S}) | \exists i \in \mathbb{K}_{|\mathcal{Y}|} : \alpha(x) = \alpha(y_i)\}$$

$$\texttt{element}(y,\mathcal{X}) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{bnd}}(\mathcal{S}) | \exists i \in \mathbb{K}_{|\mathcal{X}|} : \alpha(y) = \alpha(x_i)\}$$

Apart from evaluating whether `sortedness` outperforms its decomposition or not we are also able to test it against other global propagators as motivated by the following equivalences. Consider an `alldiff` constraint as presented in section 2.3.4. Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets of FDVars with $\mathcal{Y} = \{y_0 : E_0, \ldots, y_{n-1} : E_{n-1}\}$ such that $|\mathcal{D}(\mathcal{X})| = |\mathcal{X}| = |\mathcal{Y}|$, $\overline{E}_0 \le \ldots \le \overline{E}_{n-1}$ and $\biguplus_{i\in\mathbb{K}_{|\mathcal{Y}|}} E_i = \mathcal{D}(\mathcal{X})$ hold. Then we obtain the equivalence:

$$\texttt{sortedness}(\mathcal{X},\mathcal{Y}) \equiv \texttt{alldiff}(\mathcal{X}) \tag{29}$$

If so we are able to compare the bounds consistent propagator for `sortedness` with a bounds consistent propagation algorithm for the `alldiff` constraint and hence can test with every model using the `alldiff` constraint under the above requirements, whether the `sortedness` propagator is competitive with an established efficient and advanced propagation algorithm for a global constraint like the `alldiff`. Moreover, we observe that, given finite sets of FDVars $\mathcal{X}$ and $\mathcal{Y}$ with $\mathcal{Y} = \{y_0 : E_0, \ldots, y_{n-1} : E_{n-1}\}$, the union of all variable domains in $\mathcal{X}$, $\mathcal{D}(\mathcal{X})$ with $|\mathcal{D}(\mathcal{X})| = m$, and a finite set $\mathcal{F} \subset \{(l,u) \in \mathbb{K}_n \times \mathbb{K}_n \mid l \le u\}$ such that $|\mathcal{Y}| = \sum_{0\in\mathbb{K}_m}$, $\forall(l_j,u_j) \in \mathcal{F} : l_j = u_j \overset{def.}{=} k_j$ and

$\forall i \in \mathbb{K}_m : \left|\{E_j \mid 0 \leq j \leq \left|\mathcal{Y}\right| \wedge E_j = \{v_i\}\}\right| = k_i$, we can deduce the following correspondence:

$$\texttt{sortedness}\,(\mathcal{X}, \mathcal{Y}) \equiv \texttt{global cardinality}\,(\mathcal{X}, \mathcal{D}(\mathcal{X}), \mathcal{F}) \qquad (30)$$

Hence, every CSP model fulfilling the above requirements also provides the possibility to compare the bounds consistent `sortedness` propagator to the $\texttt{gcc}_{bnd}$.

5.1.1 Permutation problem

Problem description
Given a integer $n \in \mathbb{N}$ and the set $\mathbb{Z}_n$, the set of integer residues modulo n, the permutation problem $\texttt{perm} - \texttt{n}$ consists in finding all tuples $x = (x_0, \ldots, x_{n-1})$, $y = (y_0, \ldots, y_{n-1})$ $\in \underbrace{\mathbb{N} \times \ldots \times \mathbb{N}}_{n}$ such that x forms a permutation of $\mathbb{Z}_n$, a permutation of y and $\forall i, j \in \mathbb{K}_n, i < j : y_i \leq y_j$. Thus, a solution to $\texttt{perm} - 5$ is $x = (3, 4, 1, 2, 0)$ and $y = (0, 1, 2, 3, 4)$.

Problem representation
We encode the vectors x and y as sets of FDVars $\mathcal{X}$ and $\mathcal{Y}$ and introduce a set $\mathcal{Z}$ such that $|X| = |Y| = |Z| = n$ and $\forall i \in \mathbb{K}_n : x_i \in \mathbb{Z}_n \wedge y_i \in \mathbb{Z}_n \wedge z_i \in \{i\}$. The problem of finding assignments for the $\mathcal{X}$- and $\mathcal{Y}$-variables subject to the above description with the `sortedness` constraint and model the problem as:

1. `sortedness` $(\mathcal{X}, \mathcal{Z})$ - $\mathcal{X}$ is a permutation of $\mathbb{Z}_n$
2. `sortedness` $(\mathcal{X}, \mathcal{Y})$ - $\mathcal{Y}$ is the sorted permutation of $\mathcal{X}$

Obviously, the size of the search space is $\left|\mathcal{S}\right| = n!$ and constructing a single solution can be done in $\mathcal{O}(n)$. Thus, the difficulty of the problem consists in finding all possible permutations x of $\mathbb{Z}_n$.

