Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Sven Thiel

Saarbrücken
[?]

Datum des Kolloquiums: [?]

Dekan der naturwissenschaftlich-technischen Fakultät I:
Professor Dr. Philipp Slusallek[?]

Vorsitzender der Prüfungskomission:
[?]

Gutachter:
Professor Dr. Kurt Mehlhorn, MPI für Informatik, Saarbrücken
Professor Dr. Gert Smolka, Universität des Saarlandes, Saarbrücken

Abstract

This thesis consists of two parts. In the first part we present propagation algorithms, which are used for solving constraint satisfaction problems (CSP). One approach to solve a CSP is based on interleaving constraint propagation and search. The task of a propagation algorithm is to prune portions of the search space which do not contain a solution so that the search does not have to explore them. We present propagation algorithms for the following constraints: *Sortedness*, *Alldiff*, *WeightedPartialAlldiff* and *NonOverlapping* (of two convex polygons).

The second part deals with a tree processing problem, which is represented as a dominance graph. The task is to assemble a collection of tree fragments into a tree T such that the ancestor relation of T satisfies some given constraints. We discuss efficient algorithms for deciding whether a dominance graph D has a solved form and for enumerating all (minimal) solved forms of D.

Zusammenfassung

Diese Arbeit besteht aus zwei Teilen. Im ersten Teil behandeln wir Propagierungsalgorithmen zum Lösen von Constraint-Problemen. Ein Lösungsansatz basiert darauf, Constraint-Propagierung und Suche abzuwechseln. Durch die Propagierung werden Teile des Suchraumes eliminiert, die keine Lösung enthalten. Dadurch verringert sich der Raum, der von der Suche exploriert werden muß, und die Lösung(en) werden oftmals schneller gefunden als durch Suche alleine. Wir beschreiben Propagierungsalgorithmen für folgende Constraints: *Sortedness*, *Alldiff*, *WeightedPartialAlldiff* und *NonOverlapping*.

Der zweite Teil behandelt ein Baumverarbeitungsproblem, das durch einen Dominanzgraphen beschrieben wird. Das Problem besteht darin, Baumfragmente so zu einem Baum zusammen zu setzen, daß bestimmte Anforderungen an die Vorfahr-Relation des Baumes erfüllt sind. Wir entwickeln einen Linearzeit Lösbarkeitstest und effiziente Algorithmen zum Aufzählen aller (minimalen) gelösten Formen eines Dominanzgraphen.

Acknowledgements

First I want to thank Prof. Dr. Gert Smolka, because this thesis would not have been written, if he had not roused Prof. Dr. Kurt Mehlhorn's interest in the field of constraint programming. Shortly after becoming my advisor, Kurt Mehlhorn introduced the field to me. I thank him for helping me with invaluable advice and guidance throughout my master's and doctoral studies whenever I needed him.

In the last years, I had a lot of interesting discussions – some work related, some not – with many different members of our group and I enjoyed the nice working atmosphere. I want to thank all members of AG1 for that, in particular my room-mates Christian Lennerz and Thomas Warken.

During my studies I collaborated with many people. I received helpful support from Gert Smolka and (current and former) members of his group. I am grateful to Manuel Bodirsky, Martin Henz, Alexander Koller, Tobias Müller, Joachim Niehren and Christian Schulte for their help. Moreover, I had a very fruitful collaboration with Nicolas Beldiceanu and Mats Carlsson. I thank Friedrich Eisenbrand for helpful discussions on number theoretic issues related to the *NonOverlapping*-constraint. A special thanks is devoted to Ernst Althaus who provided many helpful comments on various parts of my work.

I want to thank Ernst Althaus and Irit Katriel for proof-reading my thesis. Of course, all remaining errors are mine.

Finally, I want to thank all the people who accompanied me on the long way of writing this thesis for their encouragement, for their support and for their patience.

Contents

Chapter 1

Road map

The purpose of this chapter is to give the reader an overview of the organisation of this thesis. The thesis is divided in two parts: The first part discusses propagation algorithms for some constraints. The second part deals with dominance graphs, these graphs can be used to represent and solve some tree processing problems arising in computational linguistics.

The first part has the following structure: In Chapter 2, we discuss some prerequisites. Then we present propagation algorithms for the following constraints:

- *Sortedness* and *Alldiff* in Chapter 3:
  The constraint $Sortedness(X_1, \ldots, X_n; Y_1, \ldots, Y_n)$ holds iff sorting the sequence $[X_1, \ldots, X_n]$ (in non-descending order) yields the sequence $[Y_1, \ldots, Y_n]$. The constraint $Alldiff(X_1, \ldots, X_n)$ holds iff $X_1, \ldots, X_n$ are pairwise different. The two constraints are discussed in one chapter because the corresponding propagation algorithms are closely related.

- *WeightedPartialAlldiff* (abbreviated as *WPA*) in Chapter 4:
  The constraint $WPA(X_1, \ldots, X_n; undef; T; W)$ is a generalization of *Alldiff*. Not all assignment variables $X_1, \ldots, X_n$ have to take different values; the special value *undef* may be assigned to several variables. Only those assignment variables which are not equal to *undef* have to take pairwise distinct values. Moreover, with every value different from *undef* that occurs in one of the domains $Dom(X_1)$, ..., $Dom(X_n)$ we associate a weight that is determined by the value-weight table T. The constraint states that $\sum_{i=1}^{n} weight(X_i) = W$, where W is the weight variable.

- *NonOverlapping* in Chapter 5:
  This constraint states that two objects in the two-dimensional plane

$\mathbb{R}^2$ should not overlap. The shape of each object is determined by a convex polygon *Shp* and the position of each object is specified by two variables X and Y. The actual object is obtained by applying the translation vector (X, Y) to *Shp*.

Each of these chapters is roughly organized as follows: First we describe the constraint in an informal way, then we define it formally. After that we present and analyse a propagation algorithm. We conclude with a discussion of related work. The fact that we talk about related work at the end of each presentation does not mean that we consider this work as unimportant. It just turns out that it easier to compare this work with our work *after* we have presented our algorithms.

The second part deals with dominance graphs. Informally, such a graph consists of a collection of tree fragments which have to be assembled into a tree T such that some given constraints are satisfied. These constraints have the form "node u should *dominate* node v", which means that u should be an ancestor of v in T.

The second part has the following structure: In Chapter 6 we briefly discuss a problem from computational linguistics (called scope underspecification) as a motivation for the subsequent work. We proceed with some basic definitions: We define (among other notions) dominance graphs and solved forms. In Chapter 7 we describe a linear time algorithm that can decide whether a given dominance graph is solvable or not. In Chapter 8 we show how the (minimal) solved forms of a dominance graph can enumerated efficiently. Chapter 9 concludes the second part of the thesis with a discussion of related work.

Chapter 2

Prerequisites

In this chapter we introduce some fundamental concepts that are used in the first part of this thesis, which presents propagation algorithms for several constraints.
In the first section of this chapter we introduce constraint programming briefly. We discuss what a constraint satisfaction problem (CSP) is and how it can be solved. The presentation is focused on the properties of propagation algorithms and how these algorithms participate in solving a CSP. The second section gives an overview of basic notions related to graphs and multigraphs. (This section is also relevant for the second part this thesis.) Finally, the third section describes some fundamental concepts from (computational) geometry.

2.1 Constraint Programming

We give a short introduction to the field of constraint programming which is based on a textbook by Apt [Apt03]. Before we formally introduce the basic notions, we give a brief historical overview on the development of the field and name some application areas.

The notion of a constraint was already used by Sutherland in 1963 in his work on an interactive drawing system called SKETCHPAD. The concept of a constraint satisfaction problem was studied by researchers in the artificial intelligence (AI) community in the seventies. From their work the constraint programming paradigm emerged: The basic idea is to combine existing search methods like backtracking or branch-and-bound with constraint propagation techniques (methods to prune the search space). As the application areas for constraint programming grew in the course of time, new types constraints were identified and new propagation algorithms for them were

developed. Often, progress was made by using and combining techniques from different fields like AI, operations research, combinatorial optimization or computer algebra. This made constraint programming an attractive area for researchers outside the AI community.
In the field of constraint programming, practice and theory are closely related. The constraints arising in practical applications often drive the theoretical work, and in turn the applications benefit from the theoretical results.

We enumerate some examples where constraint programming has been successfully used:

- operations research problems (various optimization problems, in particular scheduling)
- molecular biology (DNA sequencing, construction of 3D models of proteins)
- business applications (option trading)
- electrical engineering (location of faults in circuits, computing the circuit layouts, testing and verification of design)
- numerical computation (solving polynomial constraints with guaranteed precision)
- natural language processing (construction of efficient parsers)
- computer algebra (solving and/or simplifying equations of various algebraic structures)

In the sequel we formally define the notion of a constraint satisfaction problem and related concepts. A *variable* X is an object that is associated with a set $Dom(X)$, called the *domain* of X. The domain consists of all values that can be assigned to X. Suppose we have a finite sequence of variables $\mathcal{X} = [X_1, \ldots, X_k]$ (with $k \geq 1$). (The order of the variables is important, and we allow that a variable appears more than once.) A *constraint* C *on* $\mathcal{X}$ is a tuple $(\mathcal{X}, \mathcal{R})$ such that $\mathcal{R} \subseteq Dom(X_1) \times \ldots \times Dom(X_k)$. We call $\mathcal{X}$ the *variable sequence of* C and denote it by $Vars(C)$, and $\mathcal{R}$ is called the *relation of* C and denoted by $Rel(C)$. We say that a tuple $(d_1, \ldots, d_k)$ *satisfies* C if $(d_1, \ldots, d_k) \in Rel(C)$.
Now we are ready to define what a constraint satisfaction problem is:

Definition 2.1 (CSP) *A* constraint satisfaction problem (CSP) $\mathcal{P}$ *is a tuple* $(\mathcal{V}, \mathcal{C})$ *such that* $\mathcal{V} = \{X_1, \ldots, X_n\}$ *is a finite set of variables,* $\mathcal{C}$ *is a finite set of constraints and each constraint* C *in* $\mathcal{C}$ *is a constraint on a sequence*

of variables in $\mathcal{V}$. *We write* $\mathcal{P}$ *as* $\langle X_1 \in Dom(X_1), \ldots, X_n \in Dom(X_n) \,;\, \mathcal{C} \rangle$.
A variable assignment *for* $\mathcal{P}$ *is a mapping* $\alpha : \mathcal{V} \to \bigcup_{i=1}^{n} Dom(X_i)$ *such that* $\alpha(X_i) \in Dom(X_i)$ *for* $i = 1, \ldots, n$. *For a sequence* $\mathcal{X} = [X_{i_1}, \ldots, X_{i_k}]$ *of variables of* $\mathcal{V}$ *we define* $\alpha[\mathcal{X}] := (\alpha(X_{i_1}), \ldots, \alpha(X_{i_k}))$. *We say that* α satisfies *a constraint* C *in* $\mathcal{C}$ *if* $\alpha[Vars(C)] \in Rel(C)$. α *is a* solution *of* $\mathcal{P}$ *if it satisfies all constraints in* $\mathcal{C}$. *We write* α *as* $[X_1 = \alpha(X_1), \ldots, X_n = \alpha(X_n)]$.
$\mathcal{P}$ *is called* consistent *if it has a solution and* inconsistent *otherwise.*
The search space *of* $\mathcal{P}$ *is defined as the set of all variable assignments for* $\mathcal{P}$ *and denoted by* $S(\mathcal{P})$.

In many applications a constraint is not specified as a tuple $(\mathcal{X}, \mathcal{R})$ but rather in a symbolic way. E.g., "$X < Y$" denotes the constraint $([X, Y], \mathcal{R})$ with $\mathcal{R} = \{(x, y) \mid x \in Dom(X) \wedge y \in Dom(Y) \wedge x < y\}$. Thus the interpretation of a symbolic constraint depends on the domains of its variables and on the semantics associated with the constraint.

Example. It is now time to give an example for a constraint satisfaction problem. We consider the famous puzzle

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

The task is to replace each letter by a different digit such that the summation above becomes correct. As we shall see, the puzzle has a unique solution which is depicted below:

$$\begin{array}{cccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

We will now model this puzzle as a CSP. We use the variables S, E, N, D, M, O, R and Y. Since S and M represent leading digits, we choose the integer interval [1..9] as their domains. The domain of each remaining variable is [0..9]. The puzzle can be formulated as follows:

$$\begin{array}{rcccccccc} & & 1000 \cdot S & + & 100 \cdot E & + & 10 \cdot N & + & D \\ & + & 1000 \cdot M & + & 100 \cdot O & + & 10 \cdot R & + & E \\ = 10000 \cdot M & + & 1000 \cdot O & + & 100 \cdot N & + & 10 \cdot E & + & Y \end{array}$$

This condition can be modelled by the linear equality constraint C_{LE} where $Vars\,(C_{\text{LE}}) = [S, E, N, D, M, O, R, Y]$ and $Rel(C_{\text{LE}})$ consists of all tuples $(s, e, n, d, m, o, r, y) \in Dom(S) \times \ldots \times Dom(Y)$ that satisfy

$$9000 \cdot m + 900 \cdot o + 90 \cdot n + y - 1000 \cdot s - 91 \cdot e - 10 \cdot r - d = 0$$

We still have to express the requirement that the 8 digits are pairwise different. We could do this with an inequality constraint of the form "$A \neq B$" for each pair of variables, this would yield $\binom{8}{2} = 28$ inequality constraints in total. But we can also use a single constraint which expresses all these inequalities at once:

$$C_{\mathrm{A}} = \mathit{Alldiff}(S, E, N, D, M, O, R, Y)$$

We have $\mathit{Vars}(C_{\mathrm{A}}) = \mathit{Vars}(C_{\mathrm{LE}})$, and $\mathit{Rel}(C_{\mathrm{A}})$ consists of all the tuples in $\mathit{Dom}(S) \times \ldots \times \mathit{Dom}(Y)$ where the components are pairwise different.
This may look a little bit like cheating and the reader may wonder why we do not express the whole problem as a single constraint, which is of course also possible. The answer is that many constraint programming systems offer the *Alldiff*-constraint to the user and can handle it efficiently. In fact, in Chapter 3 we will discuss this constraint in more detail.

In the sequel we describe a generic procedure[1] which can compute all solutions of a constraint satisfaction problem (see Algorithm 2.1). The algorithm uses two main ingredients: constraint propagation, which prunes parts of the search space that do not contain a solution, and search, which explores the remaining parts.

The procedure `SolveCSP(`$\mathcal{P}$`)` starts by applying constraint propagation to $\mathcal{P}$. This step transforms $\mathcal{P}$ into an equivalent and (hopefully) simpler CSP $\mathcal{P}'$. By *equivalent* we mean that $\mathcal{P}$ and $\mathcal{P}'$ have exactly the same variables and the same solutions, only the domains of the variables and the constraints may be different. We require that the domains in $\mathcal{P}'$ are subsets of the domains in $\mathcal{P}$, i.e. constraint propagation can only shrink the search space.
Let us call a variable *determined* if its domain is a singleton. A CSP is called *ground* if all its variables are determined. For many CSPs it is hard to decide whether they are consistent or not. As long as the function call `PropagateConstraints(`$\mathcal{P}$`)` returns a CSP $\mathcal{P}'$ which is not ground, we only require that $\mathcal{P}'$ and $\mathcal{P}$ are equivalent. But if $\mathcal{P}'$ is ground, then it must be consistent, i.e. the only assignment α for $\mathcal{P}'$ must be a solution. Hence, while the input $\mathcal{P}$ is not ground, the function `PropagateConstraints` may return $\mathcal{P}$ itself. But if $\mathcal{P}$ is ground, constraint propagation must check whether $\mathcal{P}$ is consistent, and if not reduce a variable domain to the empty set.
In many cases, constraint propagation can simplify $\mathcal{P}$ even if it is not ground, we will discuss this in detail later.

Now we continue with the description of Algorithm 2.1. The algorithm distinguishes three cases for $\mathcal{P}'$:

[1] There are approaches which are more generic than ours (see [Apt03]), but this is beyond the scope of this short introduction.

- $\mathcal{P}'$ is ground:
  Then there is only one possible assignment α for $\mathcal{P}'$. Since our assumptions about the function `PropagateConstraints` imply that $\mathcal{P}$ is consistent, we conclude that α is the unique solution of $\mathcal{P}'$. We report α and terminate. As $\mathcal{P}'$ and $\mathcal{P}$ are equivalent, α is also the only solution of $\mathcal{P}$.

- At least one domain of $\mathcal{P}'$ is empty:
  Then there is no possible assignment, i.e. $\mathcal{P}'$ is inconsistent and we terminate. Observe that constraint propagation can reduce a variable domain to the empty set, if the input CSP $\mathcal{P}$ is inconsistent.

- There is no empty domain, and at least one domain contains more than one element:
  Then we call the procedure `SplitOneDomain`. This procedure chooses a variable X with $|Dom(X)| \geq 2$ and splits it into two non-empty disjoint sets D_1 and D_2. Suppose $\mathcal{P}' = \langle X \in Dom(X), \mathcal{D}\,;\, \mathcal{C}\rangle$, where $\mathcal{D}$ describes the domains of the variables different from X. Then the procedure generates the CSP $\mathcal{P}_i = \langle X \in D_i, \mathcal{D}\,;\, \mathcal{C}_i\rangle$ for $i = 1, 2$. $\mathcal{C}_i$ consists of the constraints of $\mathcal{C}$ where the relations are restricted according to the new domain of X. So `SplitOneDomain` also splits the search space: $S(\mathcal{P}') = S(\mathcal{P}_1) \dot{\cup} S(\mathcal{P}_2)$. (In particular, every solution of $\mathcal{P}'$ is a solution of either $\mathcal{P}_1$ or $\mathcal{P}_2$, and vice versa.) Thus this function determines how the search space of $\mathcal{P}'$ is explored. The CSPs $\mathcal{P}_1$ and $\mathcal{P}_2$ are solved recursively.

From the discussion above it should be clear that the algorithm is correct, if the constraint propagation and the domain splitting satisfy the conditions mentioned above. If all domains of $\mathcal{P}$ are finite, then termination is guaranteed.

Algorithm 2.1 Solving a constraint satisfaction problem

Procedure: SolveCSP($\mathcal{P}$)
1: $\mathcal{P}' \leftarrow$ PropagateConstraints($\mathcal{P}$)
2: **case 1:** $\mathcal{P}'$ is ground (i.e. every variable domain is a singleton)
3: report the unique variable assignment for $\mathcal{P}'$ as solution
4: **case 2:** some variable domain of $\mathcal{P}'$ is empty
5: terminate // $\mathcal{P}'$ and $\mathcal{P}$ are inconsistent
6: **otherwise**
7: $(\mathcal{P}_1, \mathcal{P}_2) \leftarrow$ SplitOneDomain($\mathcal{P}'$) // implements the search strategy
8: SolveCSP($\mathcal{P}_1$); SolveCSP($\mathcal{P}_2$)

One can easily imagine some variations of this algorithm. If the domains of the variables are intervals of real numbers, one might stop as soon as the following holds: the size of all intervals is smaller than a certain threshold $\epsilon > 0$ and all assignments for $\mathcal{P}$ are solutions. (Clearly, the latter condition is in general not easy to check, but in some cases it is possible.) If only one solution of $\mathcal{P}$ is needed, one can terminate all recursive calls as soon as the first solution is found.

To put it in a nutshell, in order to solve a CSP we interleave constraint propagation and search (i.e. domain splitting) until we find a solution or we have generated an inconsistent instance. The two main ingredients for `SolveCSP`, namely `PropagateConstraints` and `SplitOneDomain`, will be discussed in the next two sections.

2.1.1 Constraint propagation

This section explains some details about constraint propagation and lays the foundation for the remaining chapters of the first part of this thesis. Recall that we are given a CSP $\mathcal{P}$ and we want to transform it into an equivalent but simpler CSP $\mathcal{P}'$. The term "simpler" means that the variable domains and the relations of the constraints become smaller. An important goal of constraint propagation is to increase the overall performance of a constraint solver: Narrowing domains shrinks the search space that has to be explored by domain splitting. Thus constraint propagation usually saves more time than it costs.

Let us examine a small example that demonstrates what domain *narrowing* (also called *pruning*) is all about: $\mathcal{P} = \langle X \in [3..7], Y \in [2..6] \,;\, X < Y \rangle$. Since Y can be at most 6, we conclude that X is at most 5, hence we can shrink its domain to $[3..5]$. As X is at least 3, Y must be at least 4 and we can narrow the domain of Y to $[4..6]$. Thus we obtain the program $\mathcal{P}' = \langle X \in [3..5], Y \in [4..6] \,;\, X < Y \rangle$. This program cannot be simplified further, because $[X = 3,\ Y = 4]$, $[X = 4,\ Y = 5]$ and $[X = 5,\ Y = 6]$ are solutions of the two CSPs.

Observe that this reduction also works if the CSP contains additional constraints different from "$X < Y$" like in the following example:

$$\mathcal{Q} = \langle X \in [3..7], Y \in [2..6], Z \in [3..6] \,;\, X < Y,\ Y < Z \rangle$$

By the arguments above we can narrow the domains of X and Y as follows:

$$\mathcal{Q}' = \langle X \in [3..5], Y \in [4..6], Z \in [3..6] \,;\, X < Y,\ Y < Z \rangle$$

Now we do an analogous narrowing step for "$Y < Z$":

$$\mathcal{Q}'' = \langle X \in [3..5], Y \in [4..5], Z \in [5..6] \,;\, X < Y,\ Y < Z \rangle$$

This enables us to prune the domain of X once more:

$$\mathcal{Q}''' = \langle X \in [3..4], Y \in [4..5], Z \in [5..6] \,;\, X < Y,\ Y < Z \rangle$$

After that no further simplification is possible. We invite the reader to check that three steps would also have been necessary if we had started the propagation with the constraint "$Y < Z$".

From the examples above we can learn two important facts about constraint propagation:

- Each constraint can be propagated independently from all other constraints.
- In order to achieve the maximum amount of pruning, it may be necessary to propagate the same constraint more than once.

Therefore many constraint solvers (like e.g. Oz [SS03]) have the following architecture (see Figure 2.1): There is a domain store, which contains the variables of the CSP and their respective domains. For every constraint[2] C of the CSP there is a propagator attached to the store. Whenever the domain of a variable in $\mathit{Vars}(C)$ has changed (because a propagator has narrowed it or due to `SplitOneDomain`), the constraint solver invokes the propagator corresponding to C. The propagator reads the current domains of $\mathit{Vars}(C)$ from the store and runs an appropriate propagation algorithm which tries to prune the domains. When the propagation algorithm returns, the narrowed variable domains are written to the domain store.

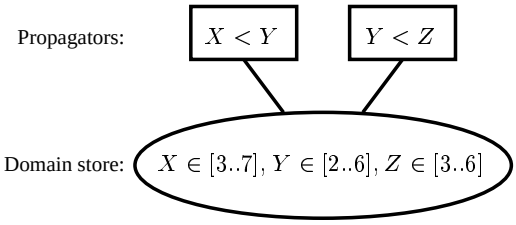


Figure 2.1: The domain store and the attached propagators for $\mathcal{Q}$.

The constraint solver invokes the propagators for the different constraints until one variable domain becomes empty or the domain store becomes stable, which means that no propagator is able to reduce a domain anymore. In some cases the propagator for a constraint C can discover that the domain store *entails* C, which means that any possible assignment satisfies C. Consider

[2] In the Oz model the terminology is slightly different: a domain requirement of the form $X \in \mathit{Dom}(X)$ is called a *basic constraint*, and what we call a "constraint" is called a *complex constraint*, because it requires a propagator.

e.g. $\mathcal{R} = \langle X \in [3..5], Y \in [8..9] \,;\, X < Y \rangle$. Any assignment for $\mathcal{R}$ satisfies "$X < Y$", thus this constraint can be removed.

We want to point out that the relations of the constraints are usually not stored explicitly because the constraints are given to the constraint solver in a symbolic (i.e. implicit) way. The user would write for instance "$X < Y$" instead of entering every tuple in the corresponding relation. Thus the propagator "knows" the semantics of its constraint and it can call a propagation algorithm which enforces this particular semantics. We will clarify this in the next section.

Propagation algorithm

We will now discuss the concept of a propagation algorithm. The propagation algorithms that will be considered in this thesis do not apply to an arbitrary constraint, but rather to a class of constraints like the class of *Alldiff*-constraints.[3] The actual constraint is determined by the variables (with their respective domains) and the semantics of the constraint class. E.g., the relation of the constraint $A := \mathit{Alldiff}(X_1, X_2, X_3)$ is uniquely determined by the variable domains and the semantics of *Alldiff*:

$$\mathit{Rel}(A) = \{(d_1, d_2, d_3) \mid d_1 \in \mathit{Dom}(X_1) \wedge d_2 \in \mathit{Dom}(X_2) \wedge d_3 \in \mathit{Dom}(X_3) \\ \wedge\, d_1 \neq d_2 \neq d_3 \neq d_1\}$$

A formal definition of the notion "constraint class" follows:

Definition 2.2 (constraint class) *A set $\mathcal{K}$ of constraints is called a* constraint class *if it satisfies the following properties:*

- *For every sequence $D_1, \ldots, D_k$ of domains there is exactly one constraint $C \in \mathcal{K}$ of arity k, i.e. $\mathit{Vars}(C) = [X_1, \ldots, X_k]$, such that $\mathit{Dom}(X_i) = D_i$ for $i = 1, \ldots, k$.*
- *Let $D_1, \ldots, D_k$ and $D'_1, \ldots, D'_k$ be two sequences of domains and let C and C' denote the corresponding constraints in $\mathcal{K}$. If $D_i \supseteq D'_i$ for $i \in [1..k]$, then $\mathit{Rel}(C') = \mathit{Rel}(C) \cap (D'_1 \times \ldots \times D'_k)$.*

The input of a propagation algorithm that applies to a particular class $\mathcal{K}$ of constraints does not contain the constraint itself. The input consists only of the variable domains $D_1, \ldots, D_k$. This determines a constraint C with $\mathit{Rel}(C) \subseteq D_1 \times \ldots \times D_k$ in $\mathcal{K}$. In the following definition we define the requirements for a propagation algorithm:

[3] However, it is common practice to talk about a "propagation algorithm for the *Alldiff*-constraint" when one actually refers to an algorithm for the *Alldiff*-class. We adopt this practice later, too.

Definition 2.3 (propagation algorithm) *A* propagation algorithm $\mathcal{A}$ *for a class* $\mathcal{K}$ *of constraints takes as input a sequence* $D_1, \ldots, D_k$ *of domains and computes a sequence* $D'_1, \ldots, D'_k$ *of domains and a status value. Let* C *be the constraint in* $\mathcal{K}$ *determined by* $D_1, \ldots, D_k$ *(that is* $Vars(C) = [X_1, \ldots, X_k]$ *and* $Dom(X_i) = D_i$ *for* $i = 1, \ldots, k$*).*
We require that the following holds:

1. $Rel(C) \subseteq D'_1 \times \ldots \times D'_k$ *(i.e. no solution of* C *is pruned).*
2. $D'_i \subseteq D_i$ *for* $i = 1, \ldots, k$ *(i.e. domains can only shrink).*
3. *status* $\in \{failure, entailment, success\}$ *and if status equals*
   - *failure, then* $D'_i = \emptyset$ *for some* i.
   - *entailment, then* $D'_1 \times \ldots \times D'_k \subseteq Rel(C)$*, and* $D'_i \neq \emptyset$ *for all* i.
   - *success, then* $|D'_i| \geq 2$ *for some* i*, and* $D'_j \neq \emptyset$ *for all* j.