Results
Table 8 presents the comparison of a $\texttt{perm} - \texttt{n}$ model using the `sortedness` propagator with a $\texttt{perm} - \texttt{n}$ model using the decomposition `d-sort` on platform $[B_2]$. Strikingly, the search tree for the `sortedness` model consists only of solvable spaces, whereas the number of failed spaces in search tree for the `d-sort` model increases with the problem size. Thus, the propagator for `sortedness` outperforms its decomposition by a factor of 6.3 due to a stronger propagation. Apart from testing the `sortedness` propagator against its decomposition we also want to compare it with a sorting constraint of a different constraint solver. As *ILOG* provides no such constraint we choose the *SICStus* platform for a cross-platform comparison. However, *SICStus* provides a `sorting` constraint taking also permutation variables into account. Hence, we compare the above presented $\texttt{sortedness}^{+}$ propagator with *SICStus*' `sorting` propagator. The results of this comparison for the permutation problem as depicted in table 9 show that the $\texttt{sortedness}^{+}$ propagator solves the problem 80% faster on average than *SICStus* and we obtain a first indicator that the `sortedness` propagator and its extension are competitive with respect to other constraint solvers. In the following benchmark problems we want to underline this statement and further analyze whether the `sortedness` constraint is also competitive with the `alldiff` constraint on the same platform.

	sortedness			d-sort
n	solutions	clones	cpu (in millisecs)	cpu (rel in %)
3	6	5	0.033515	376
4	24	23	0.188800	429.5
5	120	119	1.123400	528.24
6	720	719	7.710500	607.73
7	5040	5039	59.650000	713.12
8	40320	40319	521.090000	825.34
9	362880	362879	5202.850000	947.36
				632.47

Table 8. `sortedness` vs. `d-sort` on $[B_2]$

	sortedness$^+$			sorting
n	solutions	clones	cpu (in millisecs)	cpu (rel in %)
3	6	5	0.055780	258.59
4	24	23	0.300000	197.92
5	120	119	1.820300	171.42
6	720	719	12.671500	162.77
7	5040	5039	100.272500	158.95
8	40320	40319	893.825000	155.95
9	362880	362879	8820.050000	154.39
				180

Table 9. `sortedness`$^+$ vs. `sorting` on $[B_2]$

5.1.2 All-interval series

Problem description - Prob007[11]
Given the twelve standard pitch-classes $(c, \#c, d, \dots, h, c)$, each represented by numbers $0, 1, \dots, 11$, find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). That is, for each of the intervals, there is a pair of neighboring pitch-classes in the series, between which this interval appears. The problem of finding such a series can be easily formulated as an instance of a more general arithmetic problem on $\mathbb{Z}_n$. Given an integer $n \in \mathbb{N}$, find a vector $x = (x_0, \dots, x_{n-1})$, such that x is a permutation of $\mathbb{Z}_n$ and the interval vector $v = (|x_1 - x_0|, \dots, |x_{n-2} - x_{n-1}|)$ is a permutation of $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$. A vector v satisfying these conditions is called an *all-interval series* of size n. The problem of finding such a series is the *all-interval series problem* of size n we refer to as `allint − n`.

Problem representation
As in the previous example we encode the vectors x and v as sets of FDVars $\mathcal{X}$ and $\mathcal{V}$ and introduce sets $\mathcal{Z}$, $\mathcal{Z}^*$ such that $|X| = |Z| = n$, $|V| = \left|\mathcal{Z}^*\right| = n - 1$, $\forall i \in \mathbb{K}_n : x_i \in \mathbb{Z}_n \wedge y_i \in \mathbb{Z}_n^* \wedge z_i \in \{i\}$ and $\forall i \in \mathbb{K}_{n-1} : z_i^* \in \{i + 1\}$. Thus, we can model the problem as:

1. `sortedness` $(\mathcal{X}, Z)$ - $\mathcal{X}$ is a permutation of $\mathbb{Z}_n$
2. `sortedness` $(\mathcal{V}, \mathcal{Z}^*)$ - $\mathcal{V}$ is a permutation of $\mathbb{Z}_n^*$
3. $\forall i \in \mathbb{K}_{n-2} : |x_{i+1} - x_i| = v_i$

The implementation of this problem is provided with the *Gecode* library and uses the work of Gent *et al.* to reduce the size of the search space by static symmetry breaking [12] that eliminates:

1. the negation of the sequence x by adding $x_o < x_{n-1}$
2. the reversal of the sequence x by adding $v_o < v_{n-2}$

Results
Comparing the `sortedness` propagator with the bounds consistent `alldiff` propagator on the `allint − n` problem, table 10 shows that the `alldiff` $_{bnd}$ propagator is 30% faster on average than the `sortedness` propagator demonstrating that both propagators belong to the same complexity class.

	sortedness				alldiff $_{bnd}$
	solutions	fails	clones	cpu (in millisecs)	cpu (rel in %)
4	1	3	3	0.055037	78.93
5	2	7	8	0.171850	77.1
6	6	20	25	0.646850	74.15
7	8	69	76	2.276500	72.96
8	10	255	264	9.265500	71.84
9	30	942	971	38.071667	71.41
10	74	3845	3918	165.465000	70.44
11	162	16695	16856	770.300000	73.83
12	332	77335	77666	3789.920000	73.11
					73.75

Table 10. sortedness vs. alldiff on $[B_2]$

	sortedness$^+$				sorting
n	solutions	fails	clones	cpu (in millisecs)	cpu (rel in %)
4	1	3	3	0.079906	410.86
5	2	7	8	0.255200	363.58
6	6	20	25	0.964850	323.56
7	8	69	76	3.434400	306.14
8	10	255	264	14.023000	289.31
9	30	942	971	58.435000	281.42
10	74	3845	3918	256.640000	278.48
11	162	16695	16856	1195.300000	321.91
12	332	77335	77666	5992.975000	309.2
					320.5

Table 11. sortedness$^+$ vs. sorting on $[B_2]$

Moreover table 11 depicts that the mean runtime of the sortedness$^+$ model is 3.2 times faster on average than for the corresponding sorting constraint in *SICStus* as it was the case for the permutation problem. A third example where we can evaluate the competitiveness of the sortedness and the sortedness$^+$ propagators is *Langford's number problem*.

5.1.3 Langford's number problem

Problem Description - Prob024[11]
Consider two sets of the numbers from 1 to 4. The problem is to arrange the eight numbers in the two sets into a single sequence in which the two 1's appear one number apart, the two 2's appear two numbers apart, the two 3's appear three numbers apart, and the two 4's appear four numbers apart. The problem generalizes to the $L(k, n)$ problem, which is to arrange k sets of numbers 1 to n, so that each appearance of the number m is m numbers apart from the last. Thus the following sequence

```
x = < 1 9 1 6 1 8 2 5 7 2 6 9 2 5 8 4 7 6 3 5 4 9 3 8 7 4 3 >
```

is a solution to the $L(3, 9)$ problem. A complete history of Langford's number problem can be found on John E. Miller webpage [16]. For our benchmark we focus only on the $L(k, n)$ problem for $k = 2$.

Problem Representation
The set $\mathcal{P}$ of FDVars represents the vector containing the position of each number in the sequence of Langford's numbers. Additionally, we use the set $\mathcal{Z}$ such that, $\forall i \in \mathbb{K}_{k\cdot n}$: $p_i \in \mathbb{Z}_n \wedge z_i \in \{i\}$. Then we can model the problem as follows:

1. sortedness $(\mathcal{P}, \mathcal{Z})$
2. $\forall i \in \mathbb{K}_n : \forall j \in \mathbb{K}_{k-2} : p_{i\cdot k+j} + i + 2 = p_{i\cdot k+(j+1)}$

Again, due to Smith [37] we further introduce a set of FDVars $\mathcal{Y}$ dual to $\mathcal{X}$ representing the numbers in the Langford sequence such that $\forall i \in \mathbb{K}_n : \forall j \in \mathbb{K}_k : y_{p_{i\cdot k+j}} = i + 1$ and order the numbers in the sequence by adding $y_0 < y_{n-1}$.

Results
Like in the previous examples table 12 emphasizes that the sortedness propagator is competitive with the alldiff propagator since the runtimes for the examples using the alldiff propagator are only 0.5% faster on average than those using the sortedness propagator.

	`sortedness`				`alldiff` $_{bnd}$
	solutions	fails	clones	cpu (in millisecs)	cpu (rel in %)
2	0	1	0	0.006348	93.84057971
3	1	0	0	0.014367	98.802812
4	1	4	4	0.085656	104.3592977
5	0	14	13	0.297265	78.7109145
6	0	48	47	1.057970	129.5301379
7	26	166	191	5.066250	121.221811
8	150	620	769	23.117000	84.89423368
9	0	4015	4014	127.781000	85.44854086
10	0	21959	21958	753.633333	85.93701627
11	17792	113804	131595	5124.200000	84.03750829
12	108144	794213	902356	36522.650000	86.98150326
					95.79675956

Table 12. `sortedness` vs. `alldiff` on $[B_2]$

	`sortedness`$^+$				`sorting`
n	solutions	fails	clones	cpu (in millisecs)	cpu (rel in %)
2	0	1	0	0.006735	1010.126206
3	1	0	0	0.019796	833.0218226
4	1	4	4	0.108825	378.6951528
5	0	14	13	0.361090	315.3881304
6	0	48	47	1.330450	299.5415085
7	26	166	191	6.417750	285.6452807
8	150	620	769	29.562500	302.1919662
9	0	4015	4014	165.031000	298.6275306
10	0	21959	21958	991.133000	300.7669001
11	17792	113804	131595	6775.750000	296.7612441
12	108144	794213	902356	47504.650000	313.3097497
					310.1

Table 13. `sortedness`$^+$ vs. sorting on $[B_2]$

Comparing the Langford problem between the `sortedness`$^+$ and *SICStus* ' `sorting` propagator table 13 highlights that `sortedness`$^+$ runs 3.1 times faster on average than the corresponding `sorting` constraint in *SICStus* . Hence we can state that the implementation of the `sortedness` propagator is able to compete with an equivalent implementation of `alldiff` $_{bnd}$ propagator and moreover is also competitive with respect to the sorting constraint in *SICStus* .

5.2 Global Cardinality

Evaluating whether the propagation algorithms for the `global cardinality` constraint are able to improve constraint propagation we compare them to a propagation algorithm for a cardinality constraint combining several weaker constraints that are defined as:

Definition 42 (D-GCC). *Given a finite set of FDVars* $\mathcal{X} := \{x_0 : D_0, \ldots, x_{n-1} : D_{n-1}\}$*, the union of all variable domains* $\mathcal{D}(\mathcal{X})$*, a finite set* $V \subseteq \mathcal{D}(\mathcal{X})$ *of values for the variables with* $|V| = m$ *and a finite set* $\mathcal{F} \subset \{(l,u) \in \mathbb{K}_m \times \mathbb{K}_m \mid l \leq u\}$ *with* $\left|\mathcal{F}\right| = |V| = m$*, we define* `d-gcc` *as:*

$$\texttt{d-gcc}\,(\mathcal{X}, V, \mathcal{F}) \overset{def.}{=} \begin{cases} \bigcap\limits_{0 \leq j \leq m} \left(\texttt{atleast}\,(\mathcal{X}, v_j, l_j) \cap \texttt{atmost}\,(\mathcal{X}, v_j, u_j)\right) & \text{if } |V| \leq \left|\mathcal{D}(\mathcal{X})\right| \\ \bigcap\limits_{0 \leq j \leq m} \left(\texttt{exactly}\,(\mathcal{X}, v_j, l_j)\right) & \text{if } |V| = \left|\mathcal{D}(\mathcal{X})\right| \end{cases}$$

where

$$\texttt{atleast}\,(\mathcal{X}, v_j, l_j) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{c1}}(\mathcal{X}) | l_j \leq \#(\alpha, v_j)\}$$
$$\texttt{atmost}\,(\mathcal{X}, v_j, u_j) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{c1}}(\mathcal{X}) | \#(\alpha, v_j) \leq u_j\}$$
$$\texttt{exactly}\,(\mathcal{X}, v_j, c_j) \overset{def.}{=} \{\alpha \in \texttt{ass}_{\texttt{c1}}(\mathcal{X}) | \#(\alpha, v_j) = c_j\}$$

5.2.1 Pathological

In order to test, whether the `global cardinality` propagator is competitive with its decomposition and with the `alldiff` constraint our first test is the modified *pathological problem* [20] (originally by Puget[23]).

Problem Description
Given $n \in \mathbb{N}$, the *pathological problem* P(k, n) consists in finding a sequence x of length $|x| = k \cdot n$ such that every number $v \in \{0, \ldots, 2 \cdot n\}$ appears exactly k times in x.

Problem Representation
We represent the sequence x using a set of FDVars $\mathcal{X}$, such that $\left|\mathcal{X}\right| = k \cdot (2 \cdot n + 1)$ where $\forall i \in \mathbb{K}_{c \cdot n} : x_i : D_i = [0..n]$ and $\forall i \in \{k \cdot n, \ldots, k \cdot 2 \cdot n\} : x_i : D_i = [n..i]$. Thus, we can model the pathological problem as:

1. gcc $(x, \{0, \ldots, 2 \cdot n\}, \mathcal{F})$, where $\forall i \in \mathbb{K}_{2 \cdot n+1} : (l_i, u_i) = (k, k)$

A possible solution to P(2, 4) is the sequence

```
x = < 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 >
```

Results
Table 14 contains the comparison between the gcc $_{dom}$ propagator and its decomposed equivalent d-gcc $_{dom}$ on the *pathological problem* $P(2, n)$, where n denotes the number of problem variables. From the relative runtimes for the d-gcc $_{dom}$ propagator it follows that the gcc $_{dom}$ outperforms its decomposition by a mean factor of 50.

n	gcc $_{dom}$ cpu (in millisecs)	d-gcc $_{dom}$ cpu (rel in %)
100	3.161700	393.25
500	112.941667	1751.35
1000	451.795000	3387.52
1500	1017.660000	5121.89
2000	1898.825000	6354.38
2500	3087.100000	8206.96
3000	4715.600000	9948.51
		5023.41

Table 14. gcc $_{dom}$ vs. d-gcc $_{dom}$ on B_2

5.2.2 Social golfers

Problem Description - Prob010[11]
The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion. Possible variants of the above problem include: finding a 10-week schedule with *maximum socialisation*; that is, as few repeated pairs as possible (this has the same solutions as the original problem if it is possible to have no repeated pairs), and finding a schedule of minimum length such that each golfer plays with every other golfer at least once (*full socialisation*). The problem can easily be generalized to that of scheduling g groups of s golfers over w weeks, such that no golfer plays in the same group as any other golfer twice (i.e. maximum socialisation is achieved).

Problem Representation
As the commonly used finite set model of the golfer problem is not well-suited for the application of the `gcc` propagator we choose a more naive finite domain integer model for the *social golfers* problem as presented in [3]. This model introduces a set $\mathcal{G}$ of FDVars such that $|G| = g \cdot s \cdot w$ and $g_{w,p}$ represents the group number of player p in week w and $\forall i \in \mathbb{K}_w \forall j \in \mathbb{K}_{g \cdot s} : g_{i,j} \in \{1, \cdots, g\}$. Hence, we can model the problem as follows:

1. $\forall i \in \mathbb{K}_w :$ `gcc` $(\{G_{i,j} | 0 \le j < g \cdot s\}, \{1, \ldots, g\}, \mathcal{F})$, where $\forall i \in \mathbb{K}_g : (l_i, u_i) = (s, s)$.
2. $0 \le p < q \le g \cdot s - 1 \sum_{0 \le i < w} \left(G_{i,p} == G_{i,q}\right) \le 1$ - for maximum socialisation.