We call $\mathcal{A}$ idempotent *if applying* $\mathcal{A}$ *to its output domains always yields the same output domains again.*

We want to make some comments. Assume that C is a constraint in a CSP $\mathcal{P}$ and that $\mathcal{A}$ is a propagation algorithm that applies to the class of C. Let $Vars(C) = [X_1, \ldots, X_k]$ and suppose that $\mathcal{P}$ has the form

$$\mathcal{P} = \langle X_1 \in D_1, \ldots, X_k \in D_k, \mathcal{D}\,;\, \{C\} \cup \mathcal{C} \rangle$$

where $\mathcal{D}$ describes the domains of the variables in $\mathcal{P}$ that are not in $Vars(C)$ and $\mathcal{C}$ is a set of constraints. Let $D'_1, \ldots, D'_k$ be the domains obtained by invoking $\mathcal{A}$ on $D_1, \ldots, D_k$. Property 1 from the definition above ensures that $\mathcal{P}$ is equivalent to the CSP $\mathcal{P}'$ with the narrowed domains:

$$\mathcal{P}' = \langle X_1 \in D'_1, \ldots, X_k \in D'_k, \mathcal{D}\,;\, \{C\} \cup \mathcal{C}' \rangle$$

where $\mathcal{C}'$ is obtained by restricting the relations of the constraints in $\mathcal{C}$ to the narrowed domains. Hence, the pruning done for C does not affect $Rel(C)$ but possibly the relations of other constraints in $\mathcal{P}$.
If $\mathcal{A}$ returns *failure*, then $Rel(C) = \emptyset$ (see Property 1) and $\mathcal{P}$ is inconsistent. If *entailment* is returned, then we have $Rel(C) = D'_1 \times \ldots \times D'_k$. In that case $\mathcal{P}$ is equivalent to

$$\mathcal{P}'' = \langle X_1 \in D'_1, \ldots, X_k \in D'_k, \mathcal{D}\,;\, \mathcal{C}' \rangle$$

i.e. C can be removed.

Observe that our requirements for a propagation algorithm are not very demanding. As long as there is an undetermined variable, the algorithm may return the input domains and report *success*. Only if all domains are singletons, the algorithm has to check whether the corresponding tuple is a solution of C and return *failure* or *entailment*.

The function `PropagateConstraints`

We consider now a CSP $\mathcal{P} = \langle X_1 \in D_1, \ldots, X_n \in D_n \,;\, \mathcal{C} \rangle$, where the domains $D_1, \ldots, D_n$ are finite. We discuss a possible implementation of the function `PropagateConstraints(`$\mathcal{P}$`)` from above. It uses propagation algorithms to propagate all the constraints of $\mathcal{P}$. Our algorithm is based on an algorithm called `Arc` by Apt [Apt03, page 273]. We assume that for each constraint C of $\mathcal{P}$ we have a propagation algorithm $\texttt{propagate}_C$ for the class of C. Our algorithm computes a sequence $D'_1, \ldots, D'_n$ of domains such that $\mathcal{P}' = \langle X_1 \in D'_1, \ldots, X_n \in D'_n \,;\, \mathcal{C}' \rangle$ is equivalent to $\mathcal{P}$. Every constraint $C' \in \mathcal{C}'$ corresponds to a constraint $C \in \mathcal{C}$ for which entailment has not been detected. We have $\mathit{Vars}(C') = \mathit{Vars}(C) = [X_{i_1}, \ldots, X_{i_k}]$ and $\mathit{Rel}(C') = \mathit{Rel}(C) \cap (D'_{i_1} \times \ldots \times D'_{i_k})$.

Suppose that all propagation algorithms are deterministic. Thus each of them can be interpreted (mathematically) as a function that maps a tuple of domains to a tuple of domains (if we ignore the status value for the moment). Unless $\mathcal{P}$ is inconsistent, the algorithm finds a common *fixpoint* of these functions, which is defined as follows: Let C be a constraint with $\mathit{Vars}(C) = [X_{i_1}, \ldots, X_{i_k}]$. We say that $D'_1 \times \ldots \times D'_n$ is a *fixpoint* of $\texttt{propagate}_C$ if $\texttt{propagate}_C$ maps $(D'_{i_1}, \ldots, D'_{i_k})$ to $(D'_{i_1}, \ldots, D'_{i_k})$ itself.

Algorithm 2.2 is straightforward: It applies the propagation algorithms exhaustively until *failure* is reported or a fixpoint is reached. The algorithm maintains two sets of constraints S_0 and S. S_0 consists of all constraints of $\mathcal{P}$ for which entailment has not been detected. An invariant of the algorithm will be that the set S contains (at least) all the constraints for which the current variable domains are not a fixpoint. At the beginning both S_0 and S contain all constraints of $\mathcal{P}$ and the invariant holds.

Now we describe the main loop of the algorithm. As long as S is not empty, the algorithm chooses a constraint C in S. Let $X_{i_1}, \ldots, X_{i_k}$ be the variables of C with their respective current domains $E_1, \ldots, E_k$. The propagation algorithm $\texttt{propagate}_C$ is applied to $E_1, \ldots, E_k$; it returns narrowed domains $E'_1, \ldots, E'_k$ and a *status* value. For $j = 1, \ldots, k$ we replace the current domain $\mathit{Dom}(X_{i_j})$ by $\mathit{Dom}(X_{i_j}) \cap E'_j$. (We do not replace the domain simply by E'_j because X_{i_j} might occur more than once in $\mathit{Vars}(C)$.)
Now we distinguish three cases: If *failure* has occurred (i.e. one variable do-

main has become empty), we terminate and return a CSP where at least one variable domain is empty, namely a domain of a variable in $\mathit{Vars}(C)$. If the returned status is *entailment*, we remove C from both S_0 and S. (Observe that the variable domains will always be a fixpoint of $\texttt{propagate}_C$ from now on, because the domains can only shrink.) If the propagation algorithm returns *success* and the output domains are the same as the input domains, then these domains are a fixpoint of C and we remove C from S. (Note that we do not require that $\texttt{propagate}_C$ is idempotent; otherwise we could remove it, even if the domains have changed.)
If no failure has occurred, we update S: We insert every constraint $C' \in S_0$ such that $\mathit{Vars}(C')$ contains a variable X_{i_j} whose domain has changed during this iteration, for the new domains may not be a fixpoint of these constraints anymore. This establishes the invariant again: All constraints of $\mathcal{P}$ for which the current domains are not a fixpoint are contained in S.

Algorithm 2.2 Constraint propagation

Function: PropagateConstraints($\mathcal{P}$)
1: $S_0 \leftarrow \{C \mid C \text{ is a constraint of } \mathcal{P}\}$
2: $S \leftarrow S_0$
3: **while** $S \neq \emptyset$ **do**
4: choose $C \in S$; suppose $\mathit{Vars}(C) = [X_{i_1}, \ldots, X_{i_k}]$
5: $E_j \leftarrow \mathit{Dom}(X_{i_j})$ **for** $j = 1, \ldots, k$
6: $(E'_1, \ldots, E'_k, \mathit{status}) \leftarrow \texttt{propagate}_C(E_1, \ldots, E_k)$
7: $\mathit{Dom}(X_{i_j}) \leftarrow \mathit{Dom}(X_{i_j}) \cap E'_j$ **for** $j = 1, \ldots, k$
8: **case 1:** $\mathit{Dom}(X_{i_j}) = \emptyset$ for some $j \in [1..k]$ // *failure*
9: break from while-loop
10: **case 2:** $\mathit{status} = \mathit{entailment}$
11: remove C from S_0 and from S
12: **case 3:** $\mathit{status} = \mathit{success}$
13: **if** $E'_1 = E_1 \wedge \ldots \wedge E'_k = E_k$ **then** remove C from S
14: **for all** j in $[1..k]$ s.th. $E'_j \neq E_j$ **do**
15: $S \leftarrow S \cup \{C' \in S_0 \mid X_{i_j} \in \mathit{Vars}(C')\}$
16: **end while**
17: return the CSP $\mathcal{P}'$ containing the current variable domains and the constraints in S_0 (restricted to those domains)

When the main loop of the algorithm terminates because S has become empty, the invariant implies that a fixpoint has been found. We return the CSP $\mathcal{P}'$ that reflects the current variable domains. The constraints of $\mathcal{P}'$ are the constraints in S_0 (restricted to the current domains).

Theorem 2.1 *Let $\mathcal{P}$ be a finite domain CSP. If we apply Algorithm 2.2 to $\mathcal{P}$, then the algorithm terminates and returns a CSP $\mathcal{P}'$ equivalent to $\mathcal{P}$. If $\mathcal{P}'$ is ground, then $\mathcal{P}'$ and hence $\mathcal{P}$ are consistent. Moreover, if all propagation algorithms are deterministic and no failure occurs, then the domains in $\mathcal{P}'$ are a common fixpoint of all propagation algorithms.*

Proof. We show that the algorithm terminates. Let d be the sum of the cardinalities of all domains in $\mathcal{P}$, and let c denote the number of constraints in $\mathcal{P}$. Clearly, we always have $|S| \leq |S_0| \leq c$. Let us assume that no failure occurs, otherwise the algorithm terminates and there is nothing to show. Then in each iteration of the main loop the cardinality of S or the cardinality of a variable domain decreases. Between two reductions of a variable domain there can be at most c iterations (otherwise S would become empty), and hence there can be at most $c \cdot d$ iterations in total.

We come to the equivalence of $\mathcal{P}$ and $\mathcal{P}'$. An easy induction shows that the following holds, whenever line 3 is executed: The CSP induced by the current variable domains and the constraints in S_0 is equivalent to $\mathcal{P}$.

Suppose now that $\mathcal{P}'$ is ground. Let $X_1, \ldots, X_n$ be the variables of $\mathcal{P}'$ and let $D'_1 = \{d_1\}, \ldots, D'_n = \{d_n\}$ be their respective domains. Consider a constraint C' of $\mathcal{P}'$. It corresponds to a constraint C which belonged to S_0 upon termination of the algorithm. Assume $\mathit{Vars}(C) = [X_{i_1}, \ldots, X_{i_k}]$. We show that there has been a call to $\texttt{propagate}_C(\{d_{i_1}\}, \ldots, \{d_{i_k}\})$. If the initial domains of $X_{i_1}, \ldots, X_{i_k}$ were already singletons, this claim is obvious, because C belongs to the initial set S. Otherwise, C is inserted into S after the last domain of a variable in $\mathit{Vars}(C)$ has become a singleton. Since $\texttt{propagate}_C$ has not reported failure when applied to the singleton domains, we conclude that $(d_{i_1}, \ldots, d_{i_k})$ satisfies C and hence C'. Thus the variable assignment $[X_1 = d_1, \ldots, X_n = d_n]$ is a solution of $\mathcal{P}'$ and of $\mathcal{P}$.

The last statement about the fixpoint follows immediately from the discussion above. □

We want to make a comment that relates our presentation to the work of Apt [Apt03, Chapter 7]. Apt requires that the propagation algorithms satisfy a property called *monotonicity*. This means the following. Suppose we have a propagation algorithm which takes as input k domains. Assume that applied to $E_1, \ldots, E_k$ it returns $E'_1, \ldots, E'_k$. Suppose we choose domains $F_1, \ldots, F_k$ such that $F_1 \subseteq E_1, \ldots, F_k \subseteq E_k$, run the algorithm on these domains and obtain $F'_1, \ldots, F'_k$. If the algorithm is *monotonic*, then we have $F'_1 \subseteq E'_1, \ldots, F'_k \subseteq E'_k$.
Apt shows that the fixpoint $D'_1 \times \ldots \times D'_n$ computed by the algorithm above does not depend on how the constraint $C \in S$ is chosen in line 4, if all invoked propagation algorithms are monotonic (see Lemma 7.5 in [Apt03]).

It will turn out that all propagation algorithms presented in this thesis are monotonic.

Global constraints and local consistency

In the approach that we have described above, the propagators for the constraints are independent. They communicate with each other through the domains of the variables, which are held in the domain store. This approach makes it easy to integrate new propagation algorithms into a constraint solver because the old propagation algorithms do not have to be changed. But the fact that a propagator for a constraint C is not aware of the constraints different from C also has some drawbacks. Consider e.g. the CSP $\mathcal{P} = \langle X \in \{1,2\}, Y \in \{1,2\}, Z \in \{1,2\} \,;\, X \neq Y, Y \neq Z, Z \neq X \rangle$, which is clearly inconsistent. But since the relation of each of the inequality constraints is $\{(1,2),(2,1)\}$, a propagation algorithm that only takes into account one of these constraints cannot reduce any variable domain.

In order to overcome this limitation, so-called *global constraints* were introduced. There is no formal definition for this notion, but a global constraint is usually a "high-level" constraint that combines a set of "low-level" constraints (that are somehow related to each other) into a single constraint. In our example above we could replace the three constraints by a single *Alldiff*-constraint and obtain an equivalent CSP $\mathcal{P}' = \langle X \in \{1,2\}, Y \in \{1,2\}, Z \in \{1,2\} \,;\, \mathit{Alldiff}(X,Y,Z) \rangle$.

Using global constraints has the following advantages: A propagation algorithm for the global constraint can usually do more pruning than the algorithms for the low-level constraints together. (In our example, the propagation algorithm for *Alldiff* could recognize that there are 3 variables but only 2 values. Hence, it could immediately report failure.) Moreover, it often requires less space to store the global constraint. (An *Alldiff*-constraint with n variables replaces $\binom{n}{2}$ inequality constraints.) Finally, it is easier and less error-prone for the user of the constraint solver to state a single global constraint than to enter all the corresponding low-level constraints.

When we talk about a propagation algorithm we often want to characterize the amount of pruning that it achieves. (Recall that a propagation algorithm is not required to do any pruning unless all the domains are singletons.) In order to do so we discuss some types of "local consistency". Local consistency means that some "parts" of a CSP have a desired form. In this presentation we only describe local consistency notions which are relevant for this thesis, these are notions which apply to a single constraint of a CSP.

We begin with a discussion of arc-consistency. Informally, a constraint C is arc-consistent[4] if every value in the domain of each variable in $Vars(C)$ participates in at least one solution of C.

Definition 2.4 (arc-consistency) *Let C be a constraint on the variable sequence $[X_1, \ldots, X_k]$ with respective domains $D_1, \ldots, D_k$. We say that C is* arc-consistent *if $Rel(C) \neq \emptyset$ and the following holds: For all $i \in [1..k]$ and every value $d \in D_i$ there is a tuple $(d_1, \ldots, d_k) \in Rel(C)$ with $d = d_i$.*

Let $\mathcal{A}$ be a propagation algorithm for (the class of) C. Let $D'_1, \ldots, D'_k$ be the domains that are obtained by running $\mathcal{A}$ on $D_1, \ldots, D_k$. We say that $\mathcal{A}$ achieves arc-consistency for C if

- *either $Rel(C) \neq \emptyset$ and the constraint C' with $Rel(C') = Rel(C)$ and the domains $D'_1, \ldots, D'_k$ is arc-consistent*
- *or $Rel(C) = \emptyset$ and $\mathcal{A}$ returns failure.*

We call $\mathcal{A}$ an arc-consistency algorithm *if it achieves arc-consistency on all admissible inputs.*

Since we require that $Rel(C) \subseteq D'_1 \times \ldots \times D'_k$ (cf. Property 1 in Definition 2.3), there can be no propagation algorithm that does more pruning. Thus arc-consistency characterizes the maximum amount of pruning that can be achieved for a single constraint. However, a CSP does not have to be consistent even if all its constraints are arc-consistent, as the example $\mathcal{P}$ with the three inequalities shows.

Sometimes we consider only domains that are (closed) intervals contained in a linearly ordered set $(U, \leq)$. For two elements a and b in U we define the *interval* $[a; b]$ as $\{u \in U \mid a \leq u \leq b\}$. We call a the *lower* and b the *upper endpoint* of the interval $I = [a; b]$, and we use $\underline{I}$ to denote a and $\overline{I}$ to denote b. One reason for using intervals is that they can be stored more efficiently than arbitrary sets, in particular if $U = \mathbb{R}$. The local consistency notion that is usually used with respect to interval domains is bound-consistency.
Informally, a constraint C over interval domains is called bound-consistent if every endpoint of the domain of each variable in $Vars(C)$ participates in a solution of C. A formal definition follows:

Definition 2.5 (bound-consistency) *Let C be a constraint on the variable sequence $[X_1, \ldots, X_k]$ with respective domains $[l_1; h_1], \ldots, [l_k; h_k]$. We say that C is* bound-consistent *if $Rel(C) \neq \emptyset$ and the following holds: For*

[4]Some people use the term "arc-consistency" only for binary constraints, they call our notion of arc-consistency "generalized arc-consistency" or "hyper-arc-consistency".

all $i \in [1..k]$ *there is a tuple* $(d_1, \ldots, d_k) \in Rel(C)$ *with* $l_i = d_i$ *and a tuple* $(d'_1, \ldots, d'_k) \in Rel(C)$ *with* $h_i = d'_i$.

The definition of a *bound-consistency algorithm* is analogous to the definition of an *arc-consistency algorithm* (cf. Definition 2.4). Under the restriction that all domains must be intervals, bound-consistency is the strongest possible local consistency for a single constraint. (One cannot narrow the intervals any further without pruning solutions of the constraint.)

We will show now that an arc-consistency or bound-consistency algorithm is always monotonic and idempotent. To do so we study an equivalent formulation for arc- and for bound-consistency. For a set $S \subseteq S_1 \times \ldots \times S_k$ and $i \in [1..k]$ we define the *projection* π_i of S onto its i-th component as follows:

$$\pi_i(S) := \{s \in S_i \mid \exists (s_1, \ldots, s_k) \in S \text{ with } s = s_i\}$$

Let C be a constraint with $Vars(C) = [X_1, \ldots, X_k]$ and $Rel(C) \neq \emptyset$. Then C is arc-consistent iff $Dom(X_i) = \pi_i(Rel(C))$ for $i = 1, \ldots, k$. Thus an arc-consistency algorithm computes the projection of $Rel(C)$ onto all its components.
C is bound-consistent iff $l_i = \min \pi_i(Rel(C))$ and $h_i = \max \pi_i(Rel(C))$ exist and $Dom(X_i) = [l_i; h_i]$ for $i = 1, \ldots, k$. Hence, a bound-consistency algorithm computes the minimum and the maximum element in all the projections of $Rel(C)$.

Therefore, the output of an arc-consistency or bound-consistency algorithm $\mathcal{A}$ is uniquely determined by the relation of the constraint. Since $\mathcal{A}$ is not allowed to change this relation (see Properties 1 and 2 of Definition 2.3), it is idempotent. Monotonicity follows immediately from the fact that the projections are monotonic. This proves the following lemma:

Lemma 2.1 *Every arc-consistency and every bound-consistency algorithm is monotonic and idempotent.*

We conclude this section with the discussion of range-consistency. To the best of our knowledge this local consistency notion has only been used for the *Alldiff*-constraint. We introduce it here, because we need it in Section 3.2.3.

Definition 2.6 (range-consistency) *Let* C *be a constraint in a class* $\mathcal{K}$ *of constraints with* $Vars(C) = [X_1, \ldots, X_k]$ *and respective domains* $D_1, \ldots, D_k$. *Assume that* D_i *is contained in some linearly ordered set* U_i *and that* $l_i = \min D_i$ *and* $h_i = \max D_i$ *exist for* $i = 1, \ldots, k$. *Let* C' *denote the constraint in* $\mathcal{K}$ *determined by the domains* $[l_1; h_1], \ldots, [l_k; h_k]$.
We say that C *is* range-consistent *if* $Rel(C') \neq \emptyset$ *and the following holds: For all* $i \in [1..k]$ *and every value* $d \in D_i$ *there is a tuple* $(d_1, \ldots, d_k) \in Rel(C')$ *with* $d = d_i$.

Let us look at the example $A := \mathit{Alldiff}(X_1, X_2, X_3)$ with $\mathit{Dom}(X_1) = \mathit{Dom}(X_2) = \mathit{Dom}(X_3) = \{1, 3\} \subseteq \mathbb{Z}$. In order to check whether A is range-consistent we consider the constraint $A' := \mathit{Alldiff}(X_1', X_2', X_3')$ where all domains are the interval $[1..3]$. For $i = 1, 2, 3$ we have $\pi_i(\mathit{Rel}(A')) = [1..3] \supseteq \mathit{Dom}(X_i)$. Hence, A is range-consistent, but it is not arc-consistent, because $\mathit{Rel}(A) = \emptyset$. So range-consistency is strictly weaker than arc-consistency.

We discuss an example with integer interval domains: $\mathit{Alldiff}(X, Y, Z)$ with $\mathit{Dom}(X) = [1..4]$, $\mathit{Dom}(Y) = \mathit{Dom}(Z) = [2..3]$. The constraint is bound-consistent, but a range-consistency algorithm narrows the domain of X to $\{1, 4\}$.

Example. $\mathit{SEND} + \mathit{MORE} = \mathit{MONEY}$
To conclude this section on constraint propagation, we pick up the puzzle from the beginning (page 5). Recall that we modelled it by the following two constraints:

- a linear equality constraint:
  $9000 \cdot M + 900 \cdot O + 90 \cdot N + Y - 1000 \cdot S - 91 \cdot E - 10 \cdot R - D = 0$
- an *Alldiff*-constraint: $\mathit{Alldiff}(S, E, N, D, M, O, R, Y)$

For each of the two constraint classes we will discuss a simple propagation algorithm (which does not achieve any of the local consistency notions mentioned above). After that we will apply these algorithms to the puzzle. We begin with an algorithm for *Alldiff*, because it is the simpler one. If some variable domain is a singleton $\{d\}$, the algorithm iterates over all other domains and removes the value d from those domains. If one of the domains becomes empty, it returns *failure*. If all output domains are singletons, it reports *entailment*, otherwise it returns *success*.

Now we sketch a propagation algorithm for linear equality constraints of the form $\sum_{i=1}^{k} a_i X_i = b$ with $a_i \neq 0$ for $i = 1, \ldots, k$. We assume that the input domains are integer intervals, i.e. $D_i = [l_i..h_i]$ for $i = 1, \ldots, k$. For $i \in [1..k]$ we compute the output domain D_i' as follows: We rewrite the constraint as $X_i = \frac{b}{a_i} + \sum_{j \in [1..k] \setminus \{i\}} -\frac{a_j}{a_i} \cdot X_j$. We partition the indices in the summation above into two sets depending on the sign of $\alpha_j := -\frac{a_j}{a_i}$:

$J^+ := \{j \in [1..k] \setminus \{i\} \mid \alpha_j > 0\}$ and $J^- := \{j \in [1..k] \setminus \{i\} \mid \alpha_j < 0\}$
For $j \in J^+$, we have $\alpha_j l_j \leq \alpha_j X_j \leq \alpha_j h_j$; and for $j \in J^-$, we have $\alpha_j h_j \leq \alpha_j X_j \leq \alpha_j l_j$. Using the fact that X_i is an integer, we obtain

$$\underbrace{\left\lceil \frac{b}{a_i} + \sum_{j \in J^+} \alpha_j l_j + \sum_{j \in J^-} \alpha_j h_j \right\rceil}_{=:\, l_i'} \leq X_i \leq \underbrace{\left\lfloor \frac{b}{a_i} + \sum_{j \in J^+} \alpha_j h_j + \sum_{j \in J^-} \alpha_j l_j \right\rfloor}_{=:\, h_i'}$$

Thus the narrowed domain D_i' becomes $D_i \cap [l_i'..h_i']$. As above we report *failure*, if one domain becomes empty, and *entailment*, if all output domains are singletons. In all other cases we return *success*.

Recall that the initial domains of the leading digits S and M are $[1..9]$ and the other domains are $[0..9]$. Applying the propagation algorithm for linear equality constraints narrows the domains to

$$S = 9, E \in [0..9], N \in [0..9], D \in [0..9],$$

$$M = 1, O \in [0..1], R \in [0..9], Y \in [0..9]$$

After three iterations of the *Alldiff* algorithm the domains look as follows:

$$S = 9, E \in [2..8], N \in [2..8], D \in [2..8], M = 1, O = 0, R \in [2..8], Y \in [2..8]$$

Five successive applications of the algorithm for the linear equality reduce the domain of E to $[4..7]$ and the domain of N to $[5..8]$. After that the domain store becomes stable and the constraint propagation ends.

2.1.2 Search

Recall that the search space $S(\mathcal{P})$ of a CSP $\mathcal{P}$ is the set of all possible variable assignments for $\mathcal{P}$. Note that we do not require that the variable assignments in $S(\mathcal{P})$ are solutions of $\mathcal{P}$. As the example above shows, constraint propagation often yields a CSP $\mathcal{P}'$ with a considerably smaller search space, but in general it is not able to produce a ground CSP (i.e. a solution) or to prove inconsistency. In that case the search space of $\mathcal{P}'$ has to be explored.

The function `SplitOneDomain` (used in Algorithm 2.1) controls how this exploration is carried out. Recall that this function chooses a variable X in $\mathcal{P}'$ with $|Dom(X)| \geq 2$ and partitions $Dom(X)$ it into two non-empty disjoint domains D_1 and D_2. These domains give rise to two CSPs $\mathcal{P}_1$ and $\mathcal{P}_2$ (see page 7) which are solved recursively.

There are several strategies how to select the variable for splitting and how to split its domain. The search strategy often has a great influence on the overall performance of a constraint solver, but a deep discussion of this topic is beyond the scope of this chapter (more information can be found for instance in [Apt03, Chapter 8]). A selection strategy that often yields good results in practice is the so-called *first fail* strategy, which always chooses an undetermined variable with a domain of minimum cardinality. A common technique for splitting a domain D is to chose an element $d \in D$ (for instance $d = \min D$) and to set $D_1 = \{d\}$ and $D_2 = D \setminus d$. If D is an integer interval

$[l..h]$, one often uses bisection, i.e. $D_1 = [l..m]$ and $D_2 = [m..h]$, where $m = \lfloor \frac{l+m}{2} \rfloor$.

Example. $SEND + MORE = MONEY$
We show how constraint propagation and search solve the puzzle. We use a fairly naive search strategy: We select the first undetermined variable in the sequence $[S, E, N, D, M, O, R, Y]$; and we split its domain D into $D_1 = \{\min D\}$ and $D_2 = D \setminus D_1$. Recall that applying constraint propagation to the original CSP $\mathcal{P}$ produces the following domains:

$$S = 9, E \in [4..7], N \in [5..8], D \in [2..8], M = 1, O = 0, R \in [2..8], Y \in [2..8]$$

Eliminating the determined variables S, M and O from the linear equality constraint yields

$$90 \cdot N + Y - 91 \cdot E - 10 \cdot R - D = 0$$

According to our search strategy we split the domain of E into $\{4\}$ and $[5..7]$. Let us consider first the CSP $\mathcal{P}_1$ induced by $E = 4$. Applying the propagation algorithm for linear equality exhaustively, we obtain $N = 5$, $R = 8$, $D = 8$ and $Y = 2$, which causes the *Alldiff*-propagator to report failure.

So we come to the second CSP $\mathcal{P}_2$, which is induced by $E \in [5..7]$. Constraint propagation reduces the domain of N to $[6..8]$. Hence, we split the domain of E once more, namely into $\{5\}$ and $[6..7]$. Propagation for the program $\mathcal{P}_{21}$ induced by $E = 5$ yields the following solution

$$S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2$$

The program $\mathcal{P}_{22}$ induced by the domain $[6..7]$ for E has no solutions, which is detected after one more split.