Note that this is naive approach, since the constraint for maximum socialisation uses quadratically many variables in the size of the players. Therefore we add symmetry breaking constraint to our model as proposed by Barnier and Brisset[3]:

1. Fix the first week:
   $\forall i \in \mathbb{K}_g : \forall i \in \mathbb{K}_p : G_{0,g \cdot s+i} = i$
2. Players in the first group of the second week have smallest possible group numbers:
   $\forall p \in \mathbb{K}_s : G_{1,p} = p + 1$
3. Players with small group numbers are put in small groups:
   $\forall i \in \mathbb{K}_w : \forall j \in \mathbb{K}_p : G_{i,p} \le p + 1$
4. Groups in the second week are ordered:
   $\forall i \in \mathbb{K}_g : \forall j \in \{0, \ldots, p-1\} : G_{1,i \cdot s+j} < G_{1,i \cdot s+i+1}$
5. Groups in the second week are ordered lexicographically:
   $\forall i \in \{0, \ldots, g-1\} : \forall p \in \mathbb{K}_s : G_{1,g \cdot s+p} \le G_{1,(g+1) \cdot s+p}$

Results
As *ILOG* does not provide an explicit `gcc` the results in table 15 have been measured as comparison to a semantically equivalent `IloDistribute` algorithm[30] with domain-consistency on platform $[B_1]$. As we used the above described integer model for the social golfers problem to perform the benchmarks for this section interesting instances like $8 - 4 - 9$ did not terminate in less than 30 minutes neither on the *Gecode* platform nor on the *ILOG* platform. Hence, we have tested instances $g - s - w$ of the social golfers problem such that: $g \in \{5, 8\}$, $s \in \{2, 3, 4\}$, $w \in \{2, \ldots, 9\}$. As table 15 shows the `gcc` $_{dom}$ propagator found a first solution 4.9 times faster on average than the equivalent *ILOG* constraint for test instances with $g = 5$ and 4.0 times faster on average for test instances with $g = 8$. Thus, these benchmarks for the integer model of the social golfers problem indicate that the `global cardinality` propagator is obviously competitive with equivalent global constraints of other constraint solvers.

g-s-w	`gcc` $_{dom}$ cpu (in milliseconds)	`IloDistribute` cpu (rel in %)
5 - 2 - 2	0.197667	3562.3
5 - 2 - 3	0.425000	1883.13
5 - 2 - 4	0.923000	875.41
5 - 2 - 5	1.646667	603.82
5 - 2 - 6	2.385000	463.73
5 - 2 - 7	4.070000	356.02
5 - 2 - 8	4.425000	339.44
5 - 2 - 9	4.295000	344.35
5 - 3 - 2	0.338000	2582.1
5 - 3 - 3	1.387500	871.35
5 - 3 - 4	3.045000	500.82
5 - 3 - 5	4.645000	409.15
5 - 3 - 6	715.500000	354.65
5 - 3 - 7	2511.500000	319.99
		494.43

g-s-w	`gcc` $_{dom}$ cpu (in milliseconds)	`IloDistribute` cpu (rel in %)
8 - 2 - 2	0.534000	1860.67
8 - 2 - 3	1.875000	676
8 - 2 - 4	3.635000	443.88
8 - 2 - 5	6.030000	337.23
8 - 2 - 6	9.330000	251.02
8 - 2 - 7	14.600000	225
8 - 2 - 8	29.700000	210.54
8 - 2 - 9	64.700000	206.68
8 - 3 - 2	1.411667	1326.8
8 - 3 - 3	5.660000	449.12
8 - 3 - 4	11.875000	277.31
8 - 3 - 5	45.600000	282.92
8 - 3 - 6	4868.000000	323.31
8 - 4 - 2	2.835000	352.73
8 - 4 - 3	11.250000	280.92
8 - 4 - 4	1486.500000	397.17
		402.71

Table 15. `gcc` $_{dom}$ vs. `IloDistribute` on $[B_1]$

5.2.3 Car sequencing

In order to underline the statement from table 15 that the `gcc` propagator is competitive with *ILOG* 's `IloDistribute` implementation this section compares these two constraints in a model for the car sequencing problem.

Problem Description - Prob001[11]
In Automotive industry there are a number of cars to be produced. They are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most half of the cars passing along the line, the sequence must be built so that at most 1 car in any 2 requires that option. The problem has been shown to be NP-complete (Gent 1999).

Problem Representation
A set of FDVars $\mathcal{X}$ represents the assembly line of all cars and a set of FDVars $\mathcal{Y}$, such that $\forall y_i : E_i \in \mathcal{Y} : E = [0..1]$ representing the options a configuration of a certain car class needs. Next the required data is taken from a problem data sheet looking as follows [8]:

10 5 6		$\rightarrow$ number of cars *nc*, number of options *no*, number of classes *ncl*
1	2 1 2 1	$\rightarrow$ the maximum number of cars with that option in a block
2	3 3 5 5	$\rightarrow$ the block size to which the maximum number refers to
1	1 0 1 1 0	
1	0 0 0 1 0	
2	0 1 0 0 1	
2	0 1 0 1 0	$\rightarrow$ line for c_3: $demand(c_3) = 2$, c_3 requires options 1 and 3
2	1 0 1 0 0	
2	1 1 0 0 0	

A valid sequence for the above data sheet is:

0 1 5 2 4 3 3 4 2 5 $\rightarrow$ assembly line with classes of cars
1 0 1 0 1 0 0 1 0 1 $\rightarrow$ options the cars are using
0 0 1 1 0 1 1 0 1 1
1 0 0 0 1 0 0 1 0 0
1 1 0 0 0 1 1 0 0 0
0 0 0 1 0 0 0 0 1 0

The model has been encoded as follows:

1. There have to be as many cars of configuration c_i as required by the first column of the data sheet:
   `gcc` $(\mathcal{X}, \{c_0, \dots, c_{ncl-1}\}, \mathcal{F})$, where $\forall i \in \mathbb{K}_{ncl}(l_i, u_i) = (demand(i), demand(i))$
2. Option i occurs at most p times in a block s of size $|s| = q$: $\forall i \in \{0, \dots, nc - q + 1\}$:
   $\forall j \in \{i + 1, \dots, i + q\} : |s| = q \wedge s_{j-i} = y_{no \cdot nc + j} \wedge \sum_{j \in \mathbb{K}_q} s_q \leq p$

Additionally we added a redundant constraint on the $\mathcal{Y}$-variables as described in [8]: Consider an option o_i with capacity p/q, that is at most p cars out of a block s of x with length $|s| = q$ require option o_i. If the total number of cars using option o_i is M and we consider a block s of x, containing $v \leq p$ cars requiring option o_i then all remaining blocks t contain must contain $M - v$ cars that require o_i.

Results

In table 5.2.3 we compare the runtime of the `gcc` propagator with the runtimes of the `IloDistribute` implementation for finding a first solution to the above specified car sequencing problem. The instances we used were two instances from *A filtering algorithm for global sequencing constraints* by [29] denoted as RP0 and RP1, the example from *Solving the Car-Sequencing Problem in Constraint Logic Programming* denoted as DB1 and further examples provided by CSPlib[11]. Like in the preceding car sequencing example table 5.2.3 underlines that the `gcc` propagator was a factor 1.17 faster on average in finding a first assignment for the assembly line than the `IloDistribute` constraint.

instance	`gcc` $_{dom}$ cpu (in millisecs)	`IloDistribute` cpu (rel in %)
DB1	0.510000	1000.980392
RP0	40.150000	133.063512
RP1	45.850000	129.661941
60-01	126.050000	124.672749
60-02	121.750000	127.392197
60-03	152.200000	110.052562
60-04	164.800000	118.719660
60-05	139.900000	112.830593
60-06	960.800000	148.407577
60-07	113.400000	132.936508
60-08	268.200000	152.628635
60-09	158.700000	114.744802
60-10	134.150000	119.567648
		127.28

Table 16. `gcc` $_{dom}$ vs. `IloDistribute` on $[B_1]$

5.2.4 Sports-League Scheduling

Problem Description - Prob026[11]

As proposed by [11] the *sports-league scheduling* problem which is also referred to as the *round robin tournament* is specified as follows: The problem is to schedule a tournament of n teams over $n-1$ weeks, with each week divided into $\frac{n}{2}$ periods, and each period divided into 2 slots. The first team in each slot plays at home, whilst the second plays away. A tournament must satisfy the following three constraints:

1. Every team plays once a week
2. Every team plays at most twice in the same period over the tournament
3. Every team plays every other team.

An example schedule for 8 teams is:

	week 1	week 2	week 3	week 4	week 5	week 6	week 7
period 1	0 v 1	0 v 2	4 v 7	3 v 6	3 v 7	1 v 5	2 v 4
period 2	2 v 3	1 v 7	0 v 3	5 v 7	1 v 4	0 v 6	5 v 6
period 3	4 v 5	3 v 5	1 v 6	0 v 4	2 v 6	2 v 7	0 v 7
period 4	6 v 7	4 v 6	2 v 5	1 v 2	0 v 5	3 v 4	1 v 3

Table 17. Sports-League Schedule for n = 8

One extension of the problem is to double round robin tournaments in which each team plays every other team (as before) but now both at home and away. This is often solved by repeating the round robin pattern, but swapping home games for away games in the repeat.

Problem Representation

Analogously to the model presented in [14] we use sets of FDVars, ; $\mathcal{A}$, $\mathcal{G}$, where m̋odels all home teams, $\mathcal{A}$ models all of away teams and set $\mathcal{G}$ models all numbers of a match $m = (h_i, a_i)$ between a home and an away team, such that $|\mathcal{H}| = |\mathcal{A}| = \frac{n}{2} \cdot n$ and $|\mathcal{G}| = \frac{n}{2} \cdot (n-1)$. As mentioned in the above description the model uses the following constraints:

1. $\forall i \in \{0, \ldots n\} : \left(\forall j \in \mathbb{K}_{\frac{n}{2}} : col^i with \left|col^i\right| = n \wedge h_{j \cdot n+i} \in col^i \wedge a_{j \cdot n+i} \in col^i\right)$ $\wedge$`alldiff`(col^i) - Every team plays once a week
2. $\forall j \in \mathbb{K}_{\frac{n}{2}} : \left(\forall i \in \mathbb{K}_{2 \cdot n} : row^j with \left|row^j\right| = 2 \cdot n \wedge h_{j \cdot n+\frac{i}{2}} \in row^i \wedge a_{j \cdot n+\frac{i}{2}} \in row^j\right)$ $\wedge$`gcc`$(row^j, \{1, \ldots, n\}, \mathcal{F})$, where $\forall i \in \mathbb{K}_n : (l_i, u_i) = (2, 2)$ - Every team plays at most twice in the same period over the tournament
3. `alldiff`$(\mathcal{G})$ - Every match occurs only once

Apart from these constraints the model also uses a initial round robin schedule[14] restricting the domains of the variables such that only those matches can be scheduled that the round robin schedule allows. Furthermore we break the symmetry of interchanging the teams of a match by stating that $\forall i \in \mathbb{K}_{\frac{n}{2}} : \forall j \in \mathbb{K}_n : h_{i \cdot n+j} < a_{i \cdot n+j}$ and fix the first pair such that $h_{0,0} = 1$ and $a_{0,0} = 2$. Apart from this we order the home weeks in the first week such that $\forall i \in \{0, \ldots, \frac{n}{2} - 1\} : h_{i \cdot n} < h_{(i+1) \cdot n}$.