Observe how much constraint propagation contributes to solving this example. The search space of the original CSP $\mathcal{P}$ has $9^2 \cdot 10^6$ elements. By using constraint propagation we could prune it considerably. Thus we were able to find the solution of $\mathcal{P}$ and prove its uniqueness after only three domain splittings.

2.2 Multigraphs and graphs

In this section we introduce some basic notions and results from graph theory which are needed in the sequel of the thesis. A *multigraph* is a fundamental

concept which can model binary relationships between objects. Each object corresponds to a *node* in the multigraph and the relationships are modelled by *edges*, which can be directed or undirected. The formal definition is as follows:

Definition 2.7 (multigraph) *An undirected multigraph $\mathcal{M}$ is defined as a triple (V, E, inc) such that V and E are two finite sets and inc is a mapping from E to $\binom{V}{2}$.*[5] *The elements of V are called the* nodes *of $\mathcal{M}$, and the elements of E are the* edges *of $\mathcal{M}$. We call inc the* incidence mapping *of $\mathcal{M}$. Let $e \in E$ be an edge with $inc(e) = \{u, v\}$. Then we say that u and v are* incident *to e, and u and v are* adjacent *via e. The* degree of v *is the number of edges incident to v. The set of all nodes that are adjacent to v is called the* neighbours *of v.*
A directed multigraph $\vec{\mathcal{M}}$ is a triple (V, E, inc) such that V and E are two finite sets and inc is a mapping from E to $V \times V$. As before, V and E are the nodes *and the* edges *of $\vec{\mathcal{M}}$ and inc is called the* incidence mapping *of $\vec{\mathcal{M}}$. Let $e \in E$ be an edge with $inc(e) = (u, v)$. Then u is called the* source node *and v is the* target node *of e. We say that e is* directed *from u to v, and we call e an* outgoing edge of u *and an* incoming edge of v. *Both nodes are said to be* incident *to e and they are* adjacent *via e.*

We continue with some more definitions. A (directed or undirected) multigraph $\mathcal{M}' = (V', E', inc')$ is called a *subgraph* of a multigraph $\mathcal{M} = (V, E, inc)$ if $V' \subseteq V$, $E' \subseteq E$ and $inc'(e) = inc(e)$ for all $e \in E'$. A set of nodes $\tilde{V} \subseteq V$ *induces* the subgraph $\tilde{\mathcal{M}} = (\tilde{V}, \tilde{E}, \tilde{inc})$ where $\tilde{E}$ consists of all edges incident to two nodes in $\tilde{V}$ and $\tilde{inc} = inc|_{\tilde{E}}$. A set of edges $\hat{E} \subseteq E$ induces a subgraph $\hat{\mathcal{M}} = (\hat{V}, \hat{E}, \hat{inc})$, where $\hat{V}$ is the set of all nodes incident to an edge in $\hat{E}$ and $\hat{inc} = inc|_{\hat{E}}$.

When we visualize a multigraph like in Figure 2.2, we draw a node as a circle or a box and an edge as a line connecting its incident nodes. We introduce an important notion to talk about multigraphs: *paths*. A path p of length k (with $k \geq 0$) from a node v_0 to a node v_k is a sequence $[v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k]$ such that $v_0, \ldots, v_k$ are nodes and $e_1, \ldots, e_k$ are edges of the multigraph, and the following holds: If the multigraph is undirected, we require $inc(e_i) = \{v_{i-1}, v_i\}$ for $i = 1, \ldots, k$. And if it is directed, then $inc(e_i) = (v_{i-1}, v_i)$ must hold for all i. (Observe that we allow paths of length 0, these are called *empty*.) We say that p *visits* the nodes $v_0, \ldots, v_k$ and *uses* the edges $e_1, \ldots, e_k$. We call v_0 the *start node* and v_k the *end node* of p, the nodes $v_1, \ldots, v_{k-1}$ are called *inner nodes* of p. Very often p is

[5] $\binom{V}{2} := \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$, i.e. $\binom{V}{2}$ consists of all subsets of V with two elements.

uniquely determined by the sequence of its nodes or by the sequence of its edges[6]. Then we also write $p = [v_0, v_1, \ldots, v_k]$ or $p = e_1 \circ \ldots \circ e_k$ to simplify notation.

p is called a *simple path* if $v_i \neq v_j$ for $0 \leq i < j \leq k$ and $e_i \neq e_j$ for $1 \leq i < j \leq k$. Informally, this means that p visits no node twice and uses each of its edges only once.
In an undirected multigraph the reversed sequence $[v_k, e_k, v_{k-1}, \ldots, v_1, e_1, v_0]$ is a path from v_k to v_0, we call it the *reversal of* p and denote it by p^{rev}.

If the end node of a path p is equal to the start node of a path q, we can *concatenate* them to a single path which we denote by $p \circ q$. More precisely, if $p = [v_0, e_1, v_1, \ldots, e_k, v_k]$ and $q = [v_k, e_k, v_{k+1}, \ldots, e_l, v_l]$, then $p \circ q = [v_0, e_1, v_1, \ldots, e_k, v_k, e_{k+1}, v_{k+1}, \ldots, e_l, v_l]$.

A *cycle* is a path $C = [v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k]$ such that $v_0 = v_k$, $k > 0$ and $e_i \neq e_{i+1}$ for $i = 1, \ldots, k-1$, i.e. C is a non-empty path where the start and the end node are identical and any two consecutive edges are distinct. Observe that C cannot be a simple path, but we can make an analogous definition: We say that C is a *simple cycle* if $v_i \neq v_j$ for $0 \leq i < j < k$ and $e_i \neq e_j$ for $1 \leq i < j \leq k$.

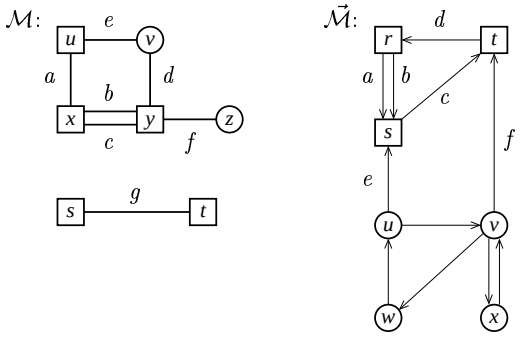


Figure 2.2: An undirected and a directed multigraph.

Example. Observe that the multigraph $\mathcal{M}$ in on the left-hand side of Figure 2.2 contains two edges, namely b and c, that are incident to same nodes, this explains the prefix "multi" in the word "multigraph". Observe that the directed multigraph $\vec{\mathcal{M}}$ on the right-hand side contains two edges (a and b) with identical source and target nodes.
$\mathcal{M}$ contains the path $p_1 = [u, e, v, d, y, b, x, c, y, f, z]$ from u to z. This path is not simple because y is visited twice. $p_2 = [u, e, v, d, y, f, z]$ is a simple path

[6]Every path of length at least two in an undirected multigraph is determined by the sequence of its edges. The same holds for every non-empty path in a directed multigraph.

from u to z. $C_1 = [x, b, y, c, x]$ is a cycle, but $[u, a, x, a, u]$ is not, because the edge a is used twice consecutively. The cycles C_1 and $C_2 = [u, a, x, c, y, d, v, e]$ are simple.
$\vec{p} = [r, a, s, c, t]$ is a simple path in $\vec{\mathcal{M}}$, but $[t, c, s, a, r]$ is not a path because the edges are traversed in the wrong direction. The path $\vec{c} = \vec{p} \circ d$ is a simple cycle.

Closely related to paths and cycles is the notion of reachability. We say that a node u *can reach* a node v if there is a (possibly empty) path from u to v. The *reachability relation* of a (directed or undirected) multigraph $\mathcal{M}$ is defined as follows:
$\text{Reach}(\mathcal{M}) := \{(u, v) \in V \times V \mid \exists \text{ a non-empty path from } u \text{ to } v \text{ in } \mathcal{M}\}$
The reader may wonder why we exclude empty paths in the definition above. The answer is that we want $\text{Reach}(\mathcal{M})$ to contain a tuple (u, u) only if there is cycle in $\mathcal{M}$ which visits u.

Another basic notion that is also related to paths is connectivity. Consider an undirected multigraph $\mathcal{M}$. A subgraph $\mathcal{M}'$ is called *connected* if any node u in $\mathcal{M}'$ can reach any node v in $\mathcal{M}'$. We say that $\mathcal{M}'$ is a *connected component* of $\mathcal{M}$ if $\mathcal{M}'$ is a maximal connected subgraph of $\mathcal{M}$. Observe that every node and every edge of $\mathcal{M}$ belongs to exactly one of its connected components.

For a directed multigraph $\vec{\mathcal{M}}$ we have a similar notion: A subgraph $\vec{\mathcal{M}}'$ is called *strongly connected* if any node u in $\vec{\mathcal{M}}'$ can reach any node v in $\vec{\mathcal{M}}'$ (by a directed path). A *strongly connected component* (SCC) of $\vec{\mathcal{M}}$ is a maximal strongly connected subgraph. Every node belongs to exactly one SCC, but there may be edges which do not belong to any SCC. Hence, the SCCs partition the node set of $\vec{\mathcal{M}}$. Since every set in this partition $\mathcal{P}$ induces an SCC of $\vec{\mathcal{M}}$, we can identify the SCCs of $\vec{\mathcal{M}}$ with the node sets in $\mathcal{P}$. It is easy to see that two nodes u and v belong to the same SCC iff there is a cycle which visits both of them.

In order to define the connected components of a directed multigraph $\vec{\mathcal{M}} = (V, E, \vec{inc})$, we define its *underlying undirected multigraph* $\mathcal{U}(\vec{\mathcal{M}}) := (V, E, inc)$, where $inc(e) := \{u, v\}$ iff $\vec{inc}(e) = (u, v)$. Informally, we obtain $\mathcal{U}(\vec{\mathcal{M}})$ by removing the arrowheads of all edges. A subgraph $\vec{\mathcal{M}}'$ is a *connected component* of $\vec{\mathcal{M}}$ if $\mathcal{U}(\vec{\mathcal{M}}')$ is a connected component of $\mathcal{U}(\vec{\mathcal{M}})$. We say that a (directed or undirected) multigraph is *connected* if it has only one connected component.
Example. The multigraph $\mathcal{M}$ in Figure 2.2 has two connected components $\mathcal{M}_1$ and $\mathcal{M}_2$, where $\mathcal{M}_1$ is the subgraph induced by the nodes $\{u, v, x, y, z\}$ and $\mathcal{M}_2$ is the subgraph induced by the edge g. $\vec{\mathcal{M}}$ has two SCCs which are induced by $\{r, s, t\}$ and $\{u, v, w, x\}$, but only one connected component.

Observe that e and f do not belong to any SCC of $\vec{\mathcal{M}}$.

We conclude this section with some remarks about the representation of multigraphs in memory. We assume that adding and removing a node or an edge can be done in time $O(1)$. Moreover, iterating over all incident, outgoing or incoming edges of a given node should take linear time in the number of these edges. The size of a representation of a multigraph with n nodes and m edges should be $O(n+m)$. These assumptions are fulfilled by a so-called *adjacency list representation* (see for example [MN99]).

2.2.1 Graphs, DAGs, trees and forests

One could define a graph as a multigraph where the incidence mapping is injective. In order to simplify notation we choose the following definition:

Definition 2.8 (graph) *An* undirected graph G *is a tuple* (V, E) *such that* V *is a finite set and* $E \subseteq \binom{V}{2}$. *A* directed graph $\vec{G}$ *is a tuple* (V, E) *such that* V *is a finite set and* $E \subseteq V \times V$. *In both cases* V *and* E *contain the* nodes *and the* edges *of* G, *respectively.*

It is easy to see that a (directed or undirected) graph (V, E) corresponds to the multigraph (V, E, id_E), where $id_E(e) = e$ for all $e \in E$. Thus all definitions from above carry over to graphs.

A directed graph $\vec{G}$ which does not contain a cycle is called a *directed acyclic graph* (DAG). If u and v are two nodes in a DAG $\vec{G}$ such that there is a path p from u to v, then u is called an *ancestor* of v and v is called a *descendant* of u. If p is not empty, then the relation is called *proper*. If p consists of the single edge (u, v), then we say that u is a *father* of v, and v is a child of u. Observe that a node x cannot be proper ancestor and a descendant of a node y, because $\vec{G}$ is acyclic.

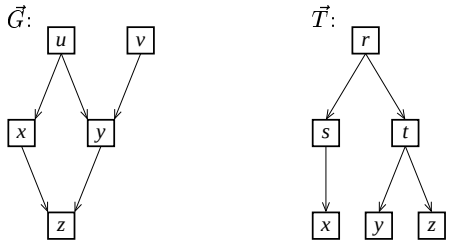


Figure 2.3: Two DAGs $\vec{G}$ and $\vec{T}$. $\vec{T}$ is a tree, $\vec{G}$ is not.

A DAG $\vec{T}$ is called a *rooted tree*, if it contains one node r without incoming edges and every node different from r has exactly one incoming edge. We say

that r is the *root* of $\vec{T}$. A node in $\vec{T}$ which has no outgoing edges is called a *leaf*. Observe that a tree is always connected.
Two nodes u and v in $\vec{T}$ always have a common ancestor, namely the root r. We say that a node w is the *lowest common ancestor* of u and v if w is a common ancestor of u and v and there is no proper descendant w' of w which is also a common ancestor of u and v. The node w is the node where the two paths p_u from r to u and p_v from r to v separate, i.e. the longest common prefix of p_u and p_v ends in w.
An undirected graph T is called a *tree* if any two nodes of T are connected by a unique path. A (directed or undirected) graph is said to be a *forest* if each of its connected components is a tree.
Example. The proper ancestors of the node y in the graph $\vec{G}$ in Figure 2.3 are its fathers u and v. The descendants of y are y itself and its child z. $\vec{G}$ is not a tree because y has two incoming edges (and there are two nodes without incoming edges). The graph $\vec{T}$ in the figure is a tree with the root r and the leaves x, y and z. The lowest common ancestor of y and z is t. The lowest common ancestor of x and z is the root r.

2.2.2 Matchings and bipartite graphs

As an introductory example consider the following problem: You are given an even-sized set of football teams and you are supposed to devise a schedule such that each team plays exactly one match. Let us assume that not any choice of two teams is an admissible pairing, but you are provided with a set of admissible pairings. This problem can be modelled as a graph problem: For each team there is a node, and two nodes are adjacent iff the respective teams may be paired. The problem is to find a set of edges M such that each node is incident to exactly one edge. (M is called a *perfect matching*.) Then M encodes a feasible schedule.
This motivates the following definition:

Definition 2.9 (matching) *Let $G = (V, E)$ be a graph. A set $M \subseteq E$ is called a* matching *in G if every node $v \in V$ is incident to at most one edge in M. A node $v \in V$ is called* matched *in M if it is incident to an edge in M; otherwise v is said to be* free *with respect to M. M is called* perfect *if all nodes in V are matched. Let V' be a subset of V, we call M a V'-*perfect matching *(or V'-*matching *for short) if all nodes in V' are matched.*

Very often the objects that have to be paired can be partitioned in two sets A and B, and in all admissible pairings one object belongs to A and

the other belongs to B. Consider e.g. the problem of assigning students to universities. Such a problem corresponds to a bipartite graph:

Definition 2.10 (bipartite graph) *A graph $G = (V, E)$ is called a* bipartite graph *with partition (A, B) if $V = A \dot\cup B$ and every edge $e \in E$ is incident to a node in A and a node in B.*

From now on we consider only undirected bipartite graphs. Most of the following theory also applies to arbitrary graphs, but in this thesis we will only deal with matchings in bipartite graphs. Let M and M' be two matchings in a graph $G = (V, E)$. In matching theory one often studies the symmetric difference $M \oplus M' := (M \setminus M') \cup (M' \setminus M)$. So $M \oplus M'$ consists of the edges that belong to exactly one of the two matchings. Informally, one could say that $M \oplus M'$ is the set of edges on which M and M' "disagree"; all other edges in E either belong to both matchings or to neither one of them. Consider the example in Figure 2.4. The graph $\hat{G}$ induced by $M \oplus M'$, which is shown on the right-hand side, contains a cycle $c = [a, u, b, v, a]$ and a path $p = [d, x, e, y, f, z]$. Since both c and p alternately use an edge in M and an edge in M', we call them *alternating paths*. The precise definition follows:

Definition 2.11 (alternating path) *Let M be a matching in a graph $G = (V, E)$. A path $p = [v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k]$ in G is called an* alternating path with respect to M *if the following holds:*

- *p alternately uses an edge in M and an edge that is not in M.*
- *If $v_0 \neq v_k$, then p is a simple path and*
  - *if v_0 is matched in M, then $e_1 \in M$ and*
  - *if v_k is matched in M, then $e_k \in M$.*

  If $v_0 = v_k$, then p is a simple cycle, k is even and exactly one of the edges e_1 and e_k belongs to M. In this case, p is called an alternating cycle.

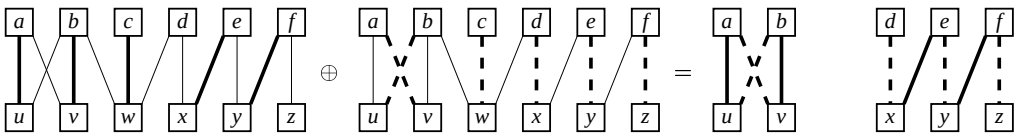


Figure 2.4: The symmetric difference of two matchings.

We can prove the following basic lemma about alternating paths:

Lemma 2.2 *Let M and M' be two matchings in a graph $G = (V, E)$ and let p be a an alternating path with respect to M. Then the following holds:*

1. *Every connected component of the graph induced by $M \oplus M'$ consists of an alternating path (with respect to either matching).*

2. *$M \oplus p$ is a matching in G (see Figure 2.5). Here $M \oplus p$ denotes the symmetric difference of M and the set of edges used by p.*

Proof. Every node v in the graph $\hat{G}$ induced by $M \oplus M'$ is incident to at most one edge of M and at most one edge of M'. Hence, its degree is either one or two. So each connected component is either a simple cycle or a simple path.
Consider a cycle c in $\hat{G}$. Each of its nodes is incident to exactly one edge in $M \setminus M'$ and exactly one edge in $M' \setminus M$. So its an alternating cycle with respect to both matchings.
Let q be a path in $\hat{G}$ that corresponds to an acyclic component. Each of its inner nodes is incident to exactly one edge in $M \setminus M'$ and exactly one edge in $M' \setminus M$. Therefore, q alternately uses edges in M and edges not in M. Let u denote the start node of q. If u is incident to an edge $e \in M$, then q starts with e because u and e belong to the same connected component of $\hat{G}$. An analogous argument can be made for the end node of q. So q is an alternating path wrt. M. The argument for M' is symmetric.

We come to the second statement. Let $\tilde{V}$ denote the set of nodes visited by p. By the definition of an alternating path $M \setminus p$ is a matching where all nodes in $\tilde{V}$ are free. Moreover, every node in $\tilde{V}$ is incident to at most one edge in $p \setminus M$. So $M \oplus p = (M \setminus p) \dot{\cup} (p \setminus M)$ is a matching. □

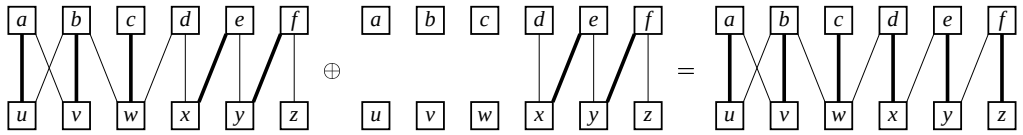


Figure 2.5: The symmetric difference of a matching and an alternating path.

The acyclic alternating path $p = [d, x, e, y, f, z]$ in Figure 2.5 starts and ends in a free node wrt. M. We call such a path an *augmenting path* because the cardinality of $M \oplus p$ is greater than that of M. If the cardinality of M is not maximal, then we can always find an augmenting path, as the following lemma shows:

Lemma 2.3 *If M is a matching in a graph G which does not have maximum cardinality, then there is an augmenting path p with respect to M.*

Proof. Let M' be a maximum cardinality matching in G. We consider an alternating path q from u to v in $M \oplus M'$ and distinguish four cases:

1. q is an alternating cycle. Then M and $M \oplus q$ match exactly the same nodes. Hence, the two matchings have the same cardinality.

2. q is acyclic and exactly one of the nodes u and v is matched in M. Then $|M \oplus q| = |M|$.

3. q is acyclic and both u and v are matched in M. Then $|M \oplus q| = |M| - 1$.

4. q is acyclic and both u and v are free in M, i.e. q is augmenting. Then $|M \oplus q| = |M| + 1$.

Let $q_1, \ldots, q_k$ denote the alternating paths in $M \oplus M'$. Since $M' = M \oplus q_1 \oplus q_2 \oplus \ldots \oplus q_k$ and $|M'| > |M|$, we conclude that at least one of the paths $q_1, \ldots, q_k$ is augmenting. □

Suppose we have a matching M in a bipartite graph $G = (V, E)$ with partition (A, B) and we want to search for an augmenting path wrt. M. Then it is often useful to construct the directed graph $\vec{G}_M = (V, \vec{E})$ as follows (cf. Figure 2.6):

- We direct the edges in E from A to B:
  For every edge $\{a, b\} \in E$ with $a \in A$ and $b \in B$, we put the edge (a, b) into $\vec{E}$.

- We add the reversal for every edge in M:
  For every edge $\{a, b\} \in M$ with $a \in A$ and $b \in B$, we insert the edge (b, a) into $\vec{E}$.

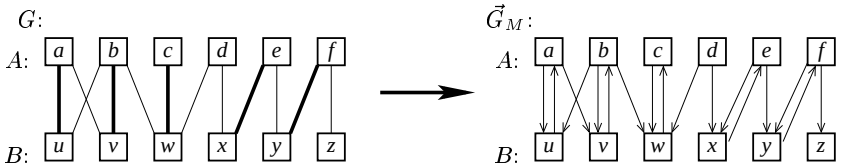


Figure 2.6: Constructing the directed graph $\vec{G}_M$ from G and a matching M.

Observe how the augmenting path $p = [d, x, e, y, f, z]$ in G translates to the directed simple path $\vec{p}$ in $\vec{G}_M$ from a free node d to a free node z, and

vice versa. So in order to find an augmenting path in G we can look for a simple path in $\vec{G}_M$ that starts in a free node in A and ends in a free node in B.

Assume now that we have an A-perfect matching M in G, which implies that M has maximum cardinality. We can use $\vec{G}_M$ to decide whether a given edge e in G can belong to some A-perfect matching in G. We will need the criterion below in Chapter 3:

Lemma 2.4 *Let G be a bipartite graph with partition (A, B) and M be an A-perfect matching in G. Let $e = \{a, b\}$ be an edge in G with $a \in A$ and $b \in B$. Then the following holds*

1. *If $|A| = |B|$, then e can belong to some perfect matching in G iff a and b belong to the same SCC of $\vec{G}_M$.*

2. *If $|A| \leq |B|$, then e can belong to some A-perfect matching in G iff a and b are in the same SCC of $\vec{G}_M$ or there is a simple path in $\vec{G}_M$ that starts with (a, b) and ends in a free node.*

Proof. The first statement follows immediately from the second statement, because there are no free nodes if $|A| = |B|$. So we only have to prove the second one.

Suppose that e belongs to some A-perfect matching M' in G. If $e \in M$, then we have the cycle $(a, b) \circ (b, a)$ in $\vec{G}_M$. So assume that $e \notin M$. Then e belongs to some alternating path p in $M \oplus M'$. If p is an alternating cycle, then it translates to a simple directed cycle in $\vec{G}_M$, which implies that a and b belong to the same SCC.
Suppose now that p is acyclic. Since M and $M \oplus p$ match all nodes in A, we conclude that both the start node and the end node of p are in B and exactly one of them is matched. Translating p to $\vec{G}_M$ yields a directed simple path that ends in a free node in B, because a free node in B has no outgoing edges.

Now we prove the converse. If a and b belong to the same SCC of $\vec{G}_M$, then there is a simple cycle $\vec{c}$ which uses the edge (a, b). Since G is bipartite, we conclude that the length k of $\vec{c}$ is even. If $k = 2$, then $\vec{c} = (a, b) \circ (b, a)$, which implies $e \in M$. Otherwise $\vec{c}$ corresponds to an alternating cycle c in G. Thus $M \oplus c$ is an A-perfect matching containing e.
Assume now that there is a simple path $\vec{p}' = (a, b) \circ \vec{p}''$ in $\vec{G}_M$ that ends in a free node in B. Let e_a denote the matching edge in M incident to a. Moreover, let p' denote the undirected path in G that corresponds to $\vec{p}'$. Then $p = e_a \circ p'$ is an alternating path wrt. M and $M \oplus p$ is an A-perfect matching containing e. □

2.3 Geometry

We discuss some geometrical notions that are used in the presentation of the *NonOverlapping*-constraint in Chapter 5.

Convexity and Topology

In this section we deal with the vector space $\mathbb{R}^d$ for some fixed dimension d and discuss convexity and some fundamental notions from topology. For two points p and q in $\mathbb{R}^d$ the straight line segment between p and q is the set $\overline{pq} = \{p + \lambda(q - p) \mid \lambda \in [0, 1]\}$. A set $S \in \mathbb{R}^d$ is called *convex* if for any two points $p, q \in S$, the straight line segment $\overline{pq}$ is also contained in S. Geometrically, this means that S has no recess and no protuberance (see Figure 2.7).

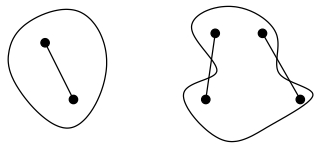


Figure 2.7: A convex and a non-convex set

The *convex hull* of a set S of points is denoted by $\mathcal{CH}(S)$ and defined as the intersection of all convex sets that contain S. Since the intersection of convex sets is convex, $\mathcal{CH}(S)$ is the smallest convex set which contains S. A visual way to obtain the convex hull is as follows: Place a nail into every point of S, put an elastic rubber band around all the nails, and let it snap around the nails. The convex hull of S is the area enclosed by the rubber band.

Now we discuss some notions from topology. For a point $p = (p_1, \ldots, p_d) \in \mathbb{R}^d$ the (Euclidean) *norm* of p is $\|p\| := \sqrt{p_1^2 + \ldots + p_d^2}$. For $\epsilon > 0$ and $p \in \mathbb{R}^d$ we define $B_\epsilon(p) := \{q \in R^d \mid \|q - p\| < \epsilon\}$, i.e. $B_\epsilon(p)$ is the ball with radius ϵ centred at p. For every $\epsilon > 0$ we call $B_\epsilon(p)$ a *neighbourhood of* p.
Consider a set $S \subseteq \mathbb{R}^d$ and a point $p \in \mathbb{R}^d$. We say that p is an *interior point of* S if there is a neighbourhood of p which is completely contained in S. The set of all interior points of S is called *the interior of* S and denoted by $\text{int}(S)$. We call S *open* if $S = \text{int}(S)$, and we say that S is *closed* if its complement $\mathbb{R}^d \setminus S$ is open. The point p lies on the *boundary of* S (denoted by ∂S) if any neighbourhood of p contains both a point in S and a point in $\mathbb{R}^d \setminus S$. (Note that p does not have to belong to S.) The *closure of* S is defined as $\overline{S} = S \cup \partial S$; it is easy to see that $\overline{S}$ is the smallest closed set that

contains S. Finally, we say that S is *bounded* if S is contained in some ball with finite radius.

Minkowski sums

Minkowski sums will become an important tool in Chapter 5. The formal definition is as follows:

Definition 2.12 (Minkowski Sum) *Let $P, Q \in \mathbb{R}^2$ be two sets of points. The Minkowski sum $P \oplus Q$ is the point set $\{p + q \,|\, p \in P \wedge q \in Q\}$.*

In the sequel we discuss some properties of Minkowski sums. We begin with a statement about the interior points of a Minkowski sum of two convex point sets:

Lemma 2.5 *Let P and Q be two convex sets in $\mathbb{R}^2$ with non-empty interior. Then $\text{int}(P) \oplus \text{int}(Q) = \text{int}(P \oplus Q)$.*

In order to prove this lemma, we need a claim, which we will show first:

Claim 2.1 *Let P be a convex set in $\mathbb{R}^2$, $p \in P$ and $p_i \in \text{int}(P)$, and let S be the line segment $\overline{pp_i}$. Then $S \setminus \{p\} \subseteq \text{int}(P)$.*

Proof. Since $p_i \in \text{int}(P)$, there is $\epsilon > 0$ with $\overline{B_\epsilon(p_i)} \subseteq P$. Let L be the line through p_i which is perpendicular to $p_i - p$. Denote by q and r the intersection of L with $\partial B_\epsilon(p_i)$, and let Δ be the triangle spanned by p, q, r (see left-hand side of Figure 2.8). By convexity we have $\Delta \subseteq P$. The claim follows from $S \setminus \{p, p_i\} \subseteq \text{int}(\Delta)$. □

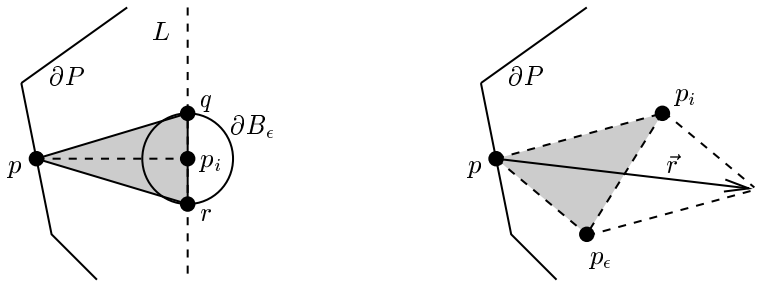


Figure 2.8: Visualization of the situations in the proofs of the Claim 2.1 and Lemma 2.5. All points in the shaded regions belong to $\text{int}(P)$.

Now we are ready to give the proof of the lemma:

Proof of Lemma 2.5. First we show $\text{int}(P) \oplus \text{int}(Q) \subseteq \text{int}(P \oplus Q)$. Fix $p \in \text{int}(P), q \in \text{int}(Q)$. Then there is $\epsilon > 0$ such that $B_\epsilon(p) \subseteq P$. Thus $P \oplus Q \supseteq B_\epsilon(p) \oplus q = B_\epsilon(p+q)$.

Now we prove $\text{int}(P) \oplus \text{int}(Q) \supseteq \text{int}(P \oplus Q)$. Fix a point $s = p + q \in \text{int}(P \oplus Q)$ with $p \in P$ and $q \in Q$. Observe that p and q could lie on the boundary of P and Q respectively. Our goal is to find a direction $\vec{r}$ with the following property: If we walk from p in direction $\vec{r}$, then we immediately hit interior points of P. And if we walk from q in the direction $-\vec{r}$, then we do not leave Q immediately. Thus there is some $\lambda > 0$ such that $p + \lambda\vec{r} \in \text{int}(P)$ and $q - \lambda\vec{r} \in Q$.

Let $p_i \in \text{int}(P)$ and $\vec{d_i} = p_i - p$. By Claim 2.1, $\vec{d_i}$ points from p to the interior of P, however $-\vec{d_i}$ might point from q to the outside of Q. Since $p+q \in \text{int}(P \oplus Q)$, we can find $\epsilon \in]0,1[$, $p' \in P$ and $q' \in Q$ with $p+q-\epsilon\vec{d_i} = p'+q'$. And hence, $q'-q = -(p'-p+\epsilon\vec{d_i}) =: -\vec{r}$. The convexity of Q implies that $q - \lambda\vec{r} \in Q$ for $0 \leq \lambda \leq 1$.
What remains to show is that $\vec{r}$ leads from p to the interior of P. By convexity of P, $p_\epsilon := p + \epsilon\vec{d_i} \in P$. Thus the whole triangle Δ spanned by p, p_i and p_ϵ is contained in P (see right-hand side of Figure 2.8). We distinguish two cases:

- If Δ is non-degenerate, i.e. its vertices are not collinear, then we are in the situation shown on the right-hand side of Figure 2.8. So $\vec{r}$ is one diagonal of the parallelogram induced by the vertices of Δ, and hence it leads from p to the interior of Δ.

- In the case that Δ is degenerate, the vectors $\vec{d_i}$ and $d := p' - p$ are linearly dependent, i.e. $d = \alpha\vec{d_i}$ for some $\alpha \in \mathbb{R}$. If $\alpha < 0$, then $p \neq p'$ and p lies on the segment $\overline{p'p_i}$. Claim 2.1 implies that $p \in \text{int}(P)$. So assume $\alpha \geq 0$. Since $\vec{r} = p' - p + \epsilon\vec{d_i} = (\alpha + \epsilon)\vec{d_i}$, $\vec{r}$ has the same direction as $\vec{d_i}$, which has been chosen to point from p into the interior of P.

In any case we can find $\lambda \in]0,1[$ with $p'' = p + \lambda\vec{r} \in \text{int}(P)$. We have $q'' = q - \lambda\vec{r} \in Q$ and $p'' + q'' = s$.
We choose $q_i \in \text{int}(Q)$, and let $\vec{e_i} = q_i - q''$. By Claim 2.1, we have $q'' + \gamma\vec{e_i} \in \text{int}(Q)$ for $0 < \gamma \leq 1$. Since $p \in \text{int}(P)$, we conclude that $\tilde{p} := p'' - \delta\vec{e_i} \in \text{int}(P)$ for some δ with $0 < \delta \leq 1$. Thus $\tilde{q} := q'' + \delta\vec{e_i} \in \text{int}(Q)$, and we have $\tilde{p} + \tilde{q} = s$. □

We want to point out that both conditions of Lemma 2.5 are necessary. The Minkowski sum of a line segment S and box B is a box again, see left-hand side of Figure 2.9. So $\text{int}(S \oplus B) \neq \emptyset$, but $\text{int}(S) = \emptyset$, which implies

$\text{int}(S) \oplus \text{int}(B) = \emptyset$.
The right-hand side of the figure shows a non-convex polygon P, a convex polygon Q and their sum $P \oplus Q$, which contains the interior point s (see marker). We observe that s cannot be represented as the sum of two interior points of P and Q (only as sum of two points on the boundaries of P and Q). And hence, $\text{int}(P) \oplus \text{int}(Q) \neq \text{int}(P \oplus Q)$.

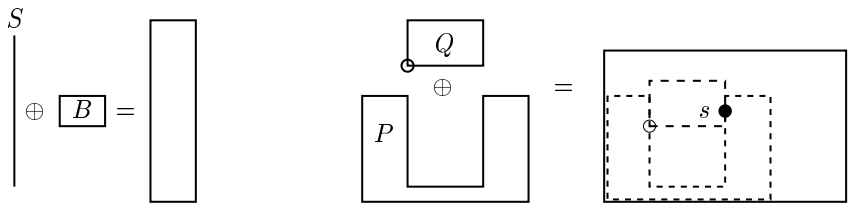


Figure 2.9: Examples showing that both conditions of Lemma 2.5 are necessary.

We conclude this section with a statement about convex combinations and Minkowski sums:

Lemma 2.6 *Let $Q \subseteq \mathbb{R}^2$ be convex and $p \in \mathbb{R}^2$ be a convex combination of $p_1, \ldots, p_n \in \mathbb{R}^2$, i.e. $p = \sum_{i=1}^n \lambda_i p_i$ with $\lambda_1, \ldots, \lambda_n \in [0,1]$ and $\sum_{i=1}^n \lambda_i = 1$. Then the following holds:*

$$p \oplus Q = \bigoplus_{i=1}^{n} \lambda_i (p_i \oplus Q)$$

Proof.

- $\subseteq$ Fix $q \in Q$. As $\sum_{i=1}^n \lambda_i = 1$, we have $p + q = \sum_{i=1}^n \lambda_i p_i + (\sum_{i=1}^n \lambda_i) q = \sum_{i=1}^n \lambda_i (p_i + q) \in \bigoplus_{i=1}^n \lambda_i (p_i \oplus Q)$.
- $\supseteq$ Fix $s \in \bigoplus_{i=1}^n \lambda_i (p_i \oplus Q)$. By the definition of a Minkowski sum, we have $s = \sum_{i=1}^n \lambda_i (p_i + q_i)$ with $q_1, \ldots, q_n \in Q$. As Q is convex, $q = \sum_{i=1}^n \lambda_i q_i \in Q$. And hence, $s = \sum_{i=1}^n \lambda_i (p_i + q_i) = p + q \in p \oplus Q$.

□

Polygons and polytopes

A *polygonal chain* is a sequence $C = \langle p_1, \ldots, p_n \rangle$ of points in the plane $\mathbb{R}^2$ such that segments $s_1 = \overline{p_1 p_2}, s_2 = \overline{p_2 p_3}, \ldots, s_{n-1} = \overline{p_{n-1} p_n}, s_n = \overline{p_n p_1}$

are disjoint except for common endpoints of consecutive segments. We call $p_1, \ldots, p_n$ the *vertices* and $s_1, \ldots, s_n$ the *edges* of the chain. We use $|C|$ to denote the number of vertices of C. C describes a set of points, namely the union of all its edges. In order to simplify notation, we denote this point set also by C. $\mathbb{R}^2 \setminus C$ consists of two (connected) open sets, one of which is bounded. Thus C splits the plane in a bounded region $B(C)$ and an unbounded region $U(C)$.

We can model C as a cyclic directed graph: The nodes of this graph are the vertices of C, and for $i = 1, \ldots, n$ the edge $s_i = \overline{p_i p_{i+1}}$ of the chain corresponds to an edge $e_i = (p_i, p_{i+1})$ in the graph, where $p_{n+1} := p_1$. Thus we obtain an orientation for the edges of C. It is easy to see that $B(C)$ is either locally to the left of each edge or locally to the right of each edge. In the former case we say that C has a *positive orientation*, and in the latter case C has a *negative orientation*. An example is depicted in Figure 2.10, we see that a chain has positive orientation if the vertices in the sequence $\langle p_1, \ldots, p_n \rangle$ are in counter-clockwise order.

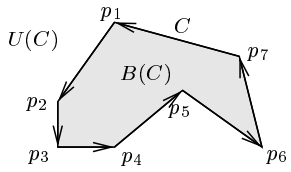


Figure 2.10: A positively oriented polygonal chain.

A positively oriented polygonal chain C defines a *polygon* P: P is the point set $C \cup B(C)$. Hence, $\partial P = C$ and $\mathrm{int}(P) = B(C)$. Sometimes we identify P and its defining polygonal chain. In particular, we use $|P|$ to denote the number of vertices of P.

Now we restrict our attention to convex polygons. A convex polygon P is uniquely defined by the (unordered) set V of its vertices, we have $P = \mathcal{CH}(V)$. P can also be written as the intersection of $|P|$ half-planes: Every edge $\overline{pq}$ of P corresponds to one half-plane H_{pq}, namely all points which lie to the left of the oriented line through p and q. In order to determine H_{pq}, we consider the outer normal vector $\vec{n}(p, q)$ of the edge. This vector is perpendicular to $\overline{pq}$ and points to the outside of P, thus $\vec{n}(p, q) = (q_y - p_y, p_x - q_x)$. H_{pq} is the set of all points (x, y) that satisfy the inequality $\langle \vec{n}(p, q), (x, y) \rangle - \langle \vec{n}(p, q), p \rangle \leq 0$. Here $\langle \cdot, \cdot \rangle$ denotes the scalar product, which is defined as $\langle (x_1, y_1), (x_2, y_2) \rangle := x_1 \cdot x_2 + y_1 \cdot y_2$.

Consider for example the triangle with the corner points $a = (1, 1)$, $b = (5, 1)$, $c = (1, 4)$. The corresponding normal vectors are $\vec{n}(a, b) = (0, -4)$,

$\vec{n}(b,c) = (3,4)$ and $\vec{n}(c,a) = (-3,0)$, which give rise to the half-planes $H_{ab} : y \geq 1$, $H_{bc} : 3x + 4y \leq 19$ and $H_{ca} : x \geq 1$.

Polygons are point sets in the two-dimensional plane. There exists a generalization for higher dimensions called *polytopes*. We restrict our attention to convex polytopes: A d-dimensional *convex polytope* P is the convex hull of a finite set of points $S \subseteq \mathbb{R}^d$ which contains at least three non-collinear points[7]. In the sequel we only talk about three-dimensional convex polytopes, i.e. we fix $d = 3$. The boundary of such a polytope P can be decomposed in (non-disjoint) features of lower dimension (see left-hand side of Figure 2.11): A *facet* of P is maximal subset of coplanar points on ∂P. Thus a facet of P is a convex polygon (embedded into $\mathbb{R}^3$). An *edge* of P is an edge of one of its facets, and a *vertex* of P is a vertex of one of its facets. It is easy to see that P is the convex hull of its vertices.

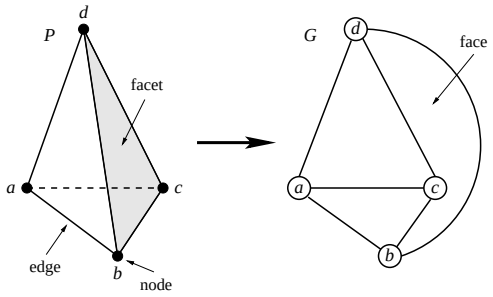


Figure 2.11: A polytope and a combinatorial representation as planar graph.

There are several ways to represent a polytope P. The choice depends on the computation which one wants to perform on P.

- pure vertex representation:
  As we have seen above, P is uniquely determined by the (unordered) set of its vertices.

- intersection of half-spaces:
  Suppose first that P is not planar, i.e. P is not contained in a plane. Each facet f of P gives rise to one half-space H_f (similar to the two-dimensional case, where every edge gave rise to a half-plane). H_f is uniquely characterized by the following properties: $P \subseteq H_f$, and the boundary of H_f is the plane containing f. Let $f_1, \ldots, f_k$ denote the

[7] The requirement that S contains at least three non-collinear points is non-standard, but convenient for us.

facets of P. Then $P = \bigcup_{i=1}^{k} H_{f_i}$.
If P is planar, then P can be written as the intersection of $n+2$ half-spaces where n is the number of edges of P: The plane p containing P is the intersection of two half-spaces, and every edge e of P gives rise to a half-space H_e such that $P \subseteq H_e$ and ∂H_e is perpendicular to p and contains e.

- combinatorial representation:
  We represent P as a planar graph[8] G: Each vertex/edge of P gives rise to a node/edge in G (like in the two dimensional case). Each facet of P corresponds to a *face* in the planar embedding of P. We will not define the notion of a *face* formally (for details see [MN99, Chapter 8]). In our example in Figure 2.11, the planar drawing of G splits the plane into four regions – three bounded ones and one unbounded one. The nodes and edges that bound a face of G correspond to the vertices and edges on the boundary of a facet of P, and vice versa.

We show now that the size of these representations is linear in the number of vertices of P. Here we assume that a point in $\mathbb{R}^3$ can be represented in constant size. (We want to point out that this result does not for higher dimensions, even if we need only constant size for a point.) We follow the presentation of de Berg et al. (see Theorem 11.1 in [dBvKOS00]).

Lemma 2.7 *Let P be a three-dimensional convex polytope with n vertices. The number n_e of edges of P is at most $3n-6$ and the number n_f of faces of P is at most $2n-4$. If P is not planar, then $n_f \geq \frac{1}{2}n+2$. Hence, any of the three representations above for P has size $\Theta(n)$.*

Proof. We use Euler's formula which states:

$$n - n_e - n_f = 2$$

Since every facet has at least three edges and every edge bounds exactly two facets, we have $3n_f \leq 2n_e$. Plugging this into Euler's formula we obtain $n + n_f - 2 \geq \frac{3}{2}n_f$, which implies $n_f \leq 2n-4$. Applying Euler's formula once more we get $n_e \leq 3n-6$.

If P not planar, then every vertex is incident to at least three edges. Thus $3n \leq 2n_e$, which implies (by Euler's formula) $n_f \geq 2 + \frac{3}{2}n - n = \frac{1}{2}n + 2$. (For the example in Figure 2.11 our bounds are tight.)

The claim on the size of the representations of P follows immediately from the discussion above. (Observe that $n_f = \Theta(n)$, if P is non-planar, and that $n_e = n$, if P is planar). □

[8] A graph is called *planar* if it can be drawn in the plane without edge crossings.

The three representations of a polytope P can be converted into each other in time $O(|P| \log |P|)$: From the combinatorial representation we can obtain the other representations in linear time. In order to convert the purely vertex representation into a combinatorial representation we can use an algorithm for computing the convex hull of a point set S, which runs in time $O(|S| \log |S|)$ (see [dBvKOS00, Chapter 11]). Moreover, the intersection of n half-spaces in $\mathbb{R}^3$ can be computed in time $O(n \log n)$ (cf. again [dBvKOS00, Chapter 11]).

Chapter 3

Sortedness and Alldiff

In this chapter we study two constraints: *Sortedness* and *Alldiff*. The constraint $Sortedness(X_1, \ldots, X_n; Y_1, \ldots, Y_n)$ takes as input two sequences of n variables and states that the second sequence $Y_1, \ldots, Y_n$ is obtained by sorting the elements of the first sequence $X_1, \ldots, X_n$ in non-decreasing order. The constraint $Alldiff(X_1, \ldots, X_n)$ takes as input n variables and holds if the elements in the sequence $X_1, \ldots, X_n$ are pairwise different.

Let us look at some variable assignments for which the two constraints hold and some which violate the constraint.

Sortedness	
Sortedness(1,3,1;1,1,3)	holds
Sortedness(2,1,3;1,2,4)	violated
Sortedness(5,2,3;3,2,5)	violated

Alldiff	
Alldiff (3,4,2)	holds
Alldiff (2,1,2)	violated

In the first example on the left-hand side *Sortedness* holds because sorting the sequence $1, 3, 1$ yields $1, 1, 3$; note that the same element may occur several times in a sequence. In the second example the constraint is violated because the second sequence is not a permutation of the first one. And in the last example the second sequence is not sorted.
Now we consider the examples for the *Alldiff* on the right-hand side. While in the first example the constraint clearly holds, we have a violation in the second one because the element 2 appears twice in the sequence.

For each of the two constraints we will develop a propagation algorithm that achieves bound-consistency, i.e. we assume that all variable domains are intervals and we show how to narrow them to the smallest possible intervals. For both constraints the best previous result was $O(n \log n)$. For the *Sortedness*-constraint it was obtained by Bleuzen-Guernalec and Colmerauer [BGC00], the algorithm for the *Alldiff*-constraint was given by Puget [Pug98]. The running time of our algorithms [MT00] will be $O(n)$ plus the time for

sorting the interval endpoints of the variable domains. For the *Sortedness*-constraint we can show that this is optimal. Let us compare our results to the previous results. Of course, we are never worse because we can always sort the interval endpoints in time $O(n \log n)$ (see for example [Meh84, Chapter II.1]), assuming that two endpoints can be compared in constant time. But we improve upon the previous results whenever we can sort the interval endpoints in linear time. This is for example the case when the interval endpoints are "small" integer numbers, i.e. when they are drawn from the range $[0..n^k - 1]$ for some fixed k (see [Meh84, Chapter II.2])[1].

Although the two constraints have quite different semantics we treat them in a single chapter because our propagation algorithms for them are very similar. In both cases we have to deal with a matching problem in a bipartite graph. A connection between the two constraints was already pointed out in [BGC00]. If the *Alldiff*-constraint encodes a permutation, which means $Dom(X_i) \subseteq [1..n]$, then $Alldiff(X_1, \ldots, X_n)$ is equivalent to $Sortedness(X_1, \ldots, X_n, \{1\}, \{2\}, \ldots, \{n\})$.

The work which we present in this chapter is based on the paper [MT00], which is joint work with Kurt Mehlhorn. We will be able to give simpler algorithms for the *Alldiff*-constraint and the presentation of both propagation algorithms should be clearer than in the paper. The chapter is divided in two parts, one for each constraint. We start with the *Sortedness*-constraint, for the matching problem that occurs here is easier to solve than the one for the *Alldiff*-constraint.

3.1 The Sortedness-Constraint

In this section we discuss the *Sortedness*-constraint. After giving an example which demonstrates the usefulness of this constraint, we provide a formal definition. Then we develop the our propagation algorithm. The section concludes with a discussion of related work.

3.1.1 Motivation

We describe an application of the *Sortedness*-constraint to a job-shop scheduling problem. This application was found by Older et al. [OSvE95]. We have to schedule n jobs on k identical machines. Each machine can execute any of the jobs, but only one at a time. Each job has a positive duration and may not be interrupted. For every machine m with $m \in [1..k]$ we may specify

[1] We take the n-nary representation of every number and interpret it as a string of length k over the alphabet $0, 1, \ldots, n-1$. The strings can be sorted in linear time.

an availability time A_m, every job on this machine must start after A_m. A schedule assigns to every job a starting time and the machine on which it is executed. It is called *feasible* if no two jobs that overlap in time are scheduled on the same machine and no job starts before its respective machine is available. In the sequel we will show how the *Sortedness*-constraint can be used to model this problem such that the solutions of the constraint program correspond one-to-one to the feasible schedules of the problem. In order to make the following presentation easier, we will neglect additional constraints like precedence constraints.

For modelling the problem it will turn out useful to introduce for $m \in [1..k]$ a dummy job j_m which is scheduled to be the first job on machine m and has its completion time set to A_m. Thus j_m occupies this machine until the first real job can be executed. The n real jobs are denoted by $j_{k+1}, \ldots, j_{k+n}$. We consider a schedule for the $k+n$ jobs and discuss how to decide whether it is feasible or not. Older et al. gave a very elegant answer to this question which involves *sorting* the starting and completion times of the jobs. Let $\sigma_{k+1}, \ldots, \sigma_{k+n}$ denote the start times of the real jobs sorted in ascending order, i.e. $\sigma_{k+1} \leq \ldots \leq \sigma_{k+n}$. Note that σ_{k+i} is in general not the start time of job j_{k+i}. Denote by $\tau_1, \ldots, \tau_{k+n}$ the sorted completion times of all jobs.
The crucial observation is that all completion times can be treated as *machine availabilities*: Whenever a job completes at time τ_i on some machine m, then this machine is available again until the next job begins which is scheduled for machine m.

Let us examine a schedule that is feasible. Consider the point in time σ_{k+i} (for some i in $[1..n]$). At this time $k+i$ jobs have been started (including dummy jobs). Since there are only k machines, at least i out of these $k+i$ jobs must have been completed at this time. And hence, we have $\tau_i \leq \sigma_{k+i}$.

We will show that the converse is also true: If $\tau_i \leq \sigma_{k+i}$ for all $i \in [1..n]$, then the schedule is feasible. Let ϕ and ψ denote sorting permutations of the jobs according to the start and completion times. To be more precise, ϕ is a permutation of $[k+1..k+n]$ such that $\sigma_{\phi(k+i)}$ is the starting time of job j_{k+i}, for $i = 1, \ldots, n$. And ψ is a permutation of $[1..k+n]$ such that job j_l is completed at time $\tau_{\psi(l)}$ for $l = 1, \ldots, k+n$. We describe how to construct a feasible schedule. Since the dummy jobs are scheduled to be the first job on their machine, the schedule is uniquely determined if we know for each job j_l its successor $j_{succ(l)}$ (in case there is one). For $l \in [1..k+n]$ we define $succ(l)$ as follows:

$$succ(l) := \begin{cases} \phi^{-1}(k+\psi(l)), & \text{if } \psi(l) \in [1..n] \\ none, & \text{otherwise} \end{cases}$$

If $succ(l) \neq none$, then $j_{succ(l)}$ is a real job and $\tau_{\psi(l)} \leq \sigma_{\phi(succ(l))}$, i.e. j_l ends before $j_{succ(l)}$ starts. (The latter follows immediately from the condition $\tau_i \leq \sigma_{k+i}$.) Since every real job has a positive duration, the starting times of the jobs in the sequence $j_{succ(l)}, j_{succ^2(l)}, \ldots$ strictly increase, hence the sequence contains no repetitions. Moreover, as ϕ and ψ are permutations, we see that every real job is the successor of exactly one job.
For $m = 1, \ldots, k$ we schedule the jobs $j_m, j_{succ(m)}, j_{succ^2(m)}, \ldots$ on machine m in that order. Thus the jobs on machine m do not overlap. Since every real job is placed on exactly one machine, we obtain a feasible schedule.

We finish this section with the constraint program which encodes the feasible schedules for the problem. We use the following variables:

- For every real job j_{k+i} we have a variable S_i for the start time, a variable T_i for the completion time, and the duration is stored in a variable D_i.
- We have variables $\sigma_{k+1}, \ldots, \sigma_{k+n}$ for the sorted start times of the real jobs and variables $\tau_1, \ldots, \tau_{k+n}$ for the completion times of all jobs.
- For $m = 1, \ldots, k$ the availability times of machine m is given by A_m.

The program looks as follows:
$Sortedness(S_1, \ldots, S_n; \sigma_{k+1}, \ldots, \sigma_{k+n})$
$Sortedness(A_1, \ldots, A_k, T_1, \ldots, T_n; \tau_1, \ldots, \tau_{k+n})$
$T_i = S_i + D_i$ for all $i \in [1..n]$
$\tau_i \leq \sigma_{k+i}$ for all $i \in [1..n]$

3.1.2 Definition

Consider a *Sortedness*-constraint on the variables $X_1, \ldots, X_n, Y_1, \ldots, Y_n$. For $i = 1, \ldots, n$ let D_i denote $Dom(X_i)$ and E_i denote $Dom(Y_i)$. As we have said above, we require that all domains are intervals, but we restrict ourselves neither to finite nor to integer intervals. We only assume that all domains are drawn from a linearly ordered universe $(U, \leq)$ such that $|U| \geq 2$ and two elements of U can be compared in constant time. For two elements a, b in U we define the interval $I = [a; b]$ to be the set $\{u \in U \mid a \leq u \leq b\}$. Note that I may be empty. We denote by $\underline{I}$ the lower endpoint a and by $\overline{I}$ the upper endpoint b.

Before we define the relation of the constraint, we introduce the mapping *sort* which maps every n-tuple over U to its sorted version, i.e. for $(d_1, \ldots, d_n) \in U^n$ we have $sort(d_1, \ldots, d_n) = (e_1, \ldots, e_n)$ with $e_1 \leq \ldots \leq e_n$ and there is a permutation π of $[1..n]$ s.th. $d_i = e_{\pi(i)}$ for $i = 1, \ldots, n$. Now we are ready to define the relation $\mathcal{S} := Rel(Sortedness(X_1, \ldots, X_n; Y_1, \ldots, Y_n))$.