Results
Comparing the `gcc` $_{dom}$ propagator with the `IloDistribute` in case of the sports league scheduling problem we can see in table 5.2.4 that the mean runtime of the `gcc` $_{dom}$ propagator is 1.2 times faster on average than the corresponding *ILOG* constraint.

	`gcc` $_{dom}$			`IloDistribute`
n	failures	clones	cpu (in milliseconds)	cpu (rel in %)
6	2	4	0.619000	841.03
8	18	25	4.135000	232.89
10	2	12	3.490000	315.33
12	121	142	39.610000	123.48
14	1984	2013	627.500000	103.63
16	2092	2132	794.400000	100.77
18	236	293	160.500000	109.53
20	1778	1851	984.800000	99.6
22	2456	2549	1568.000000	100.96
24	1837	1949	1935.000000	95.61
26	9298	9428	7034.500000	91.63
28	68048	68105	47486.000000	103.57
30	6033	6094	6510.000000	98.62
				121.2

Table 18. `gcc` $_{dom}$ vs. `IloDistribute` on $[B_1]$

Summary Although benchmarking itself “is a difficult business” [41] and “even more so for complex constraint programming systems”[41] the experiments conducted during this section allow the following conclusions: propagation algorithms for global constraints clearly outperform their local decompositions as we have seen for `sortedness` and `d-sort` and `global cardinality` and `d-gcc` . Hence, the use of global constraints in a constraint solver definitely increases the efficiency of constraint propagation. Furthermore, we saw that in case the `sortedness` propagator can be used instead of a bounds consistent `alldiff` propagator, the time complexity of the `sortedness` propagator is not dramatically worse than the complexity of `alldiff` $_{bnd}$, that is they belong to the same complexity class. Moreover, a cross-platform comparison with *SICStus* showed that the propagator for the `sortedness` constraint as well as its extension are definitely competitive with the `sorting` constraint as provided in *SICStus* . Finally, the same result directly transfers to the global cardinality propagator being at least as efficient as *ILOG* ’s `IloDistribute` constraint with respect to semantics and consistency level.

6 Conclusion & Contributions

The contributions of this paper are as follows: we recapitulated basic definitions and notations of *constraint programming* focusing on the principle of *constraint propagation*. We discussed advanced propagation algorithms for global constraints developed in the work of Thiel [39], Quimper *et al.* [24] and Quimper *et al.* [25] and provided a possible extension for the `sortedness` constraint[39]. Moreover, we also gave a trace of how propagation is handled in *Gecode* , a generic constraint development environment and finally the above propagation algorithms have been implemented as part of this *Gecode* -library emphasizing that these propagation algorithms are not only theoretically smart but also practically increase performance of constraint propagation.

7 Future Work

In so far as further research on the discussed propagation algorithms is concerned we want to point out that there remains interesting aspects to look at that would have gone beyond the scope of this thesis and the related implementation: It would be interesting to use the extended `sortedness` constraint in order to model the *job-shop* scheduling problem as referenced in the work of [42]. Furthermore, one could implement the bounds consistent propagator for the `global cardinality` as presented by Katriel and Thiel in [17] and compare it to the propagator we discussed during the course of this paper. Finally it would also be nice to further exploit the knowledge of the `global cardinality` constraint to implement and evaluate propagation algorithms for the `cardinality matrix constraint`[28] and the `global sequencing constraint` [29] as proposed by Régin and Puget.

8 Acknowledgement

First of all, I want to thank Gert Smolka, the responsible professor, for giving me the opportunity of doing my *Fortgeschrittenenpraktikum* at his chair for programming systems at Saarland University. I also want to thank my supervisor Guido Tack who introduced me to the topic of *Constraint Programming* and who was very patient answering all my questions and helping me out if got stuck. Further I want to thank Marco Kuhlmann and Guido Tack for their work in the *Constraint Programming* lecture formalizing very nicely the necessary notation I used throughout my paper Last, but not least I want to thank Christian Schulte, chief supervisor of the *Gecode* - project for the possibility of taking part in that project.

9 References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*, volume - of *Addison-Wesley Series in Computer Science and Information Processing*. Addison-Wesley, Reading,Massachusetts - Menlo Park,California - London.

[2] Baptiste, P., Pape, C. L., and Nuijten, W. (2001). *Constraint-Based Scheduling*. Kluwer Academic Publishers.

[3] Barnier, N. and Brisset, P. (2002). Solving the kirkman's schoolgirl problem in a few seconds.

[4] Barták, R. (1999). Constraint programming: In pursuit of the holy grail. In *In Proceedings of the Week of Doctoral Students (WDS99)*, volume Part IV, Prague. MatFyzPress.

[5] Berge, C. (1979). *Graphs and hypergraphs*, volume 6 of *North - Holland Mathematical Library*. North-Holland, Elsevier, Amsterdam, repr. of the 2., rev. ed. edition.

[6] Bleuzen-Guernalec, N. and Colmerauer, A. (2000). Optimal narrowing of a block of sortings in optimal time. *Constraints: An International Journal*, **5**(1/2), 85–118m.

[7] Cormen, T. H., Leierson, C. E., and Rivest, R. L. (2001). *Introduction to Algorithms, second edition*. MIT Press, Cambridge, Massachusetts - London, England.