It consists of all $2n$-tuples $(d_1, \ldots, d_n, e_1, \ldots, e_n)$ such that $(e_1, \ldots, e_n) = sort(d_1, \ldots, d_n)$ and $d_i \in D_i$, $e_i \in E_i$ for all i. Our task is to decide whether $\mathcal{S}$ is non-empty and, if so, to compute the minimal and maximal elements in the projection of $\mathcal{S}$ on each of its $2n$ components.

3.1.3 Propagation Algorithm

Since the last n components of any $2n$-tuple in $\mathcal{S}$ are sorted in non-decreasing order, we may assume from now on that $\underline{E}_i \leq \underline{E}_{i+1}$ and $\overline{E}_i \leq \overline{E}_{i+1}$ for $i = 1, \ldots, n-1$. If this holds, we say that the domains of the Y-variables are *normalized*. Normalization can be achieved algorithmically by setting $\underline{E}_i$ to $\max(\underline{E}_{i-1}, \underline{E}_i)$ for i from 2 to n and $\overline{E}_i$ to $\min(\overline{E}_i, \overline{E}_{i+1})$ for i from $n-1$ to 1.

Example. We use the following running example:

$$Sortedness(X_1, \ldots, X_5; Y_1, \ldots, Y_5)$$

with the respective variable domains:

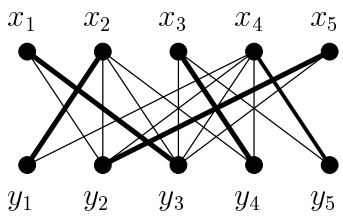
$D_1 = [7;10]$	$D_2 = [1;13]$	$D_3 = [13;15]$	$D_4 = [3;17]$	$D_5 = [5;6]$
$E_1 = [2;4]$	$E_2 = [4;7]$	$E_3 = [2;13]$	$E_4 = [12;19]$	$E_5 = [14;18]$

We observe that the domains of the Y-variables are not normalized. Normalization changes $\underline{E}_3$ to the value 4 and $\overline{E}_4$ to 18.

Our algorithm works on a bipartite graph G which we call the *intersection graph*. For every variable of the constraint we have a corresponding node in G, so the nodes are $\{x_i \mid 1 \leq i \leq n\}$ and $\{y_j \mid 1 \leq j \leq n\}$. There is an edge $\{x_i, y_j\}$ iff $D_i \cap E_j \neq \emptyset$. Given a tuple $(d_1, \ldots, d_n, e_1, \ldots, e_n) \in \mathcal{S}$, one can always construct a perfect matching in G as follows. If π denotes a permutation such that $d_i = e_{\pi(i)}$ for $i = 1, \ldots, n$, then the set $\{\{x_i, y_{\pi(i)}\} \mid 1 \leq i \leq n\}$ is a perfect matching. A partial converse also holds, a perfect matching in G implies the existence of certain tuples in $\mathcal{S}$, as we shall see below in Lemma 3.1. But let us revisit our running example first.

Example.

On the right-hand side we show the intersection graph for our running example. The bold edges indicate a perfect matching that corresponds to the following tuple in $\mathcal{S}$: $(\overset{8}{X_1}, \overset{3}{X_2}, \overset{14}{X_3}, \overset{15}{X_4}, \overset{6}{X_5}, \overset{3}{Y_1}, \overset{6}{Y_2}, \overset{8}{Y_3}, \overset{14}{Y_4}, \overset{15}{Y_5})$



As one can see in the example, the graph G tends to be dense, in fact it may have n^2 edges. So our algorithm will not create G explicitly.

We now come back to the correspondence between perfect matchings in G and tuples in $\mathcal{S}$. The lemma below shows how to construct tuples in $\mathcal{S}$ if one has a perfect matching. It will allow us to determine the projection of $\mathcal{S}$ onto its first n components.

Lemma 3.1 *Fix a perfect matching $\{\{x_i, y_{\pi(i)}\} \mid 1 \leq i \leq n\}$ in the intersection graph. For each i let d_i be an arbitrary element in $D_i \cap E_{\pi(i)}$. Then $(d_1, \ldots, d_n, e_1, \ldots, e_n) \in \mathcal{S}$, where $(e_1, \ldots, e_n) = sort(d_1, \ldots, d_n)$.*

Proof. Let $\phi = \pi^{-1}$ and consider the sequence $d_{\phi(1)}, \ldots, d_{\phi(n)}$. If this sequence is sorted, then $e_j = d_{\phi(j)}$ for all j, and we are done, because $d_{\phi(j)} \in E_j$, by the choice of the d's. So assume the sequence is not sorted, then there is a smallest index k with $d_{\phi(k)} > d_{\phi(k+1)}$. We get the following chain of inequalities $\underline{E}_k \overset{1}{\leq} \underline{E}_{k+1} \overset{2}{\leq} d_{\phi(k+1)} < d_{\phi(k)} \overset{3}{\leq} \overline{E}_k \overset{4}{\leq} \overline{E}_{k+1}$. Here the inequalities 1 and 4 follow from the assumption that the E's are normalized. The inequalities 2 and 3 are implied by $d_{\phi(k+1)} \in E_{k+1}$ and $d_{\phi(k)} \in E_k$ respectively. And hence, we have $d_{\phi(k)} \in E_{k+1}$ and $d_{\phi(k+1)} \in E_k$ so that we can swap the two elements. We construct a new permutation ϕ' with $\phi'(k) = \phi(k+1)$, $\phi'(k+1) = \phi(k)$, and $\phi'(j) = \phi(j)$ for all $j \notin \{k, k+1\}$. Again we have $d_{\phi'(j)} \in E_j$ for all j. And $d_{\phi'(1)}, \leq \ldots \leq d_{\phi'(k)} \leq d_{\phi'(k+1)}$. By applying the argument above until we obtain a sorting permutation of the d's, we can prove the claim. □

Before we explain how to narrow the domains of $X_1, \ldots, X_n$ in the corollary below, we introduce the *reduced intersection graph*: This graph is obtained by removing all edges from the intersection graph which cannot belong to any perfect matching.

Corollary 3.1 (Narrowing of X-domains) *Fix $i \in [1..n]$ and let S_i be the projection of $\mathcal{S}$ onto the i-th component. Let H denote the reduced intersection graph. Then $S_i = D_i \cap \bigcup_{\{x_i, y_j\} \in H} E_j$. In particular, we get $\underline{S}_i = \max(\underline{D}_i, \underline{E}_l)$ and $\overline{S}_i = \min(\overline{D}_i, \overline{E}_h)$ where y_l and y_h are the y-nodes adjacent to x_i in H with minimal and maximal index respectively.*

Proof. It suffices to show that $S_i = D_i \cap \bigcup_{\{x_i, y_j\} \in H} E_j$.

- $\subseteq$: For any element $d_i \in S_i$ we find a tuple in $\mathcal{S}$ whose i-th component equals d_i. This tuple corresponds to a perfect matching M in H. Let $\{x_i, y_j\} \in M$ denote the matching edge incident to x_i. Then $d_i \in D_i \cap E_j$.
- $\supseteq$: Consider an edge $e = \{x_i, y_j\}$ in H. Then there is a perfect matching in the intersection graph containing e. By Lemma 3.1 we have $D_i \cap E_j \subseteq S_i$.

□

We want to point out that we cannot use a similar strategy for narrowing the Y-domains[2]. We will show this with the aid of our running example. If one exchanges the mates of x_1 and x_5, one obtains a perfect matching M that contains the edge $\{x_1, y_2\}$. But there is no solution of the constraint where Y_2 is assigned the value 7, although this value is in $D_1 \cap E_2$. This can be seen as follows. Consider a tuple $(d_1, \ldots, d_5, e_1, \ldots, e_5)$ which satisfies the example constraint. By inspecting the domains of the variables, we find $e_1 < d_5 < d_1 < e_4 \leq e_5$. (The last inequality follows from the fact that the e's are sorted.) Since the e's are a permutation of the d's, we obtain $e_2 = d_5$. This means that in any solution Y_2 can only take a value in D_5.

In the sequel we will show how to find the edges that can belong to some perfect matching in the intersection graph. First we compute a certain perfect matching, and then we can determine for any edge whether it can belong to any perfect matching or not. In order to do this efficiently we exploit a crucial property of the intersection graph. Let us look at our running example. In the table below we list for any x-node its neighbours (i.e. its adjacent nodes) on the y-side:

x_1	x_2	x_3	x_4	x_5
y_2, y_3	y_1, y_2, y_3, y_4	y_3, y_4, y_5	y_1, y_2, y_3, y_4, y_5	y_2, y_3

We see that the neighbours of an x-node form an "interval" in the y-nodes. Or more formally, if y_l and y_h are two neighbours of a node x_i with $l \leq h$, then for all $j \in [l..h]$ the node y_j is adjacent to x_i. This property of the intersection graph follows directly from the normality of the domains of the Y-variables. (We have $\underline{E}_j \leq \underline{E}_h \leq \overline{D}_i$ and $\underline{D}_i \leq \overline{E}_l \leq \overline{E}_j$, and hence $D_i \cap E_j \neq \emptyset$.)

Glover called bipartite graphs with this property *convex* and gave a simple matching algorithm for them (see [Glo67] and [Law76, Section 6.6.6]): For a node v let $N(v)$ denote the set of its neighbours. For $j = 1, \ldots, n$ we determine for every node y_j its matching mate $x_{\phi(j)}$. Assume that $y_1, \ldots, y_{j-1}$ are already matched. Then the candidates for y_j are all its free neighbours, i.e. the nodes in $N(y_j) \setminus \{x_{\phi(1)}, \ldots, x_{\phi(j-1)}\}$. From the set of candidates we choose the node x_i such that $\overline{D}_i$ is minimal, and define $\phi(j) := i$. In Figure 3.1, we give an intuitive explanation for this choice of x_i. Consider another candidate x_c, i.e $\overline{D}_i \leq \overline{D}_c$. Since the interval of x_c on the y-side ends later than that of x_i, we see the following: All currently free nodes on the y-side that can be matched by x_i can also be matched by x_c, but x_c can (possibly) match some nodes which x_i cannot match. So it is reasonable to use x_i now and save x_c for later.

[2]Actual reason: Not every perfect matching corresponds to a "sorting permutation".

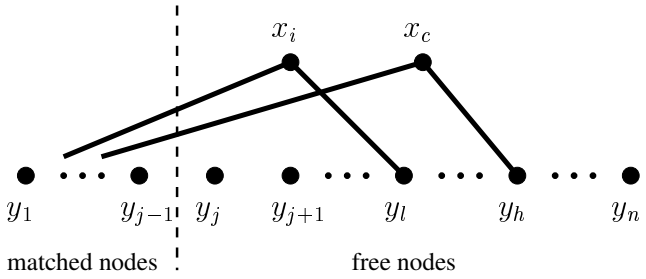


Figure 3.1: The nodes $y_j, \ldots, y_l$ can be matched with both x_i and x_c, the nodes $y_{l+1}, \ldots, y_h$ only with x_c. So we match y_j with x_i keeping x_c for later.

Example. Let us look how Glover's algorithm constructs a perfect matching in our running example. We give a table which shows for y_j the set of candidates (sorted according to $\overline{D}$) and the choice of $\phi(j)$.

y_j	$N(y_j) \setminus \{x_{\phi(1)}, \ldots, x_{\phi(j-1)}\}$	$\phi(j)$
y_1	x_2, x_4	2
y_2	x_5, x_1, x_4	5
y_3	x_1, x_3, x_4	1
y_4	x_3, x_4	3
y_5	x_4	4

So we obtain exactly the matching shown in the drawing of the intersection graph on page 43.

Lemma 3.2 (Glover) *If the intersection graph has a perfect matching, the algorithm above constructs one.*

Proof. Assume that the intersection graph has a perfect matching M. We use induction on j to show that there is a perfect matching M_j which matches y_k with $x_{\phi(k)}$ for $k = 1, \ldots, j$. The claim holds for $j = 0$ with $M_0 = M$. So assume $j > 0$. If M_{j-1} matches y_j with $x_{\phi(j)}$, we set $M_j = M_{j-1}$ and are done. Otherwise M_{j-1} matches y_j with some other node x_c and $x_{\phi(j)}$ with some node y_r. From the definition of $\phi(j)$ we conclude $\overline{D}_{\phi(j)} \leq \overline{D}_c$. Since the nodes $y_1, \ldots, y_{j-1}$ are matched with $x_{\phi(1)}, \ldots, x_{\phi(j-1)}$, we have $r > j$. As y_r lies to the right of y_j and the interval of x_c ends later than that of $x_{\phi(j)}$, we get $y_r \in N(x_c)$. Thus we can exchange the mates of y_j and y_r to construct M_j from M_{j-1}, i.e. we match y_j with $x_{\phi(j)}$ and y_r with x_c. □

We discuss how to implement Glover's algorithm efficiently. An important observation is the following. Suppose that the intersection graph

contains a perfect matching, and consider the iteration j of the algorithm. All free neighbours of the nodes $y_1, \ldots, y_{j-1}$ are also neighbours of y_j. In other words, as soon as an x-node becomes a candidate, it remains one until it is matched. Formally this means that all nodes in the set $S_j = N(y_1, \ldots, y_{j-1}) \setminus \{x_{\phi(1)}, \ldots, x_{\phi(j-1)}\}$ are contained in $N(y_j)$. This can be seen as follows. Assume the observation is false, i.e. there is a node $x_i \in S_j \setminus N(y_j)$. Since x_i is a neighbour of some y_k with $k < j$ but not of y_j, its interval in the y-nodes ends before the interval of y_j, i.e. $\overline{D}_i < \underline{E}_j$. So x_i is not a neighbour of $y_j, \ldots, y_n$. After Glover's algorithm has matched $y_1, \ldots, y_{j-1}$, there are at most $n - j$ candidates for the remaining $n - j + 1$ nodes on the y-side, and hence it will get stuck. By Lemma 3.2 this cannot happen if the intersection graph has a perfect matching.

The implementation of Glover's algorithm which is probably most suggesting maintains a priority queue P. After iteration $j - 1$, the queue P contains the set S_{j-1} sorted according to the upper interval endpoint $\overline{D}$ of the corresponding domains. In iteration j we insert into P those neighbours of y_j which are not neighbours of $y_1, \ldots, y_{j-1}$, these are the nodes x_i with $\overline{E}_{j-1} < \underline{D}_i \leq \overline{E}_j$. Now, P contains all candidates for y_j, and – if G has no perfect matching – maybe some x-nodes which are not neighbours of y_j. If P is not empty, we extract a node x_i from P with smallest $\overline{D}_i$. If P is empty or $\overline{D}_i < \underline{E}_j$, we detect that the intersection graph has no perfect matching, which implies that $\mathcal{S}$ is empty. Otherwise we set $\phi(j) = i$ and continue. Since every operation on P takes time $O(\log n)$, this implementation, which is shown in Algorithm 3.1, has complexity $O(n \log n)$.

Now we show how the algorithm can be implemented in linear time, if one knows the sorting of the X-variables according to both the lower and the upper endpoints of their domains. Let σ, τ denote permutations of $[1..n]$ with $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(n)}$ and $\overline{D}_{\tau(1)} \leq \ldots \leq \overline{D}_{\tau(n)}$. We replace the priority queue by an instance of the offline-min problem [AHU74, Chapter 4.8], which can be solved in linear time using the union-find data structure by Gabow and Tarjan [GT85]. The offline-min problem can be described as follows: one is given a sequence of the priority queue operations *insert* and *extractmin*, and one has a sorting permutation of the elements inserted by the *insert* operations. The goal is to check whether the sequence is valid, i.e. no *extractmin* is performed on an empty queue, and – if so – to compute the pairs of corresponding *extractmin* and *insert* operations in the sequence.
Since in Algorithm 3.1 the sequence does not depend on the outcome of the *extractmin* operations (if it runs to completion), we can easily construct the whole sequence offline. If we simply delete lines 8 – 10 and 12 – 15 in Algorithm 3.1, we obtain an algorithm which computes the desired sequence in linear time. Of course, we have to check that the sequence is valid and

Algorithm 3.1 Finding a perfect matching in the intersection graph G

Procedure: GloverWithPQ($D_1, \ldots, D_n$, $E_1, \ldots, E_n$, σ)
Require: σ is a permutation of $[1..n]$ s.th. $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(n)}$.
1: $P \leftarrow [\,]$ // PQ stores x-indices $[i_1, \ldots, i_k]$ s.th. $\overline{D}_{i_1} \leq \ldots \leq \overline{D}_{i_k}$
2: $s \leftarrow 1$
3: **for** $j = 1$ **to** n **do**
4: **while** $s \leq n$ and $\underline{D}_{\sigma(s)} \leq \overline{E}_j$ **do**
5: insert $\sigma(s)$ into P
6: $s \leftarrow s + 1$
7: **end while**
8: **if** P is empty **then**
9: report "no perfect matching in G" and terminate
10: **end if**
11: extract the first element i from P, i.e. one with smallest $\overline{D}_i$
12: **if** $\overline{D}_i < \underline{E}_j$ **then**
13: report "no perfect matching in G" and terminate
14: **end if**
15: $\phi(j) \leftarrow i$
16: **end for**
17: return ϕ

that for $j = 1, \ldots, n$ the j-th *extractmin* corresponds to an *insert*(i) with $\overline{D}_i \geq \underline{E}_j$, otherwise no perfect matching exists. Altogether, we get a linear time algorithm that can decide the existence of a perfect matching in the intersection graph and if possible compute one.

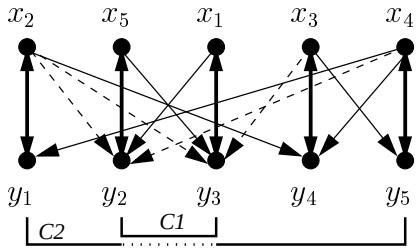
We have computed a distinguished perfect matching in G. In order to compute the reduced intersection graph H, we have to find all edges of G that can belong to some perfect matching. These edges are easy to find, we use the same characterization that was employed by Régin in his arc-consistency algorithm for the *Alldiff*-constraint. It is based upon the strongly connected components of a directed graph. Let u, v be two nodes in a directed graph, we say that u *can reach* v if there is a directed path from u to v. A set C of nodes is called *strongly connected* if any two nodes $u, v \in C$ can reach each other. And C is a *strongly connected component* (SCC) if it is a maximal strongly connected set of nodes. Now we can formulate the characterization:

Lemma 3.3 *Assume that M is a perfect matching in G. Let us construct the* oriented intersection graph $\vec{G}$ *by directing all edges in G from their x-endpoint to their y-endpoint and adding the reverse edge for all edges in M.*

An edge $\{x_i, y_j\}$ belongs to some perfect matching of G iff x_i and y_j lie in the same strongly connected component of $\vec{G}$.

Proof. See Lemma 2.4. □

Example. We revisit our running example:
On the right-hand side we have drawn the oriented intersection graph. The x-nodes have been sorted such that every node is located above its matching mate on the y-side. Every bold edge corresponds to a matching edge, it has two arrow heads because it represents two directed edges in $\vec{G}$.



The edges in dashed style connect nodes in different SCCs, and hence they do not belong to any perfect matching. Or in other words, these are exactly the edges that have to be deleted in order to obtain the reduced intersection graph.
As we have indicated, there are two SCCs $C_1 = [y_2, x_5, y_3, x_1]$ and $C_2 = [y_1, x_2, y_4, x_3, y_5, x_4]$. We see that C_1 is "nested" in C_2, this shows that the y-nodes of an SCC do not have to form an interval.

We develop an algorithm that computes the strongly connected components of $\vec{G}$ in time $O(n)$, which is not trivial since $\vec{G}$ may have up to $\Omega(n^2)$ edges. We use the algorithm of Cheriyan and Mehlhorn [CM96] as a basis and adapt it to the special structure of our graph.

We observe that a node y_j and its matching mate $x_{\phi(j)}$ always belong to the same SCC because of the edges $(y_j, x_{\phi(j)})$ and $(x_{\phi(j)}, y_j)$ in $\vec{G}$. We represent a component C as a list $[y_{j_1}, x_{\phi(j_1)}, \ldots, y_{j_k}, x_{\phi(j_k)}]$ such that $j_1 < \ldots < j_k$. We use $left\_y(C)$ to denote j_1 and $right\_y(C)$ for j_k. Moreover, we say that a component C_1 can reach a component C_2 if some node in C_1 can reach some node in C_2. (This implies that every node in C_1 can reach every node in C_2.)

Algorithm 3.2 explores the graph $\vec{G}$ from left to right[3] and maintains the SCCs of the currently explored graph. To be more precise, it computes the SCCs of the graphs $\vec{G}_0, \ldots, \vec{G}_n$ where $\vec{G}_j$ is the subgraph of $\vec{G}$ induced by the nodes $y_1, \ldots, y_j$ and their matching mates on the x-side. Note that $\vec{G}_0$ is empty and $\vec{G}_n = \vec{G}$. The components are stored in two data structures: a stack *CS* and a list *SCCs*. In *CS* we store tentative components which are strongly connected but may grow as the exploration of $\vec{G}$ goes on. In contrast to this, the components in *SCCs* are completed, i.e. they are SCCs of the final graph $\vec{G}$.

[3] This means the nodes are scanned in the order $y_1, x_{\phi(1)}, y_2, x_{\phi(2)}, \ldots, y_n, x_{\phi(n)}$.

Algorithm 3.2 Computing the SCCs of the oriented intersection graph $\vec{G}$

Function: ComputeSCCs($D_1, \ldots, D_n$, $E_1, \ldots, E_n$, ϕ)

1: $SCCs \leftarrow$ empty list
2: $CS \leftarrow$ empty stack
3: **for** j=1 **to** n **do**
4: **while** CS not empty and $\overline{D}_{\phi(right\_y(top(CS)))} < \underline{E}_j$ **do**
// i.e. $top(CS)$ cannot reach y_j or a node to the right of it
5: pop C' from CS
6: append C' to $SCCs$
7: **end while**
8: $i \leftarrow \phi(j)$
9: $C \leftarrow [y_j, x_i]$
10: **while** CS not empty and $\underline{D}_i \leq \overline{E}_{right\_y(top(CS))}$ **do**
// i.e. C can reach $top(CS)$
11: pop C' from CS
12: $C \leftarrow C' \circ C$ // merge components
13: **end while**
14: push C onto CS
15: **end for**
16: **while** CS not empty **do**
17: pop C' from CS
18: append C' to $SCCs$
19: **end while**
20: return $SCCs$

We will now explain the details of the algorithm and prove its correctness based on the invariants below. When line 15 is executed the following holds:

I1: Every component C in the list $SCCs$ is a strongly connected component of $\vec{G}$. In particular, C cannot reach $y_j, \ldots, y_n$.

I2: The union $SCCs \cup CS$ consists of the SCCs of the currently explored graph $\vec{G}_j$.

I3: Let $CS = \langle C_1, C_2, \ldots, C_t \rangle$ (where C_t is the top element). If $l < h$ then C_l can reach C_h but not vice versa and $right\_y(C_l) < left\_y(C_h)$.

I4: Let C denote a component in $SCCs \cup CS$ and let i_r be $\phi(right\_y(C))$, i.e. x_{i_r} is the mate of the rightmost y-node of C. Then for all $x_i \in C$ we have $\overline{D}_i \leq \overline{D}_{i_r}$.

When Algorithm 3.2 enters the for-loop (line 3) for the first time the invariants clearly hold with $j = 0$. So let us now consider an iteration j with $j > 0$ and assume that the invariants hold for $j - 1$ at the beginning. First we identify some components on CS that can be declared completed and move them to $SCCs$ (lines 4–7). When is the topmost component C' of CS *not* completed? We will see that this can only be the case if y_j is a neighbour of x_{i_r}, where $i_\mathrm{r} = \phi(right\_y(C'))$. We know by invariant I2 that C' is an SCC of $\vec{G}_{j-1}$. So if it is not an SCC of $\vec{G}$, then it must be able to reach one of the nodes $y_j, \ldots, y_n$. By invariant I3, C' cannot reach nodes in other components on CS; and by invariant I1, the components in $SCCs$ cannot reach any of the nodes $y_j, \ldots, y_n$. And hence, there must be a node x_l in C' itself such that x_l is a neighbour of some y_k with $j \leq k \leq n$. Recalling that the neighbours of x_l form an interval and the mate of x_l lies to the left of y_j, we can conclude that x_l is incident to y_j. By invariant I4 this implies that x_{i_r} is also a neighbour of y_j. So after the while-loop in line 4 has finished, invariant I1 holds.

When the algorithm reaches line 8, CS is either empty or its topmost component can reach y_j. By invariant I3, this implies that all components on CS can reach $C = [y_j, x_i]$, where x_i is the mate of y_j. So the SCC of y_j in $\vec{G}_j$ is the union of C with all components on CS that C can reach. From invariant I3 we infer that C can reach a component C' on CS iff x_i is a neighbour of some y-node in C'. Since the neighbours of x_i form an interval, this is equivalent to $\underline{D}_i \leq \overline{E}_{right\_y(C')}$. Invariant I3 implies that the components with this property are a suffix of CS. After merging them with C and pushing C on CS, invariant I3 holds again. Clearly, the final C is the SCC of y_j in $\vec{G}_j$. Invariant I2 follows from the fact that every other SCC of $\vec{G}_j$ is also an SCC of $\vec{G}_{j-1}$.
We have to show that invariant I4 holds for C after merging it with C' in line 12. Let $j' = right\_y(C')$ and $i' = \phi(j')$. Then what we have to prove is $\overline{D}_{i'} \leq \overline{D}_i$. Observe that x_i is a neighbour of $y_{j'}$ because C and C' are merged. Consider the point in time when Glover's algorithm matched the node $y_{j'}$ with $x_{i'}$. Since x_i is a neighbour of $y_{j'}$ and was free at that time, our claim follows.

After the termination of the for-loop, all remaining components on CS are SCCs of the final graph because of invariant I2 and $\vec{G}_n = \vec{G}$. Thus, these components can be moved to $SCCs$.

We want to say a few words about the implementation of the algorithm. First it is clear that we only have to store the indices of the y-nodes in a component C, for the x-nodes are the matching mates of the y-nodes. If we keep the indices in a list in ascending order, then we can determine

$right\_y(C)$ and merge components in constant time. Moreover, the number of push operations as well as the number of pop operations is bounded by n. And hence, the whole algorithm runs in time $O(n)$.

Example. We show how the algorithm computes the SCCs of our running example. In Table 3.1 we give a trace of its computation. The first two columns contain the value of the loop variable j and the line of the pseudo-code. The next two columns give the state of the data structures CS and C after the algorithm has executed the respective line. We do not list the state of $SCCs$, but we indicate the completion of a component by a comment in the last column.

j	line	CS	C	comment
1	3	$\langle\rangle$	$[y_1, x_2]$	new component
2	3	$\langle[y_1, x_2]\rangle$	-	
2	9	$\langle[y_1, x_2]\rangle$	$[y_2, x_5]$	new component
3	3	$\langle[y_1, x_2], [y_2, x_5]\rangle$	-	
3	9	$\langle[y_1, x_2], [y_2, x_5]\rangle$	$[y_3, x_1]$	new component
3	12	$\langle[y_1, x_2]\rangle$	$[y_2, x_5, y_3, x_1]$	merge
4	3	$\langle[y_1, x_2], [y_2, x_5, y_3, x_1]\rangle$	-	
4	6	$\langle[y_1, x_2]\rangle$	-	SCC $[y_2, x_5, y_3, x_1]$
4	9	$\langle[y_1, x_2]\rangle$	$[y_4, x_3]$	new component
5	3	$\langle[y_1, x_2], [y_4, x_3]\rangle$	-	
5	9	$\langle[y_1, x_2], [y_4, x_3]\rangle$	$[y_5, x_4]$	new component
5	12	$\langle[y_1, x_2]\rangle$	$[y_4, x_3, y_5, x_4]$	merge
5	12	$\langle\rangle$	$[y_1, x_2, y_4, x_3, y_5, x_4]$	merge
5	15	$\langle[y_1, x_2, y_4, x_3, y_5, x_4]\rangle$	-	
5	19	-	-	SCC $[y_1, \ldots, x_4]$

Table 3.1: Computation of the SCCs of the running example

Now we discuss the task of narrowing the domains of the X-variables. Let us consider a variable X_i and let S_i denote the projection of $\mathcal{S}$ onto the i-th component. Suppose we want to determine $\underline{S}_i$. By Corollary 3.1 and Lemma 3.3, we have to do the following. Let $C = [x_{\phi(j_1)}, y_{j_1}, \ldots, x_{\phi(j_k)}, y_{j_k}]$ be the SCC of x_i in $\vec{G}$ such that $j_1 < \ldots < j_k$. We look at $j_1, \ldots, j_k$ in that order until we find the first node y_{j_κ} which is a neighbour of x_i in G. So j_κ is the first index in the sequence with $\underline{D}_i \leq \overline{E}_{j_\kappa}$. And we get $\underline{S}_i = \max(\underline{D}_i, \underline{E}_{j_\kappa})$.

Assume now that we have an ordering $x_{i_1}, \ldots, x_{i_k}$ of the x-nodes in C such that $\underline{D}_{i_1} \leq \ldots \leq \underline{D}_{i_k}$. Then we can determine $\underline{S}$ for all the x-nodes in C in time $O(k)$. All we have to do is to merge the sequence $\underline{D}_{i_1} \ldots \underline{D}_{i_k}$ with

$\overline{E}_{j_1} \ldots \overline{E}_{j_k}$. Note that the latter sequence is also non-decreasing due to the normalization of the Y-domains.
The question is now how we can find the sorting of the x-nodes of C. We may assume that we have a global sorting of all x-nodes of $\vec{G}$ such that $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(n)}$. So we can compute the order for each component with bucket sort: First we generate a bucket (i.e. a list) for every SCC, then we label every x-node with the respective bucket. Finally we consider the nodes in the order $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ and append every node at the end of its bucket.

We show now that the ordered sequence of x-nodes does not have to be generated explicitly for each SCC: Suppose that every component is represented by the sequence of the indices of the y-nodes sorted in increasing order, i.e. $C = \langle j_1, \ldots, j_k \rangle$. For every component we maintain an iterator *iter*. This is a data structure which is similar to a pointer. It references an item of the sequence and supports two operations: *iter*.`YIdx` returns the referenced item, and *iter*.`advance` makes *iter* reference the next element in the sequence. With this data structure we can compute $\underline{S}_1, \ldots, \underline{S}_n$ as shown in Algorithm 3.3. A symmetric procedure can be used to determine $\overline{S}_1, \ldots, \overline{S}_n$.

Algorithm 3.3 Narrowing of the X-domains (lower endpoints)

Procedure: NarrowXDomsLE($D_1, \ldots, D_n$, $E_1, \ldots, E_n$, σ, ϕ, *SCCs*)
Require: σ is a permutation of $[1..n]$ s.th. $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(n)}$.

```
 1: for all C = ⟨j_1, ..., j_k⟩ ∈ SCCs do
 2:   generate iterator iter and make it reference the first element of C
 3:   for κ = 1 to k do
 4:     label x_{φ(j_κ)} with iter
 5:   end for
 6: end for
 7: for i = 1 to n do
 8:   iter ← iterator of x_{σ(i)}
 9:   while E̅_{iter.YIdx} < D̲_{σ(i)} do
10:     iter.advance
11:   end while
12:   // y_{iter.YIdx} is leftmost neighbour of x_{σ(i)} in its SCC
13:   S̲_{σ(i)} ← max(D̲_{σ(i)}, E̲_{iter.YIdx})
14: end for
15: return S̲_1, ..., S̲_n
```

Example. We illustrate the computation of the $\underline{S}$-values for the running example. The table below is self-explanatory, we only want to point out that the position of the iterator for the respective SCC is marked by underlining

the corresponding j-index:

i	$x_{\sigma(i)}$	relevant SCC	$D_{\sigma(i)}$	E_{YIdx}	$\underline{S}_{\sigma(i)}$
1	x_2	$\langle\underline{1}, 4, 5\rangle$	$[1;13]$	$[2;4]$	**2**
2	x_4	$\langle\underline{1}, 4, 5\rangle$	$[3;17]$	$[2;4]$	3
3	x_5	$\langle\underline{2}, 3\rangle$	$[5;6]$	$[4;7]$	5
4	x_1	$\langle\underline{2}, 3\rangle$	$[7;10]$	$[4;7]$	7
5	x_3	$\langle 1, \underline{4}, 5\rangle$	$[13;15]$	$[12;19]$	13

As we can see there is not much narrowing, only the lower endpoint of the domain of X_2 increases. One can check that the upper endpoints are not changed at all. So the narrowed X-domains of our running example are $S_1 = [7;10], S_2 = [\mathbf{2};13], S_3 = [13;15], S_4 = [3;17], S_5 = [5;6]$.

We describe how to narrow the domains of the Y-variables. In the following lemma we show that the upper endpoints of the narrowed domains can be easily read off given the matching computed by Glover's algorithm:

Lemma 3.4 (Narrowing of Y-domains) *Let $T_1, \ldots, T_n$ denote the projections of $\mathcal{S}$ onto the last n components. And let ϕ be the bijection computed by Algorithm 3.1 (Glover). Then $\overline{T}_j = \min(\overline{E}_j, \overline{D}_{\phi(j)})$ for $j = 1, \ldots, n$.*

Proof. By the choice of ϕ, we have $E_j \cap D_{\phi(j)} \neq \emptyset$ for all j. So let $\tau_j := \max(E_j \cap D_{\phi(j)}) = \min(\overline{E}_j, \overline{D}_{\phi(j)})$ for $j = 1, \ldots, n$. We will prove that $\mathcal{S}$ contains the tuple $(\tau_{\phi^{-1}(1)}, \ldots, \tau_{\phi^{-1}(n)}, \tau_1, \ldots, \tau_n)$. Clearly, $\tau_{\phi^{-1}(i)} \in D_i$ for all i, and $\tau_j \in E_j$ for all j. So what remains to show is $\tau_1 \leq \ldots \leq \tau_n$. Suppose otherwise, i.e. $\tau_j < \tau_{j-1}$ for some j. Since $\overline{E}_j \geq \overline{E}_{j-1}$ by the assumption of normalization, this implies $\overline{D}_{\phi(j)} = \tau_j < \tau_{j-1} \leq \overline{D}_{\phi(j-1)}$. And hence, if $x_{\phi(j)}$ had been a candidate for y_{j-1} in Glover's algorithm, it would have been preferred to $x_{\phi(j-1)}$. So it was not a candidate, although it was free, which implies $\overline{E}_{j-1} < \underline{D}_{\phi(j)}$. Thus we get $\overline{E}_{j-1} < \underline{D}_{\phi(j)} \leq \overline{D}_{\phi(j)} < \tau_{j-1} \leq \overline{E}_{j-1}$, a contradiction.

Consider now a tuple $t = (d_1, \ldots, d_n, e_1, \ldots, e_n) \in \mathcal{S}$. What remains to show is that $e_j \leq \tau_j$ for all j. Assume this is wrong, i.e. there is an index k with $e_k > \overline{D}_{\phi(k)}$. For $j = k, \ldots, n$ we replace the lower endpoint of E_j by $\max(\underline{E}_j, e_k)$, which gives us a new set of normalized Y-domains and induces an intersection graph G'. Clearly, t is a solution of the new constraint, and hence G' contains a perfect matching.
Suppose we run Glover's algorithm on the new domains. As $E_1, \ldots, E_{k-1}$ and the upper endpoint of E_k are the same as before, the algorithm will perform exactly the same computation until it extracts $\phi(k)$ from the priority queue in iteration k. Then it cannot match y_k with $x_{\phi(k)}$ again, and it will report that no perfect matching exists in G', a contradiction. □

In order to determine $\underline{T}_1, \ldots, \underline{T}_n$, we have to compute a new matching ϕ', which is obtained as follows. We match $y_n, \ldots, y_1$ in that order. When we match y_j, we choose among all candidates in $N(y_j) \setminus \{x_{\phi'(n)}, \ldots, x_{\phi'(n-1)}\}$ the node x_i such that $\underline{D}_i$ is maximal and set $\phi'(j) = i$. Then we get $\underline{T}_j = \max(\underline{E}_j, \underline{D}_{\phi'(j)})$ for $j = 1, \ldots, n$.

Example. We complete our running example by computing the narrowed Y-domains. The following table shows the function ϕ' obtained with the "reverse Glover" algorithm:

j	1	2	3	4	5
$\phi'(j)$	2	5	1	4	3

And we get $T_1 = [2; 4], T_2 = [\mathbf{5}; \mathbf{6}], T_3 = [\mathbf{7}; \mathbf{10}], T_4 = [12; \mathbf{15}], T_5 = [14; \mathbf{17}]$. (The endpoints which have been adjusted are typeset in bold.)

We give a summary of the full algorithm:

1. Sort the domains of the X-variables according to their lower and upper endpoints.

2. Normalize the domains of the Y-variables.

3. Compute the matchings ϕ and ϕ' with Glover's algorithm.

4. Compute the strongly connected components of the oriented intersection graph.

5. Narrow the domains of the variables.

Except for the first step, all steps take linear time. Thus the complexity of the whole propagation algorithm is asymptotically the same as for sorting the lower and upper endpoints of the X-domains. This is $O(n \log n)$ in general, but is $O(n)$ if interval endpoints are integers drawn from a range of size $O(n^k)$ for some fixed k.

Our algorithm is optimal in all models of sorting: Bleuzen-Guernalec and Colmerauer [BGC00] observed that a propagation algorithm A for the *Sortedness*-constraint which achieves bound-consistency can be used for sorting n elements $d_1, \ldots, d_n$ of the universe U in time $O(n)$ plus the running time of A. This can be done as follows. We compute the minimum value $\underline{d}$ and the maximum value $\overline{d}$ of the n elements. Then we call A with the domains $D_i = [d_i; d_i]$ and $E_i = [\underline{d}; \overline{d}]$ for $i = 1, \ldots, n$. The algorithm will shrink the E-domains to singletons such that the element of E_i equals the i-th element of $sort(d_1, \ldots, d_n)$.

3.1.4 Comparison with related work

The *Sortedness*-constraint was introduced by Older et al. in [OSvE95], where they gave the application to the job-shop scheduling problem that we have discussed in Section 3.1.1. But they do not introduce *Sortedness* as a global constraint. They use a general Prolog sorting algorithm (namely quicksort) where they plug in constraint variables instead of ground terms. Thus the constraint is broken down into elementary constraints of the form $X \leq Y$.

Later Zhou [Zho97] used a variant of the *Sortedness*-constraint to solve some hard job-shop scheduling problems that had been open before. Zhou extended the argument list of the constraint by n extra variables that encode the sorting permutation. The constraint

$$\mathit{SortednessPerm}(X_1, \ldots, X_n; Y_1, \ldots, Y_n; P_1, \ldots, P_n)$$

is semantically equivalent to the following constraints:

$$Y_1 \leq \ldots \leq Y_n \ \wedge \ \mathit{Alldiff}(P_1, \ldots, P_n) \ \wedge \ \forall i \in [1..n] : X_i = Y_{P_i}$$

The advantage of this formulation is that Zhou can use a global propagation algorithm for the *Alldiff* constraint, the constraints of the form "$X_i = Y_{P_i}$" are still transformed into elementary constraints. Moreover, he can use the P-variables to guide the search process.

We have thought about dealing with the P-variables in our algorithm. Although the straightforward approach does not yield a bound-consistency algorithm, we have decided to discuss it briefly, for it achieves at least the same pruning as the formulation by Zhou. Denote the domains of the variables by $D_1, \ldots, D_n$, $E_1, \ldots, E_n$ and $F_1, \ldots, F_n$. We change the definition of the intersection graph G slightly in order to take into account the permutation variables: An edge $\{x_i, y_j\}$ is in G iff $D_i \cap Y_j \neq \emptyset$ and $j \in F_i$. In order to narrow the domains of the X- and the Y-variables, we would do the same as before. Recall that we compute for every node x_i the leftmost node y_l and the rightmost node y_r that can be matched with x_i. The narrowed domain of P_i is simply $[l..r]$. It is clear that this still yields a narrowing algorithm for the *SortednessPerm*-constraint, because every tuple in the relation of the constraint corresponds to a perfect matching.
Moreover, one can observe that Lemma 3.4 still holds, which means that our approach achieves bound-consistency on the Y-domains. But for the domains of the X- and the P-variables this is not the case, one can see that Lemma 3.1 breaks down. (In the proof we start with an initial matching and transform until it corresponds to a sorting permutation. This construction does not work anymore.)

We make another observation: A perfect matching $\{\{x_i, y_{\pi_i}\} \mid i \in [1..n]\}$ in G corresponds to the tuple $(\pi_1, \ldots, \pi_n)$ in the relation of $\mathit{Alldiff}(P_1, \ldots, P_n)$. Thus applying a bound-consistency propagation algorithm for the *Alldiff*-constraint to the domains of $P_1, \ldots, P_n$ would not give more propagation than our algorithm.

Let us look at an example where our algorithm does not achieve bound-consistency on the domains of the permutation variables. We consider a *SortednessPerm*-constraint of arity $3 \cdot 4$ with the following variable domains:

$$\begin{array}{llll} D_1 = [1;2] & D_2 = [1;1] & D_3 = [2;2] & D_4 = [1;2] \\ E_1 = [1;1] & E_2 = [1;2] & E_3 = [1;2] & E_4 = [2;2] \\ F_1 = [1;2] & F_2 = [1;3] & F_3 = [2;4] & F_4 = [3;4] \end{array}$$

One can verify that the domains $D_1, \ldots, D_4$ of the X-variables and the domains $E_1, \ldots, E_4$ of the Y-variables are bound-consistent. But there is no solution to the constraint where P_3 is assigned the value 2: Assume otherwise. If P_3 equals 2, then P_1 must be set to 1. And hence, P_2 must take the value 3. This cannot be, because we have $X_2 < X_3$ and $Y_{P_2} = Y_3 \geq Y_2 = Y_{P_3}$. But it is easy to see that all edges in the intersection graph can belong to a perfect matching.

Finally, we want to talk about the work of Bleuzen-Guernalec and Colmerauer [BGC00]. They describe the first propagation algorithm for the *Sortedness*-constraint as a single global constraint. In the previous work the constraint was always decomposed into elementary constraints. Although Bleuzen-Guernalec and Colmerauer do not use the language of graph theory at all in their paper, one can identify some similarities with our work. Their central objects are bijections between the indices of the input variables and the output variables. Clearly, there is a one-to-one correspondence between these bijections and the perfect matchings in the bipartite graphs, which we consider. (In fact, in our implementation of Glover's algorithm we encode the computed matching as a bijection.) In the sequel we will summarize their approach and express it in the language of graph theory which makes it – as we feel – easier to understand and facilitates the comparison with our approach. They achieve a running time of $O(n \log n)$, which is in some cases slightly worse than our result.

We consider the constraint $\mathit{Sortedness}(X_1, \ldots, X_n; Y_1, \ldots, Y_n)$ and the domains $D_1, \ldots, D_n$ and $E_1, \ldots, E_n$. Let G denote the corresponding intersection graph. Let us first discuss the narrowing of the upper endpoints of the Y-domains. Our way to do this is as follows (cf. Lemma 3.4): With Glover's algorithm we compute a certain matching $\{\{x_{\phi(j)}, y_j\} : 1 \leq j \leq n\}$ and then the narrowed endpoint of $\mathit{Dom}(Y_j)$ is simply $\min(\overline{E}_j, \overline{D}_{\phi(j)})$. Bleuzen-Guernalec and Colmerauer compute the bijection ϕ^{-1} (see Complexity 3 in

[BGC00])[4]. But in the remaining algorithm they use ϕ. For the narrowing step they apply the same rule as we do (see Theorem 2 in [BGC00]).

Now we deal with narrowing the lower endpoints of the X-domains. Our approach is based on identifying the edges of G that can belong to some perfect matching (see Corollary 3.1). Property 4 of [BGC00] expresses the same statement but with different words. In order to find the matchable edges, we decompose the oriented intersection graph $\vec{G}$ into strongly connected components as it is done in standard matching theory. Bleuzen-Guernalec and Colmerauer do something different. They decompose G in what we will call "zig-zag" components (they call them blocks of type I). A zig-zag component is a bipartite graph which contains the skeleton shown on the left-hand side in Figure 3.2. Formally, a *zig-zag graph* Z is a bipartite graph which has $2n$ nodes $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ and contains (at least) the edges $\{x_i, y_i\}$ for $i = 1, \ldots, n$ and $\{x_i, y_{i+1}\}$ for $i = 1, \ldots, n-1$. The crucial property of Z is that any edge $\{x_i, y_j\}$ with $j \leq i$ can belong to a perfect matching of Z (see Figure 3.2).

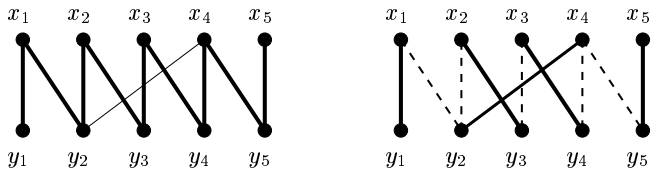


Figure 3.2: The left-hand side shows a zig-zag graph, which consists of the skeleton (bold edges) and one additional edge $e = \{x_4, y_2\}$. The bold edges on the right-hand side form a perfect matching containing e.

Let us consider the intersection graph G again. Assume that the layout of G is as follows: at the bottom we have the nodes $y_1, \ldots, y_n$ and above every node y_j we draw its mate $x_{\phi(j)}$. With this sorting of the x-nodes, G is in general not a zig-zag graph. But Bleuzen-Guernalec and Colmerauer were able to show that G can be decomposed into zig-zag components $Z_1, \ldots, Z_k$ such that the following holds: If there is an edge which connects an x-node in a component Z_l with a y-node in a different component Z_h, then $l < h$ (see Figure 3.3).[5] The meaning of this property becomes clear when we look at the oriented intersection graph $\vec{G}$ and its decomposition: The property ensures that edges between different zig-zag components are directed from top-left to bottom-right. So there can be no directed cycle that

[4] The bijection which is denoted by "φ" in [BGC00] corresponds to ϕ^{-1}.

[5] Their algorithm for computing this decomposition exploits certain properties of ϕ.

visits more than one component, which implies that nodes lying in different zig-zag components also belong to different strongly connected components of $\vec{G}$. Therefore an edge between different zig-zag components cannot belong to a perfect matching of G. And hence, it can be ignored in the narrowing process.

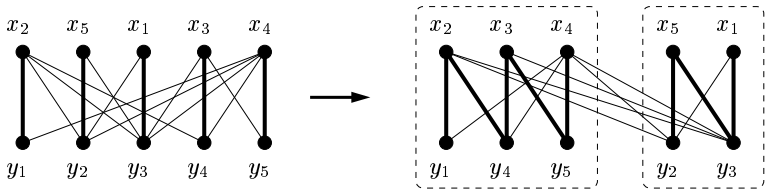


Figure 3.3: On the left-hand side we show the intersection graph G for the running example from the previous section. On the right-hand side G is decomposed into two zig-zag components.

We want to mention that the zig-zag components of G and the SCCs of $\vec{G}$ do not have to coincide as they do in the example in Figure 3.3, but in general a zig-zag component may span several SCCs.

With the zig-zag decomposition at hand, narrowing the lower endpoints of the X-domains is easy. For every node x_i its leftmost neighbour y_l in its zig-zag component is computed, and then we have $\underline{S}_i = \max(\underline{D}_i, \underline{E}_l)$.
We want to point out that the decomposition into zig-zag components which is based on ϕ can only be used to narrow the lower endpoints of the X-domains. For the upper endpoints one has to compute a second decomposition that is based on the bijection ϕ' (see page 55). In contrast to this, decomposing $\vec{G}$ into strongly connected components can be used to narrow both endpoints of the X-domains, which is an advantage of our approach.

3.2 The Alldiff-Constraint

The following section deals with the *Alldiff*-constraint. It arises in many applications, and several propagation algorithms have been developed for it, which achieve different degrees of consistency, see [vH01a] for a survey. After giving a formal definition of the constraint, we will describe our propagation algorithm. At the end of this section we will discuss related work.

3.2.1 Definition

We consider an *Alldiff*-constraint on the variables $X_1, \ldots, X_n$ with respective domains $D_1, \ldots, D_n$. For this constraint we allow only finite integer intervals as variable domains[6]. The *Alldiff*-relation $\mathcal{A} := \mathit{Rel}(\mathit{Alldiff}(X_1, \ldots, X_n))$ is defined as the set of all tuples $(d_1, \ldots, d_n)$ such that for $i, j \in [1..n]$ with $i \neq j$ we have $d_i \neq d_j$ and $d_i \in D_i$. Our task is to decide whether $\mathcal{A}$ is not empty, and, if so, to compute the smallest and the largest element in the projection of $\mathcal{A}$ on each of its n components.

3.2.2 Propagation algorithm

Our approach is based on the same ideas as Régin's arc-consistency algorithm [Rég94] for the *Alldiff*-constraint. He considers the *value* graph G which is an undirected bipartite graph defined as follows: For every variable X_i we have a node x_i, and for every value j that occurs in some domain D_i we have a node y_j. There is an edge $\{x_i, y_j\}$ iff $j \in D_i$. Let m denote the number of edges of G. Clearly, m is the sum of the cardinalities of the domains, and hence it does not depend on n. Since Régin's algorithm has running time $O(\sqrt{n}m)$ and we want to achieve a running time which only depends on n, we cannot use his algorithm directly. We will adapt his ideas and take advantage of the special structure of our graph. From the definition of the value graph, we can see that the neighbours of an x-node form an interval in the y-nodes. So it has the same nice property as the intersection graph from the previous section.

But in general, there are more values than variables, which means that there are more y-nodes than x-nodes. (If there are less values than variables, the constraint has no solution.) And hence, there is usually no perfect matching in the value graph. So we are interested in matchings which cover all x-nodes, but may leave some y-nodes free, we call them *x-perfect matchings*. There is a one-to-one correspondence between the solutions of the constraint and the x-perfect matchings of G. An x-perfect matching $\{\{x_i, y_{j_i}\} \mid i \in [1..n]\}$ corresponds to the tuple $(j_1, \ldots, j_n)$ in $\mathcal{A}$ and vice versa.

Example. We use the following running example:

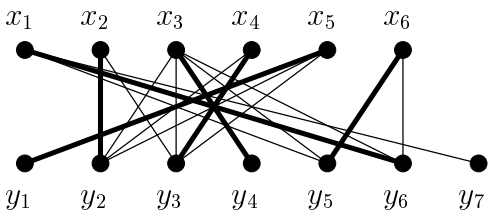
$$\mathit{Alldiff}(X_1, \ldots, X_6)$$

with the respective variable domains:

[6] What we actually need is a linearly ordered universe such that all intervals contain finitely many elements. And for an element u we must be able to compute its predecessor $u-1$ and its successor $u+1$ in constant time.

$$D_1 = [5..7]\ D_2 = [2..3]\ D_3 = [2..6]\ D_4 = [2..3]\ D_5 = [1..3]\ D_6 = [5..6]$$

On the right-hand side we show the value graph for our example. The bold edges indicate an x-perfect matching that corresponds to the tuple $(\overset{6}{X_1}, \overset{2}{X_2}, \overset{4}{X_3}, \overset{3}{X_4}, \overset{1}{X_5}, \overset{5}{X_6})$ in $\mathcal{A}$, the node y_7 is free.



In order to narrow the variable domains, we have to find all edges of G that can belong to some x-perfect matching. The characterization of the matchable edges is slightly more complicated than in the previous section, because we have to take into account the free nodes. The characterization given in the lemma below was already used by Régin [Rég94, Proposition 1]:

Lemma 3.5 *Let M be an x-perfect matching in G. Construct the* oriented value graph $\vec{G}$ *by directing all edges in G from the x-nodes to the y-nodes and adding the reverse edge for all edges in M. An edge $\{x_i, y_j\}$ belongs to some x-perfect matching of G iff one of the following holds:*

1. *x_i and y_j lie in the same strongly connected component of $\vec{G}$.*
2. *There is a path in $\vec{G}$ that starts with (x_i, y_j) and ends in a free y-node.*

Proof. See Lemma 2.4. □

So our first task is to compute an arbitrary x-perfect matching in G. Glover's algorithm, which was presented in the previous section, can be used to compute perfect matchings in convex bipartite graphs. Since G is also convex, we can modify the algorithm such that it computes x-perfect matchings. Let us recall the basic idea of the algorithm. We scan the y-nodes from left to right. In order to match y_j we look at all candidates, i.e. all free neighbours on the x-side, and choose the one where the corresponding domain ends earliest. When there is no candidate, we report failure and terminate. This is exactly the point which we have to change: When we have no candidate for y_j, then y_j remains free and we go to the next y-node. (Failure can only occur when we discover a free x-node where all neighbours on the y-side are matched.)

Before we develop an efficient implementation of the algorithm, we show its correctness:

Lemma 3.6 *If the value graph G contains an x-perfect matching, the modified Glover algorithm will construct one.*

Proof. The idea of the proof is as follows. First we present an algorithm whose correctness is obvious and then we argue that our modified algorithm above does the same computations.
Consider the following algorithm A:

1. Construct a new graph G' from G as follows. Let n' denote the number of y-nodes in G, and denote by h the greatest value that occurs in some variable domain. (We may assume $n' \geq n$, otherwise there is no x-perfect matching.) Add new nodes $x_{n+1}, \ldots, x_{n'+1}$ on the x-side and y_{h+1} on the y-side. Connect each of the nodes $x_{n+1}, \ldots, x_{n'}$ with every y-node by an edge, and add an edge between $x_{n'+1}$ and y_{h+1}.

2. Run Glover's original algorithm on G' and obtain (if possible) a perfect matching M'.

3. Construct M from M' by deleting all edges that are incident to a new node.

It is clear that G' contains a perfect matching iff G has an x-perfect one. And hence, the correctness of algorithm A follows directly from Lemma 3.2. Now we observe that for every added x-node the interval of y-neighbours ends strictly later than the interval of every original node, because every new x-node is connected to y_{h+1} but all old ones are not. Thus, A matches a node y_j with one of the new nodes only if there is no candidate among the original nodes. This is exactly the situation when the modified Glover algorithm (applied to G) declares y_j as free. So we conclude, that A and the modified Glover algorithm compute the same x-perfect matching M. □

Now we discuss the implementation of the modified algorithm (see Algorithm 3.4). We represent an x-perfect matching $\{\{x_{i_k}, y_{j_k}\} \mid k \in [1..n]\}$ by two arrays *XMate* and *YMate* such that $XMate[k] = i_k$ and $YMate[k] = j_k$ for all k. Thus free y-nodes are not stored explicitly. In the next paragraph we describe an $O(n \log n)$ implementation that uses a priority queue P. As it was the case for Algorithm 3.1, the queue can be replaced by an offline-min data structure. This yields a running time of $O(n)$, if one has a sorting of the variables according to upper and lower endpoints of their domains.