[8] Dincbas, M., Simonis, H., and Hentenryck, P. V. (1988). Solving the car-sequencing problem in constraint logic programming. In *ECAI*, pages 290–295.

[9] Emden-Weinert, T., Hougardy, S., Kreuter, B., Prömel, H. J., and Steger, A. (1996). Einführung in Graphen und Algorithmen.

[10] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[11] Gent, I. and Walsh, T. (1999). Csplib: a benchmark library for constraints. Technical report, Technical report APES-09-1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[12] Gent, I., McDonald, I., and Smith, B. (2003). Conditional symmetry in the all-interval series problem. In B. Smith, I. Gent, and W. Harvey, editors, *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 55–65.

[13] Glover, F. (1967). Maximum matching in a convex bipartite graph. *Naval Research Logistics Quarterly*, **14**(3), 313–316.

[14] Hentenryck, P. V., Michel, L., Perron, L., and Régin, J.-C. (1999). Constraint programming in opl. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116.

[15] Hopcroft, J. E. and Karp, R. M. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM: Journal of Computing*, **2**(4), 225–231.

[16] John E. Miller (20064). Langford's problem. Available from `http://www.lclark.edu/~miller/langford.html`.

[17] Katriel, I. and Thiel, S. (2003). Fast bound-consistency for the global cardinality constraint. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003 : 9th International Conference, CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 437–451, Kinsale, Ireland. Springer.

[18] Kuhlmann, M. and Tack, G. (2005a). Constraint satisfaction problems.

[19] Kuhlmann, M. and Tack, G. (2005b). Propagators.

[20] López-Ortiz, A., Quimper, C.-G., Tromp, J., and van Beek, P. (2003a). A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 245–250, Acapulco, Mexico.

[21] López-Ortiz, A., Quimper, C.-G., Tromp, J., and van Beek, P. (2003b). A fast and simple algorithm for bounds consistency of the alldifferent constraint, technical report. Technical report, School of Computer Science, University of Waterloo, Waterloo, Canada, Acapulco, Mexico.

[22] Mehlhorn, K. (1984). *Data Structures and Algorithms*, volume 2 Graph Algorithms and NP-Completeness of *EATCS Monographs*. Springer Verlag.

[23] Puget, J.-F. (1998). A fast algorithm for the bound consistency of alldiff constraints. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 359–366, Menlo Park, CA, USA. American Association for Artificial Intelligence.

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

[25] Quimper, C.-G., van Beek, P., López-Ortiz, A., and Golynski, A. (2004). Improved Algorithms for the Global Cardinality Constraint. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, volume 3528, Toronto, Canada.

[26] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of 12th National Conference on AI (AAAI'94)*, volume 1, pages 362–367, Seattle.

[27] Régin, J.-C. (2005). Combination of Among and Cardinality Constraints. In R. Barták and M. Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: Second International Conference, CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 288 – 303, Prague, Czech Republic. Springer-Verlag.

[28] Régin, J.-C. and Gomes, C. P. (2004). The cardinality matrix constraint. In *CP*, pages 572–587.

[29] Régin, J.-C. and Puget, J.-F. (1997). A filtering algorithm for global sequencing constraints. In *CP*, pages 32–46.

[30] S.A., I. (2000). *ILOG Solver 5.0:Reference Manual*. ILOG S.A.

[31] Schulte, C. (2000). *Programming Constraint Services*. Doctoral dissertation, Saarland University, Faculty for Natural Sciences I, Department of Computer Science, Saarbrücken, Germany.

[32] Schulte, C. (2006). *Gecode 1.0.0 Reference Documentation*, 1.0.0 edition.

[33] Schulte, C. and Smolka, G. (2004). *Finite Domain Constraint Programming in Oz. A Tutorial, 1.3.0 edition.*

[34] Schulte, C. and Stuckey, P. J. (2004). Speeding up constraint propagation. In M. Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada. Springer-Verlag.

[35] Schulte, C. and Tack, G. (2005). Views and iterators for generic constraint implementations. In C. Schulte, F. Silva, and R. Rocha, editors, *Proceedings of the Fifth International Colloqium on Implementation of Constraint and Logic Programming Systems*, pages 37–48, Sitges, Spain. To appear.

[36] Shapira, A. (1997). An exact performance bound for an $o(m+n)$ time greedy matching procedure. *The Electronic Journal of Combinatorics*, **4**.

[37] Smith, B. (2000). Modelling a permutation problem. In *Proceedings of ECAI'2000 Workshop on Modelling and Solving Problems with Constraints*. Also available as Research Report from http://www.comp.leeds.ac.uk/bms/papers.html.

[38] The Gecode team (2006). Generic constraint development environment. Available from `http://www.gecode.org`.

[39] Thiel, S. (2004). *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. Doctoral dissertation, Saarland University, Faculty for Natural Sciences I, Department of Computer Science, Saarbrücken, Germany.

[40] Tsunetomo, Y. (1716). *The Hagakure - A Code to the Way of the Samurai*.

[41] Wallace, M., Schimpf, J., Shen, K., and Harvey, W. (2004). On benchmarking constraint logic programming platforms. response to fernandez and hill's "a comparative study of eight constraint programming languages over the boolean and finite domains". *Constraints*, **9**(1), 5–34.

[42] Zhou, J. (1973). A permutation-based-approach for solving the job-shop problem. *Constraints: An International Journal*, **2**(2), 185–213.