The algorithm scans the y-nodes from left to right and maintains a priority queue P which stores the indices of the possible matching candidates on the x-side. We assume that we have a sorting permutation σ such that $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(n)}$. The indices are inserted into P in the order $\sigma(1), \ldots, \sigma(n)$. Suppose that the algorithm is scanning y_j. After y_{j-1} has been processed, P contains the indices of all free nodes x_i with $\underline{D}_i \leq j-1$. Let $\sigma(s)$ be the next index that is to be inserted into P, i.e. we have $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(s-1)} <$

$j \leq \underline{D}_{\sigma(s)}$. If P is empty and $j < \underline{D}_{\sigma(s)}$, then all the nodes $y_j, \ldots, y_{\underline{D}_{\sigma(s)}-1}$ cannot be matched, because there is no candidate for them. Thus we can skip these nodes by assigning $\underline{D}_{\sigma(s)}$ to j, and we continue the scan with the corresponding y-node. This is important for achieving the desired running time of $O(n \log n)$.
The rest of the algorithm remains basically the same as in Algorithm 3.1. We add all indices i to P where $\underline{D}_i = j$. Then we extract the element i such that $\overline{D}_i$ is minimal. If $\overline{D}_i < j$, this means that all neighbours of x_i are matched and no x-perfect matching exists; we report this and terminate. Otherwise we match x_i and y_j and continue our scan.
Note that we have to verify after the scan that P is empty to make sure that every x-node has been matched (see line 18).

Algorithm 3.4 Finding a perfect matching in the value graph G

Function: GloverAlldiff($D_1, \ldots, D_n$, σ)
Require: σ is a permutation of $[1..n]$ s.th. $\underline{D}_{\sigma(1)} \leq \ldots \leq \underline{D}_{\sigma(n)}$.

```
 1: P ← [ ] // PQ stores x-indices [i_1, ..., i_k] s.th. D̄_{i_1} ≤ ... ≤ D̄_{i_k}
 2: s ← 1
 3: for k = 1 to n do
 4:    if P is empty then
 5:       j ← D_{σ(s)} // (possibly) skip free nodes
 6:    end if
 7:    while s ≤ n and D_{σ(s)} = j do
 8:       insert σ(s) into P
 9:       s ← s + 1
10:    end while
11:    extract the first element i from P, i.e. one with smallest D̄_i
12:    if D̄_i < j then
13:       report "no x-perfect matching in G" and terminate
14:    end if
15:    XMate[k] ← i; YMate[k] ← j
16:    j ← j + 1
17: end for
18: if P is empty then
19:    return XMate, YMate
20: else
21:    report "no x-perfect matching in G" and terminate
22: end if
```

We want to make some observations about Algorithm 3.4. If P is empty

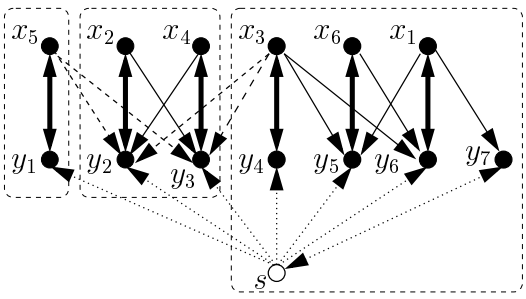
in line 4, then $s \leq n$, which means that the assignment to j in the next line is valid. This can be seen as follows. The number of insertions into P is $s-1$ and the number of extractions equals $k-1$. So if P is empty, we have $s = k \leq n$. Thus whenever P is empty in line 4, there will be an insertion in line 8, and hence P can never be empty in line 11.

Example. For our running example the algorithm computes the following x-perfect matching:

k	1	2	3	4	5	6
XMate	5	2	4	3	6	1
YMate	1	2	3	4	5	6

Assume that we have an x-perfect matching M in G. By Lemma 3.5, an edge $\{x_i, y_j\}$ can belong to some x-perfect matching iff x_i and y_j belong to the same SCC or y_j can reach a free y-node in $\vec{G}$. We will show that we can solve the problem with a single SCC computation if we extend $\vec{G}$ slightly. The idea is based on Régin's arc-consistency algorithm for the global cardinality constraint [Rég96], and it differs from the algorithm in the paper [MT00].[7] The *extended oriented value graph* $\vec{G}^\star$ is constructed by adding a special node s to $\vec{G}$, for every y-node we have an edge (s, y) and for every free y-node we also have the opposite edge (y, s). Before we prove in Lemma 3.7 that the SCC of s in $\vec{G}^\star$ consists exactly of those nodes that can reach a free y-node in $\vec{G}$, let us visualize the construction with the aid of our running example.

Example.
On the right-hand side we show the extended oriented value graph $\vec{G}^\star$ that corresponds to the previously computed matching. We have rearranged the x-nodes such that every x-node is located directly above its y-mate. The SCCs are indicated by the three dashed boxes.



The dashed xy-edges cannot belong to any x-perfect matching, because they cross components. Observe that all nodes in the SCC of s can reach the free node y_7.

Now we prove that $\vec{G}^\star$ has indeed the desired properties:

[7] This idea can also be used to simplify Régin's arc-consistency algorithm for the *Alldiff*-constraint [Rég94].

Lemma 3.7 *An edge $\{x_i, y_j\}$ can belong to some x-perfect matching in the value graph G iff x_i and y_j lie in the same SCC of the extended oriented value graph $\vec{G}^\star$.*

Proof. Suppose first that $\{x_i, y_j\}$ can belong to some x-perfect matching. By Lemma 3.5, this implies that x_i and y_j lie in the same SCC of $\vec{G}$ (case 1), or there is a path p in $\vec{G}$ that starts with (x_i, y_j) and ends in a free node y_f (case 2). For case 1 there is nothing to show, so assume that case 2 holds. Let y_m denote the matching mate of x_i. Then $\vec{G}^\star$ contains the cycle $(s, y_m) \circ (y_m, x_i) \circ p \circ (y_f, s)$. So x_i and y_j are in the same SCC in $\vec{G}^\star$.

For the other direction, suppose that x_i and y_j lie in the same SCC in $\vec{G}^\star$. Thus there is a cycle c in $\vec{G}^\star$ with (x_i, y_j) as its first edge. If c does not visit s, then x_i and y_j are in the same SCC of $\vec{G}$, and we are done. Otherwise, we decompose c as $c = p \circ (y_f, s) \circ q$, where p is a path from x_i to a free node y_f and p starts with the edge (x_i, y_j). Since p is a path in $\vec{G}$, this proves that $\{x_i, y_j\}$ can belong to some x-perfect matching by Lemma 3.5. □

In the sequel we discuss how to compute the SCCs of $\vec{G}^\star$ in linear time. It will turn out that we can use a slightly modified version of the algorithm that computes the SCCs of $\vec{G}$ (see Algorithm 3.2), which we have developed for the *Sortedness*-constraint. Both algorithms maintain a list *SCCs* of completed components and a stack *CS* of tentative components. In its k-th iteration Algorithm 3.5 computes the components of the graph $\vec{G}^\star_k$, which is a subgraph of $\vec{G}^\star$ and defined as follows. Let $YMate[n+1] := \max\{\overline{D}_1, \ldots, \overline{D}_n\} + 1$. For $k \in [0..n]$, $\vec{G}^\star_k$ is the subgraph induced by s, all nodes y_j with $j < YMate[k+1]$ and the matching mates of these y-nodes on the x-side. We want to point out that $\vec{G}^\star_n = \vec{G}^\star$ and that $\vec{G}^\star_0$ consists only of the node s, because the leftmost y-node is always matched. So $[s]$ is the only SCC of $\vec{G}^\star_0$, which explains the initialization steps in the algorithm.

The first part of the for-loop (lines 5 – 11) is the same as in the algorithm for *Sortedness*. It computes the SCCs of the subgraph of $\vec{G}^\star$ that is induced by the nodes of $\vec{G}^\star_{k-1}$ and the two nodes x_i, y_j, where $i = XMate[k]$ and $j = YMate[k]$. By $right\_x(C)$ and $right\_y(C)$ we denote the indices of the rightmost x- and y-node of the component C, where $right\_y([s])$ is defined as $-\infty$. Observe that s can reach any y-node in $\vec{G}^\star$, and hence, its component cannot be completed and popped from *CS* until the for-loop is finished. So it will remain the bottom-most component on *CS* all the time.

In the second part of the for-loop (lines 12 – 20) the algorithm deals with the free nodes between y_j and the next matched y-node, which has the index $YMate[k+1]$. This part was not necessary for the *Sortedness*-constraint,

Algorithm 3.5 Computing the SCCs of $\vec{G}^\star$

Function: ComputeSCCs($D_1, \ldots, D_n$, *XMate*, *YMate*)

```
 1: SCCs ← empty list
 2: CS ← [s]
 3: YMate[n + 1] ← max{D̄_1, ..., D̄_n} + 1
 4: for k=1 to n do
 5:    i ← XMate[k]; j ← YMate[k]
 6:    while |CS| > 1 and D̄_{right_x(top(CS))} < j do
 7:       pop C' from CS; append C' to SCCs
 8:    C ← [y_j, x_i]
 9:    while CS not empty and D_i ≤ right_y(top(CS)) do
10:       pop C' from CS; C ← C' ∘ C
11:    push C onto CS
12:    // deal with free nodes btw. y_j and the next matched y (if any)
13:    if j + 1 < YMate[k + 1] then
14:       C ← [y_{j+1}, ..., y_{YMate[k+1]−1}] // (store only y_{j+1} and y_{YMate[k+1]−1})
15:       while |CS| > 1 and D̄_{right_x(top(CS))} ≤ j do
16:          pop C' from CS; append C' to SCCs
17:       while CS not empty do
18:          pop C' from CS; C ← C' ∘ C
19:       push C onto CS
20:    end if
21: end for
22: while CS not empty do
23:    pop C' from CS; append C' to SCCs
24: return SCCs
```

because there were no free nodes. If free nodes exist, i.e. $j+1 < YMate[k+1]$, we put them in a tentative component C. Clearly, C will eventually be merged with the component of s, but before we come to the merging step, we can complete any component on CS that is not able to reach y_{j+1}. When the algorithm reaches line 17, the topmost component on CS can reach y_{j+1}. The bottom-most component contains s, and hence there is a path from s to y_{j+1} that visits at least one node in every component on CS (cf. invariant I3 below). By construction of $\vec{G}^\star$ there is an edge from the free node y_{j+1} to s. Thus C and all components on CS are merged into a single SCC of $\vec{G}^\star_k$.

It is clear that the algorithm runs in time $O(n)$. Concerning correctness we have essentially the same invariants as for the previous algorithm (see page 50). Whenever the algorithm reaches line 21 the following holds:

I1: Every component C in *SCCs* is an SCC of $\vec{G}$. In particular, C cannot reach any y_l with $l \geq \mathit{YMate}[k]$.

I2: $\mathit{SCCs} \cup \mathit{CS}$ contains the SCCs of the $\vec{G}_k^\star$.

I3: Let $\mathit{CS} = \langle C_1, C_2, \ldots \rangle$ (ordered from bottom to top). If $l < h$ then C_l can reach C_h but not vice versa and $\mathit{right\_y}(C_l) < \mathit{left\_y}(C_h)$.

I4: Let C denote a component in $\mathit{SCCs} \cup \mathit{CS}$ and let $i_\mathrm{r} = \mathit{right\_x}(C)$. Then for all $x_i \in C$ we have $\overline{D}_i \leq \overline{D}_{i_\mathrm{r}}$.

These invariants can be proven in the same way as for the *Sortedness*-constraint.

Narrowing the variable domains is now easy. Suppose we want to compute the narrowed domain S_i of a variable X_i. Then we determine the leftmost neighbour y_l and the rightmost neighbour y_r of x_i such that y_l and y_r belong to same SCC as x_i. By Lemma 3.7, we have $S_i = [l..r]$. The narrowing step can be done in time $O(n)$ with a similar algorithm as for the *Sortedness*-constraint (cf. Algorithm 3.3).

Example. We give the narrowed domains $S_1, \ldots, S_6$ for our running example: $S_1 = [5..7]$, $S_2 = [2..3]$, $S_3 = [\mathbf{4}..6]$, $S_4 = [2..3]$, $S_5 = [1..\mathbf{1}]$, $S_6 = [5..6]$.

We conclude this section with a summary of the full narrowing algorithm:

1. Sort the variable domains according to their lower and upper interval endpoints.
2. Compute an x-perfect matching with a modified version of Glover's algorithm.
3. Compute the strongly connected components of the extended oriented value graph.
4. Narrow the domains of the variables.

Except for the first step, all steps run in linear time. Thus the complexity of the whole algorithm is asymptotically the same as for the sorting step. This is $O(n \log n)$ in general, but it can go down to $O(n)$ if the endpoints of the domains are integers drawn from a range of size $O(n^k)$. We want to point an interesting special case: When we have n variables and all domains are contained in $[1..n]$, i.e. the variables encode a permutation of $[1..n]$, then the whole algorithm runs in linear time.

3.2.3 Comparison with related work

We conclude this section by comparing our algorithms with related work. Of course, one can decompose the constraint *Alldiff*$(X_1, \ldots, X_n)$ into $\binom{n}{2}$ binary constraints of the form $X_i \neq X_j$. As Puget [Pug98] points out this is suitable for simple problems, but leads to very poor propagation. This becomes obvious in the following example:

$$X_1, X_2, X_3 \in [1..2] \wedge X_1 \neq X_2 \wedge X_2 \neq X_3 \wedge X_3 \neq X_1$$

The decomposition into inequality constraints does not detect infeasibility.

Régin [Rég94] used exactly the same graph theoretic approach as we did in order to develop an arc-consistency algorithm for the case where the variable domains are not intervals but can be arbitrary sets of values. The algorithm works as follows. First it builds the variable-value graph G. Note that G is in general not convex, but it can be any bipartite graph. Then the matchable edges of G are identified, which requires the same steps as in our algorithm: Determine some x-perfect matching in G and orient G. Compute the SCCs of $\vec{G}$ and the edges that lie on a path to a free node. Let m denote the number of edges of G. The computation of the matching takes time $O(\sqrt{n} \cdot m)$, and the other steps can be done in $O(n + m)$. We want to point out that $m = \sum_{i=1}^{n} |Dom(X_i)|$. Thus m cannot be bounded above by a function in n. If all domains are for example the interval $[1..n]$, then we have $m = n^2$. This shows that Régin's algorithm, which is good for general domains, is not suitable for interval domains.

Leconte [Lec96] proposed an $O(nm)$ algorithm which achieves range-consistency (see Definition 2.6). It is a stronger notion of consistency than bound-consistency but weaker than arc-consistency. Leconte's algorithm is based on the Hall interval approach that we will sketch later.

In their paper about the *Sortedness*-constraint, Bleuzen-Guernalec and Colmerauer [BGC00] make an interesting observation. In order to encode a permutation on the variables $X_1, \ldots, X_n$, one can write the following constraint: *Sortedness*$(X_1, \ldots, X_n; 1, \ldots, n)$. And hence, their algorithm can be used to achieve bound-consistency for permutation constraints in time $O(n \log n)$. But their algorithm cannot be applied to the general *Alldiff*-constraint.

Puget [Pug98] was able to give a bound-consistency algorithm with a running time of $O(n \log n)$ for the general *Alldiff*-constraint. His approach is also based on matching theory but it is very different from ours, it does not rely on graphs or alternating paths at all. His reasoning is based directly on intervals. Consider the constraint *Alldiff*$(X_1, \ldots, X_n)$. For an interval I let *Vars*(I) denote the set of all variables such that $Dom(X_i) \subseteq I$. We

can make some observations: If $|Vars(I)| > |I|$, then the constraint has no solution, because every variable in $Vars(I)$ has to be assigned a different value in I, but we have more variables than values. We call such an interval *over-constrained*. Puget states as a corollary of Hall's Theorem [Hal35] that the constraint is solvable if and only if there is no over-constrained interval.

In order to derive his narrowing algorithm, Puget considers an interval H with $|Vars(H)| = H$, which he calls a *Hall interval*. In any solution of the constraint, every value in H is taken by exactly one variable in $Vars(H)$ and vice versa. Now let X_i denote a variable that is not in $Vars(H)$ and let $D_i = Dom(X_i)$. If $\underline{D_i} \in H$, then the constraint cannot be bound-consistent, for there can be no solution where X_i takes the value $\underline{D_i}$. In fact, one can move the lower endpoint of the domain of X_i to $\overline{H} + 1$ without loosing any solution to the constraint. A similar argument holds for the upper endpoint. Puget observed that this actually is a bound-consistency narrowing algorithm:

while there is an interval I with $|Vars(I)| \geq |I|$ **do**
  if $|Vars(I)| > |I|$ **then**
    report failure
  else
    for all $X_i \notin Vars(I)$ with $\underline{D_i} \in I$ **do**
      increase $\underline{D_i}$ to $\overline{I} + 1$
    for all $X_i \notin Vars(I)$ with $\overline{D_i} \in I$ **do**
      decrease $\overline{D_i}$ to $\underline{I} - 1$

In order to make this algorithm run in time $O(n \log n)$, Puget made some more observations. For example, it suffices to examine only intervals of the form $[\underline{D_i}..\overline{D_j}]$, where D_i and D_j are variable domains.

Example. Let us examine how the algorithm processes our running example with 6 variables and the domains

$$D_1 = [5..7]\ D_2 = [2..3]\ D_3 = [2..6]\ D_4 = [2..3]\ D_5 = [1..3]\ D_6 = [5..6]$$

We observe that $H = [2..3]$ is a hall interval, because its size is two and it contains the domains D_2 and D_4. So we can move the upper endpoint of X_5 from 3 to 1, and the lower endpoint of X_3 to 4. And we obtain the narrowed domains. We want to point out that there is not much correspondence between Hall intervals and the strongly connected components of $\vec{G}$. In the example above $H' = [1..3]$ is also a Hall interval, but the nodes corresponding to $Vars(H') = \{X_5, X_2, X_4\}$ belong to different SCCs.

Recently, López-Ortiz et al. [LOQTvB03] described an algorithm that is also based on the Hall interval approach and achieves bound-consistency in the same asymptotic running time as our algorithm.

Finally, we discuss briefly the global cardinality constraint (*GCC*), which was introduced by Régin [Rég96]. This constraint is a generalization of *Alldiff*. In addition to the n assignment variables $X_1, \ldots, X_n$, its input contains for each value v a lower and an upper capacity bound l_v and u_v. The constraint holds, if each value v is assigned to at least l_v and at most u_v variables. (If all lower bounds are zero and all upper bounds are one, we obtain the *Alldiff*-constraint.) Régin gave an arc-consistency algorithm for this constraint, which runs in time $O(n^{3/2} \cdot d)$, where d is the number of distinct values in the variable domains.

For the case, where all the variable domains are intervals Katriel and Thiel [KT03] were able to generalize the ideas of this chapter. They obtained a bound-consistency algorithm which runs in time $O(n + d)$ plus the time for sorting the interval endpoints. Interestingly, applying the Hall interval approach to this problem yields an algorithm with the same asymptotic running time (see [QvBLO+03]).

Chapter 4

Weighted Partial Alldiff

In this chapter we discuss a constraint called *WeightedPartialAlldiff* (abbreviated as *WPA*). The constraint $WPA(X_1, \ldots, X_n; undef; T; W)$ encodes a partial assignment to the variables $X_1, \ldots, X_n$, where undefined variables are represented by assigning the value *undef* to them. All variables which are defined must have pairwise distinct values. With every value v that occurs in the domain of some variable we associate a weight which is defined by the value-weight table T. The constraint states that W must be equal to the sum of the weights of the values that are assigned to the variables. We assume that the weight of the value *undef* is always equal to zero.
We give some examples, which will use the following value-weight table T:

value:	0	1	2	4	5	6
weight:	0	2	-1	7	-8	2

The constraint $WPA(4, 0, 1, 2, 0; 0; T; 8)$ holds, because no value except for $undef = 0$ is used more than once, and $weight(4) + weight(1) + weight(2) = 8$. But $WPA(5, 2, 5, 0; 0; T; 2)$ does not hold, for the value 5, which is different from the *undef*-value, is assigned twice. And $WPA(1, 6, 2; 0; T; 5)$ does not hold either, because of the weight condition: $weight(1) + weight(6) + weight(2) = 3 \neq 5$.
We discuss some possible scenarios where the constraint can be applied:

1. In the first scenario the weights are costs which one has to pay if a certain resource (i.e. a value) is used. Every resource can be used at most once. And the cost of a resource is independent from the consumer (i.e. the variable) to which it is assigned. In this case the constraint will probably be employed together with additional constraints that impose an upper bound on the weight variable W, because one wants to minimize the costs.

2. In the second scenario the weights describe profits which can be made by accepting certain offers. The offer can be accepted at most once and the profit does not depend on the acceptor. In these circumstances one is interested in maximizing the total profit. So it is likely that *WPA* is used with additional constraints that impose lower bounds on the weight variable.

3. In the previous two scenarios the constraint was used for optimization purposes. We will now deal with an application to an over-constraint problem. Suppose the constraint model of a problem uses – among other constraints – an *Alldiff*-constraint. And assume further that there is no solution where all variables take pairwise different values. So one may want to relax the problem a little bit, allowing some variables to be "undefined". By setting the weights of all values different from *undef* equal to 1, the number of defined variables can be controlled via the weight variable W.

The work presented in this chapter is partly based on joint work with Nicolas Beldiceanu and Mats Carlsson [BCT02]. The chapter is organized as follows. First we give a formal definition of the constraint, then we derive a propagation algorithm, and finally we discuss some related work.

4.1 Definition

We consider the constraint $WPA(X_1, \ldots, X_n; undef; T; W)$. The domains of the assignment variables $X_1, \ldots, X_n$ can be arbitrary finite sets. Let D denote the union of all variable domains. We require that T is a set of pairs which contains for every value $v \in D$ exactly one pair in which the first component is equal to v. The second component must be a number, which we will denote by $weight(v)$. The parameter W is a number variable. Its domain must be an interval, and we assume that the number type matches the type of the weights. Moreover, we suppose that the arithmetic operations "+" and "−" as well as comparisons can be done in constant time.

Now we can define $Rel(WPA(X_1, \ldots, X_n; undef; T; W))$ to be the set of all $(n+1)$-tuples $(v_1, \ldots, v_n, w)$ with the following properties: We have $w \in Dom(W)$ and $v_i \in Dom(X_i)$, for $i = 1, \ldots, n$. Moreover, for $1 \leq i < j \leq n$, we have $v_i \neq v_j$ or $v_i = v_j = undef$. And finally, $w = \sum_{i=1}^{n} weight(v_i)$.

4.2 Propagation algorithm

In this section we develop a propagation algorithm for the *WPA*-constraint. The worst-case running time will be $O((n+p)m)$, where m is the sum of the cardinalities of the domains of $X_1, \ldots, X_n$ and p is the number of pruned variable-value pairs. But for the three scenarios given in the introduction, we will be able to upper bound the running time by $O(nm)$, and we will show that we achieve arc-consistency for the assignment variables. In the third scenario, where all weights are equal to 1, we will also have bound-consistency for the weight variable W. The algorithm integrates well into frameworks where incrementality is important, and its best case running time is $\Theta(m)$.

Our propagation algorithm computes the following quantities:

- We determine the minimum and the maximum weight that can be achieved. These values can be used to detect failure and to narrow the bounds of the weight variable W.
- We compute for every variable X_i and every value $v \in Dom(X_i)$ the minimum and the maximum weight ($w_{\min}$ and $w_{\max}$) which can be achieved, if X_i is fixed to v and all other domains remain the same. This can be used for pruning the domain of X_i: If $w_{\min} > \overline{W}$ or $w_{\max} < \underline{W}$, we can remove v from the domain of X_i.

We want to point out that it suffices to restrict our attention to maximum weights, because the respective minima can be obtained as follows: Multiply all weights by -1, compute the maxima, and multiply these by -1.

4.2.1 A connection to matching theory

In Section 3.2.2 about the *Alldiff*-constraint we have seen that variable assignments, where every variable takes a different value, correspond in a natural way to perfect matchings in the value graph G. Since our new problem deals with weighted assignments, we consider the *weighted value* graph G. It is an undirected bipartite graph. On one side we have a node for each assignment variable and on the other side we have a node for each value that occurs in the domain of some assignment variable. Thus we can identify variables and values with the corresponding nodes in G. In the sequel we will denote a variable node by *Var* and a value node by *val*. There is an edge $\{Var, val\}$ in G iff $val \in Dom(Var)$. We assign a weight to every node as follows: Every variable node *Var* gets weight zero, and the weight of a value node *val* is given by the value-weight table T.

The weight of a matching M in G is defined as the sum of the weights of all matched nodes. Note that this differs from standard matching theory where one assigns weights to the edges, but not to the nodes. As usual M is called a *maximum weight matching* if there is no matching M' in G with $weight(M') > weight(M)$. Since there may be non-positive weights, a maximum weight matching is in general not a maximum cardinality matching.

In order to find the connection between the solutions of the constraint and weighted matchings in G, we make one observation. We can partition the variables in the constraint in two sets depending on whether the respective variable domain contains the value *undef*. If $undef \notin Dom(Var)$, then we should require that *Var* must be matched in G, otherwise it can be matched, but it does not have to be. This motivates the following definition: Let S denote a set of nodes and M a matching in G, we call M an *S-matching* if all nodes in S are matched in M.

Now we can make the connection between solutions and matchings:

Lemma 4.1 *Consider the constraint* $WPA(X_1, \ldots, X_n; undef; T; W)$, *and let* G *be its weighted value graph. Let* $\mathcal{R}$ *denote* $\{X_i \mid undef \notin Dom(X_i)\}$. *Any solution* $(v_1, \ldots, v_n, w)$ *of the constraint corresponds to an* $\mathcal{R}$*-matching in* G *of weight* w. *And for any* $\mathcal{R}$*-matching* M *of weight* w *with* $w \in Dom(W)$, *we can construct a solution of weight* w.

Proof. Let $(v_1, \ldots, v_n, w)$ be a solution. We construct an $\mathcal{R}$-matching M of weight w as follows: For $i = 1, \ldots, n$, we put the edge $\{X_i, v_i\}$ into M, if $v_i \neq undef$. Since all values different from *undef* are pairwise distinct, M is a matching, and it covers all nodes in $\mathcal{R}$. As $weight(undef) = 0$, the weight of M is equal to w.

For the other direction we consider an $\mathcal{R}$-matching M in G. Let w denote the weight of M, and for $i = 1, \ldots, n$ define v_i as follows: If X_i is matched in M, then v_i is the matching mate of X_i, otherwise set $v_i = undef$. It is easy to verify that $(v_1, \ldots, v_n, w)$ is a solution. □

So we have transformed our problem into a matching problem. We want to compute a maximum weight $\mathcal{R}$-matching in G efficiently. There are well known algorithms that compute a maximum weight matching in a bipartite network where the edges have weights, but the nodes do not. These algorithms run in time $O(n(m + k \log k))$ (see [AMO93]), where n is the number of variable nodes, k is the number of nodes and m is the number of edges in G. We could easily transform our weighted value graph into such a network – we would only have to assign to every edge the weight of its incident value node – but we develop a new algorithm, which takes advantage of the special properties of our weights to be more efficient. The new algorithm has a

slightly better asymptotic running time of $O(nm)$, and it uses only simple data structures like arrays and lists, while the algorithms for networks with edge weights employ elaborate data structures like Fibonacci heaps.

A very important property of our algorithm is that we can run it without constructing G explicitly. In addition, it is incremental, which means that one can start with an arbitrary matching M_{init}. If M_{init} has nearly optimal weight the running time may be much better: It can go down to $\Theta(m)$.

4.2.2 Constructing an $\mathcal{R}$-matching

This section deals with the following problem: We have an arbitrary matching M in the weighted value graph G. Let $\mathcal{R}$ denote the set of all variables which are not adjacent to the value *undef*. How can we construct an $\mathcal{R}$-matching, i.e. a matching which covers all variables in $\mathcal{R}$? We will not care about weights in this section, we defer this problem to the next section. We need two important facts from standard matching theory (see Lemma 2.2):

- If M' is another matching in G, then $M \oplus M'$ is a collection of node-disjoint alternating paths and cycles.
- If p is an alternating path or cycle with respect to M, then $M \oplus p$ is a matching again.

Suppose M' is an $\mathcal{R}$-matching, but M is not. So there is a node $x \in \mathcal{R}$ which is free in M, but matched in M'. And hence, $M \oplus M'$ contains an (acyclic) alternating path p that starts in x and ends in some node y. As we can see in Figure 4.1, there are two possibilities for y. Either y is a value which is free in M; or y is a variable that is matched in M and free in M', which implies $y \notin \mathcal{R}$.

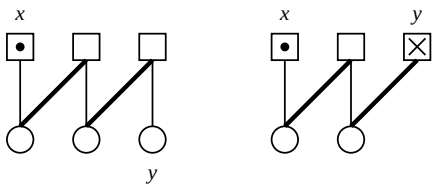


Figure 4.1: The two possibilities for the alternating path starting in $x \in \mathcal{R}$. (Variables are drawn as squares and values are drawn as circles. The dot in the square of x indicates that the variable is in $\mathcal{R}$, and the cross in the square of y means that y is not in $\mathcal{R}$.)

Without loss of generality we may assume that the value *undef* is free in M, and in this case we say that M is an *undef-free* matching. So if y is a

variable not in $\mathcal{R}$, then we can replace p by $p' = p \circ \{y, \mathit{undef}\}$, and obtain an alternating path wrt. M which starts in x and ends in a free value.

This motivates the following definition:

Definition 4.1 *Let M denote an undef-free matching in G. An alternating path p wrt. M is called $\mathcal{R}$-augmenting if one end node of p is a free variable in $\mathcal{R}$ and the other end node is a free value.*

By the discussion above we can construct an $\mathcal{R}$-matching from an arbitrary matching M as follows. As long as M is not an $\mathcal{R}$-matching, we ensure that the value *undef* is free, look for an $\mathcal{R}$-augmenting path p wrt. M and replace M by $M \oplus p$.

4.2.3 Weight-augmenting paths

In this section, we assume that we have an $\mathcal{R}$-matching M, and we solve the problem of maximizing its weight. Suppose there is an $\mathcal{R}$-matching M' with $\mathit{weight}(M') > \mathit{weight}(M)$. Then $M \oplus M'$ is a set of alternating paths and cycles. Since any node $\mathit{Var} \in \mathcal{R}$ is matched in both M and M', the degree of Var in $M \oplus M'$ is either 0 or 2. And hence, any (acyclic) alternating path in this set can neither start nor end in Var.

Let us consider an alternating path or cycle p in $M \oplus M'$ and compare the weights of M and $M \oplus p$. If p is a cycle, then M and $M \oplus p$ match exactly the same nodes, and hence they have the same weight (see case 1 of Figure 4.2).
Otherwise, denote by x and y the first and the last node of the (acyclic) path p. If both x and y are variable nodes, M and $M \oplus p$ match the same value nodes, and then they also have the same weight, because variable nodes have weight zero (cf. case 2 of Figure 4.2).
Now we come to the case that both x and y are value nodes. Then one of the two nodes (let us say x) is matched and the other one (in our case y) is free. And we have $\mathit{weight}(M \oplus p) = \mathit{weight}(M) - \mathit{weight}(x) + \mathit{weight}(y)$ (see case 3 of Figure 4.2).
Finally, we have the case that x is a variable node and y is a value node. Either both nodes are matched in M and we have $\mathit{weight}(M \oplus p) = \mathit{weight}(M) - \mathit{weight}(y)$, because $\mathit{weight}(x) = 0$. Or both nodes are free, and we get $\mathit{weight}(M \oplus p) = \mathit{weight}(M) + \mathit{weight}(y)$ (cf. case 4 in Figure 4.2).

So we have three possible constellations for p such that $\mathit{weight}(M \oplus p) > \mathit{weight}(M)$:

1. One end node is a matched value x and the other one is a free value y with $\mathit{weight}(y) > \mathit{weight}(x)$.

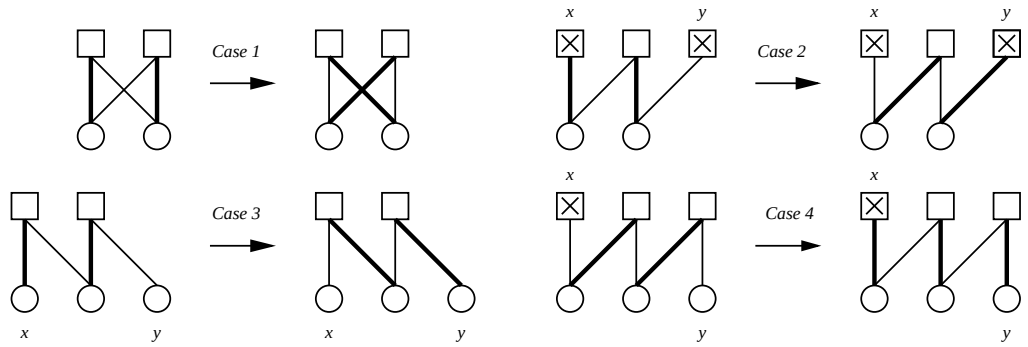


Figure 4.2: Comparing the weights of the matchings M and $M \oplus p$. (Variables are drawn as squares and values are drawn as circles.)

2. One end node is a free variable x (not in $\mathcal{R}$) and the other one is a free value y with $weight(y) > 0 = weight(x)$.

3. One end node is a matched variable x (not in $\mathcal{R}$) and the other end node is a matched value y with $weight(y) < 0 = weight(x)$.

We can simplify things and eliminate the third case by a similar observation as in the previous section: Let us suppose that M is an *undef*-free matching. So if x is a matched variable which does not belong to $\mathcal{R}$ and y is a matched value, then we can replace p by $p' = \{undef, x\} \circ p$. And we obtain an alternating path such that $weight(M \oplus p') = weight(M \oplus p)$. We observe that the first case applies to p'.

Now we can state the definition of a weight-augmenting path:

Definition 4.2 *Let M be an undef-free $\mathcal{R}$-matching in G. An alternating path p wrt. M is said to be* weight-augmenting *if one of the first two cases from above holds for p.*

So if we want to build a maximum weight $\mathcal{R}$-matching from an arbitrary $\mathcal{R}$-matching M, we proceed as follows: We ensure that M is *undef*-free, we search a weight-augmenting path p wrt. M, and replace M by $M \oplus p$. We repeat this process until there is no weight-augmenting path anymore.

4.2.4 The oriented value graph

Constructing a maximum weight $\mathcal{R}$-matching requires to search repeatedly for $\mathcal{R}$-augmenting and weight-augmenting paths in the value graph $G = (V, E)$ with respect to the current matching M. In order to make this search

easier, we introduce the *oriented value graph* $\vec{G}_M = (V, \vec{E})$. The definition is the same as in the propagation algorithm for the *Sortedness*- and the *Alldiff*-constraint. We direct all the edges in E from the variable nodes to the value nodes, and for every edge $\{Var, val\}$ in M we add the edge (val, Var) which is directed in the opposite direction. An example is shown in Figure 4.3.

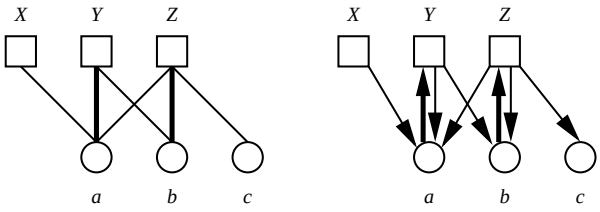


Figure 4.3: The left-hand side shows a value graph G and a matching M (bold edges). On the right-hand side the corresponding graph $\vec{G}_M$ is drawn.

Every simple path $\vec{p}$ in $\vec{G}_M$ from a node x to a different node y corresponds to a path p in G from x to y which alternately uses edges in M and in $E \setminus M$. Of course, there is also a correspondence in the opposite direction, so we can identify the alternating paths in G with certain simple directed paths in $\vec{G}_M$. In the figure above, the alternating path $p = [Y, a, Z, b]$ in G corresponds to the path $\vec{p} = [b, Z, a, Y]$ in $\vec{G}_M$. Thus the directed path visits exactly the same nodes as the undirected one, but maybe in reverse order.

We are interested now, how $\mathcal{R}$-augmenting and weight-augmenting paths translate to $\vec{G}_M$. We begin with an $\mathcal{R}$-augmenting path p (see left-hand side of Figure 4.4). The start node x of $\vec{p}$ is a free variable $x \in \mathcal{R}$, which will become matched by the augmentation. And the end node y of $\vec{p}$ is a free value.

If $\vec{p}$ is the translation of a weight-augmenting path, then the start node x is either a free variable or a matched value, and the end node y is a free value with $weight(y) > weight(x)$. So there are two possibilities, which are shown on the right-hand side of Figure 4.4.

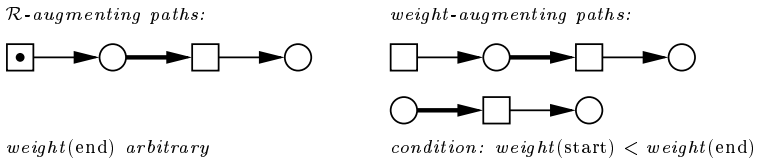


Figure 4.4: Possible directed augmenting paths.

We observe that $\mathcal{R}$-augmenting and weight-augmenting paths are very

similar, in particular the possible end nodes are always free values. Moreover, for both kinds of augmenting paths we have $\mathit{weight}(M \oplus p) = \mathit{weight}(M) - \mathit{weight}(x) + \mathit{weight}(y)$. So if we look for an augmenting path that starts in a certain node x, we should choose a path $\vec{p}$ where the end node y has maximum weight, for this gives us the maximum weight increase (or the minimum weight decrease if we are searching an $\mathcal{R}$-augmenting path and all possible end nodes have negative weight). We call $\vec{p}$ an *optimal augmenting path for* x. And we define $\mathit{weight}(y)$ to be the potential of x. The formal definition is as follows:

Definition 4.3 (potential π) *Let M be a matching and x be a node in the value graph G. Denote by $R_M(x)$ the set of all nodes y such that there is a path from x to y in $\vec{G}_M$. Then the* potential of x with respect to M *is defined as* $\pi_M(x) := \sup\{\mathit{weight}(y) \mid y \in R_M(x) \wedge y \text{ is free wrt. } M\}$.[1]

We want to make some remarks. If we have $\pi_M(u) = -\infty$ for a node u, then there can be no $\mathcal{R}$-augmenting path that starts in u. If $\pi_M(u) \leq \mathit{weight}(u)$, then u cannot be the source of a weight-augmenting path. In the lemma below we state another reason for using optimal paths to augment a matching: This guarantees that the potential of all nodes in the graph can only decrease. In other words, this cannot introduce new sources for augmenting paths, but it can remove existing ones.

Lemma 4.2 *Let $\vec{p}$ be an $\mathcal{R}$-augmenting or weight-augmenting path in $\vec{G}_M$, and let x and y be the start and the end node, respectively. If p is optimal for x (i.e. $\pi_M(x) = \mathit{weight}(y)$), then we have $\pi_{M\oplus p}(u) \leq \pi_M(u)$ for every node u of G.*

Proof. Let $M' = M \oplus p$, and consider a node u with $\pi_{M'}(u) > -\infty$. Thus there is a path $\vec{q}$ in $\vec{G}_{M'}$ from u to a node v such that v is a free value wrt. M' and $\mathit{weight}(v) = \pi_{M'}(u)$. If $\vec{p}$ and $\vec{q}$ are node disjoint, then $\vec{q}$ is a path in $\vec{G}_M$, and the claim clearly holds.

Otherwise the two paths have a common node. Let w be the first node on $\vec{q}$ that also lies on $\vec{p}$. The prefix of $\vec{q}$ from u to w is a path in $\vec{G}_M$, and hence there is a path from u to y in $\vec{G}_M$. This implies $\pi_M(u) \geq \mathit{weight}(y)$.

In order to prove the claim we will show $\mathit{weight}(v) \leq \mathit{weight}(y)$. First we convince ourselves that $v \in R_M(x)$. Since w is visited by $\vec{p}$, we have $w \in R_M(x)$. As all nodes on $\vec{p}$ are in $R_M(x)$, there is no edge in $\vec{G}_{M'}$ that is directed from a node in $R_M(x)$ to a node outside of $R_M(x)$. Thus there is no

[1] Here "$\sup S$" denotes the supremum of S, which is equal to $\max(S \cup \{-\infty\})$ if S is finite.

path in $\vec{G}_{M'}$ which leads from w to a node outside of $R_M(x)$. In particular, $v \in R_M(x)$.

If $x \neq v \neq y$, then v is free with respect to both matchings, and hence $weight(y) = \pi_M(x) \geq weight(v)$. For $v = y$, there is nothing to show. So the only remaining case is $v = x$. Since $v \notin \mathcal{R}$ and the start node of an $\mathcal{R}$-augmenting path is always a variable in $\mathcal{R}$, we conclude that $\vec{p}$ is weight-augmenting. This implies $weight(v) = weight(x) < weight(y)$. □

4.2.5 Computation of a maximum weight matching

In this section we discuss an algorithm for computing a maximum weight $\mathcal{R}$-matching in the value graph G. It takes as input an arbitrary matching M_{init}, which may be empty. The algorithm works as sketched above: It sets $M = M_{init}$, and as long as there is an augmenting path wrt. M, it selects an optimal augmenting path p and replaces M by $M \oplus p$. The algorithm operates in two phases. In the first phase it searches for $\mathcal{R}$-augmenting paths and turns M into an $\mathcal{R}$-matching. In the second phase it constructs a maximum weight $\mathcal{R}$-matching with the aid of weight-augmenting paths.

Recalling the similarity of $\mathcal{R}$-augmenting and weight-augmenting paths in $\vec{G}_M$ (see Figure 4.4), it is no surprise that we can use a single function for finding both types of paths. We use breadth-first-search as strategy, because it finds shortest augmenting paths. The function `BFS(`x`,`w`,`M`,`$mark$`)` (see Algorithm 4.1) takes as input four parameters: a start node x, a weight threshold w, a matching M and a node array *mark*. The function searches for an optimal augmenting path p for x and returns a boolean value indicating whether such a path exists. If so, it replaces M by $M \oplus p$. An alternating path p is only accepted to be augmenting if the weight of its end node is greater than the threshold w. So when we look for an $\mathcal{R}$-augmenting path, we set $w = -\infty$; and for finding weight-augmenting paths we choose $w = weight(x)$. The array *mark* stores for every value a flag which can be *reached* or *unreached*. More details will be given later, but for now we may assume that if a node u is marked *reached*, then its potential $\pi_M(u)$ is not greater than w, and hence it does not have to be explored.

`BFS` grows a tree of alternating paths with x at the root, for every other node u in the tree we store its father in an array *father*, and we set $father[x] = none$. Moreover, we maintain a list *tree_values* that contains all values in the tree. We use a first-in, first-out queue Q to store every variable that has been discovered but not explored yet. At the beginning, we initialize Q with x. Since we are looking for an optimal augmenting path, we maintain a variable *opt_end*. As long as we have not found an augmenting path, its value is

none. Afterwards it stores the end node of the best augmenting path that we have discovered so far.

We describe the main loop of the function (starting in line 8). As long as Q is not empty, we extract the first variable node Var from Q and scan its outgoing edges, i.e. we enumerate the domain of Var. We explore every value $val \in Dom(Var)$ which is not marked reached. After marking val as reached, we add it to the BFS tree by updating the data structures *father* and *tree_values*. If val is matched, we append its mate $M[val]$ to Q and add it to the tree. (As we have visited val for the first time, the same holds for $M[val]$, which explains why we do not store marks for the variables.) If val is free in M, then we may have found an augmenting path from x to val. This is the case iff $weight(val) > w$. Whenever we find a new augmenting path, we update *opt_end*, if necessary. If the main loop terminates with $opt\_end \neq none$, we have an optimal augmenting path p from x to *opt_end* and we replace M by $M \oplus p$.

It is now time to discuss the maintenance of the *mark*-array in more detail. We will see that it is not always necessary to reset the marks of the values in *tree_values* when a `BFS`-call terminates. Before the first call to `BFS` all marks are initialized to *unreached*, and we will make sure in the main algorithm that the weight threshold can only increase from call to call. Under these circumstances we can maintain the following invariant: Consider a `BFS` call with threshold w and let M' denote the matching after the call, then $\pi_{M'}(u) \leq w$ for all nodes u which are marked *reached* after the call. In order to establish the invariant when the function terminates we distinguish three cases:

- $num\_augmenting\_paths = 0$:
  Thus $\pi_M(u) \leq w$ for all nodes in the tree, and we do not have to reset any marks.

- $num\_augmenting\_paths = 1$:
  So before the augmentation, y is the only free value in $R_M(x)$ with weight greater than w. And hence, after the augmentation all free values in $R_M(x)$ have weight at most w (see also the proof of Lemma 4.2). Thus we do not have to reset the marks in this case either.

- $num\_augmenting\_paths > 1$:
  In this case we reset the marks of all variables in the tree (see line 25). Afterwards the invariant clearly holds again.

The implementation of the matching algorithm (see Algorithm 4.2) is now straightforward. It takes as input a matching M_{init} and the set $\mathcal{R}$ of

Algorithm 4.1 Searching an optimal augmenting path starting in x

Function: BFS(x,w,M,$mark$,$\mathcal{R}$)

1: $Q \leftarrow []$; $opt\_end \leftarrow none$; $num\_augmenting\_paths \leftarrow 0$
2: **if** x is a variable **then**
3: $Q \leftarrow [x]$; $father[x] \leftarrow none$; $tree\_values \leftarrow \emptyset$
4: **else if** $mark[x] = unreached$ **then**
5: $mark[x] \leftarrow reached$; $father[x] \leftarrow none$; $tree\_values \leftarrow \{x\}$
6: $Q \leftarrow [M[x]]$; $father[M[x]] \leftarrow x$
7: **end if**
8: **while** Q not empty **do**
9: extract first node Var from Q
10: **for all** values $val \in Dom(Var)$ with $mark[val] = unreached$ **do**
11: $mark[val] \leftarrow reached$; $father[val] \leftarrow Var$;
$tree\_values \leftarrow tree\_values \cup \{val\}$
12: **if** $M[val] \neq none$ **then**
13: append $M[val]$ to Q; $father[M[val]] \leftarrow val$
14: **else if** $weight(val) > w$ **then**
15: $num\_augmenting\_paths \leftarrow num\_augmenting\_paths + 1$
16: **if** $opt\_end = none$ or $weight(val) > weight(opt\_end)$ **then**
17: $opt\_end \leftarrow val$
18: **end if**
19: **end for**
20: **end while**
21: **if** $opt\_end \neq none$ **then**
22: augment M with path from x to opt_end (with the aid of $father$)
23: $M[undef] \leftarrow none$
24: **if** $num\_augmenting\_paths > 1$ **then**
25: **for all** $val \in tree\_values$ reset $mark[val]$ to $unreached$
26: return $true$
27: **else**
28: return $false$
29: **end if**

variables that are required to be matched. It starts with $M = M_{init}$, and in the first phase it tries to turn M into an $\mathcal{R}$-matching. We observe that once a variable from $\mathcal{R}$ is matched in the current matching M, it will remain matched till the end of the algorithm. So the source for an $\mathcal{R}$-augmenting path can only be a variable that is free in M_{init}. If the algorithm cannot find an $\mathcal{R}$-augmenting path for a free variable in $\mathcal{R}$, then it reports failure and terminates.

If the first phase succeeds, M is an $\mathcal{R}$-matching, and it will remain one till the end. In the second phase we maximize the weight of M. It will turn out that we only have to consider the values which are matched in M_{init} and the variables which are free in M_{init} and not in $\mathcal{R}$. Note that an end node of an augmenting path is always a free value. And hence, every value that is matched in M_{init} will remain matched until it is considered in the second phase, but the matching mate may change. A similar argument shows that variables which are free in M_{init} remain free until they are processed.

Since we have to make sure that the weight threshold does not decrease from one `BFS` call to the next, we split the second phase into three steps. First we consider all values with negative weights in weight increasing order. Then we process the variables, which all have weight zero. And finally, we deal with the values with non-negative weights in increasing order.

Algorithm 4.2 Computation of a maximum weight $\mathcal{R}$-matching in G

Function: ComputeMaxWeightRMatching(M_{init},$\mathcal{R}$)
Require: $M_{init}[val]$ stores for every value *val* its matching mate or the value *none* if *val* is free.

```
1: M ← M_init; M[undef] ← none
2: for all values val set mark[val] to unreached
3: // phase 1: turn M into an R-matching
4: for all variables Var ∈ R which are free in M_init do
5:    found ← BFS(Var, −∞, M, mark)
6:    if not found then report failure and terminate
7: // phase 2: maximize the weight of M
8: for all values val matched in M_init with weight < 0 in incr. order do
9:    BFS(val, weight(val), M, mark)
10: for all variables Var ∉ R which are free in M_init do
11:    BFS(Var, 0, M, mark)
12: for all values val matched in M_init with weight ≥ 0 in incr. order do
13:    BFS(val, weight(val), M, mark)
14: return M
```

We will prove the correctness of the algorithm. We have already explained

why BFS will always find an optimal augmenting path if one exists. Moreover, it is easy to see that the algorithm constructs an $\mathcal{R}$-matching if G contains one. What remains to show is that there is no weight-augmenting path when the algorithm terminates. This will be done in the proof of the following theorem.

Theorem 4.1 *When Algorithm 4.2 is applied to an arbitrary matching M_{init} and a set $\mathcal{R}$ of variables, it computes a maximum weight $\mathcal{R}$-matching M in the value graph G, if one exists. If no $\mathcal{R}$-matching exists, it reports failure.*

Proof. Suppose the statement is false, i.e. the algorithm terminates with an $\mathcal{R}$-matching M that does not have maximum weight. Then there exists a weight-augmenting path $\vec{p}$ in $\vec{G}_M$. Let x and y denote the start and the end node of $\vec{p}$, respectively. We have $\pi_M(x) \geq weight(y) > weight(x)$, we will derive a contradiction by showing $\pi_M(x) \leq weight(x)$.

Assume first that x is a value. Then it is matched in M (see right-hand side of Figure 4.4). We show that x is also matched in M_{init} and stays matched till the end of the algorithm. Suppose otherwise, i.e. at some point in time x is free in the current matching M'. As x has no outgoing edge in $G_{M'}$, we have $\pi_{M'}(x) = weight(x)$. By Lemma 4.2 this implies $\pi_M(x) \leq weight(x)$, a contradiction.
Since x is matched in M_{init}, it is considered as a start node for an augmenting path in the second phase (see lines 8 and 12), but none is found. (Otherwise x would be free after the augmentation.) Thus we can conclude $\pi_M(x) \leq weight(x)$, again a contradiction.

So x must be a variable not in $\mathcal{R}$, and x is free in M. This can only be the case if x is free in M_{init} and stays free till the end: Suppose otherwise, i.e. at some point in time x is matched in the current matching M' and becomes free as M' is augmented with a path q. If we view q as a directed path $\vec{q}$ in $\vec{G}_{M'}$, then $(x, undef)$ is the last edge of $\vec{q}$; let z denote the start node. Since q is chosen optimal for z, we have $\pi_{M'}(z) = 0 = weight(x)$. From $x \in R_{M'}(z)$ we conclude $\pi_{M'}(x) \leq \pi_{M'}(z)$. So we can infer $\pi_M(x) \leq weight(x)$, a contradiction.
Thus x is free in M_{init}, and hence it has been the start node of a BFS during the second phase (see line 10). As it has not become matched then, we can conclude again that $\pi_M(x) \leq 0 = weight(x)$, a contradiction. □

Example. It is now time to give an example which illustrates the algorithm. We have 5 variables $X_1, \ldots, X_5$ with the following domains: $Dom(X_1) = \{2\}$, $Dom(X_2) = \{0, 2, 3\}$, $Dom(X_3) = \{0, 1, 5, 6\}$, $Dom(X_4) = \{0, 4, 5, 6\}$ and $Dom(X_5) = \{-1, 4\}$. In order to make the descriptions easier, we define

the weight of each value to be equal to the value itself. And we assume $undef = 0$. Thus the variables X_1 and X_5 cannot be undefined, and hence $\mathcal{R} = \{X_1, X_5\}$. Let us suppose that we apply the algorithm to the initial matching $M_{init} = \{\{X_2, 2\}, \{X_4, 5\}, \{X_5, -1\}\}$. The corresponding oriented value graph is shown in Figure 4.5.

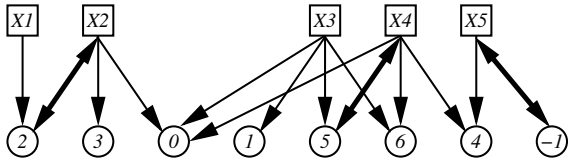


Figure 4.5: The oriented value graph $\vec{G}_{M_{init}}$ for the example.

We see that $\mathcal{R}$ contains only one variable (namely X_1) which is free in M_{init} so that the first phase consists of a single `BFS` call. We grow a tree with root X_1 (see Figure 4.6), and we discover two $\mathcal{R}$-augmenting paths, one ends in the value 3 and the other one in the value 0. Since the first path yields the bigger weight increase, we use this path to augment the matching and obtain the graph shown on the right-hand side of the figure. After that our current matching M is an $\mathcal{R}$-matching.

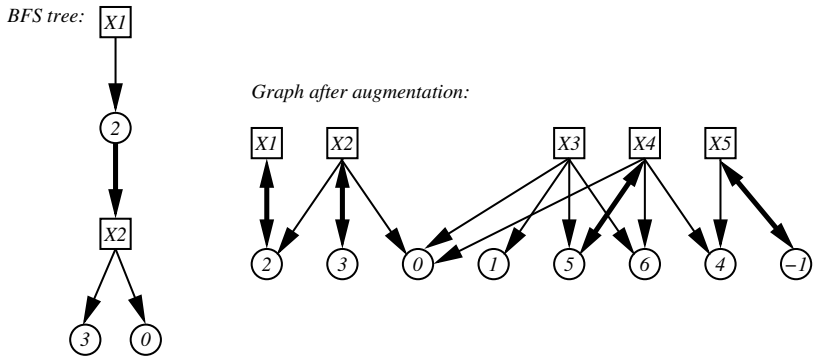


Figure 4.6: On the left-hand side the BFS tree with root X_1 is depicted. The right-hand side shows the oriented value graph after the augmentation.

Now we come to the second phase of the algorithm. First we build a BFS tree with the value -1 at the root (see Figure 4.7), because this value is matched in M_{init}. We find one weight-augmenting path ending in the value 4, and update the current matching M accordingly.

Finally, we have to process the variable X_3, for it is free in M_{init} and not contained in $\mathcal{R}$. The function `BFS` constructs the tree which is shown

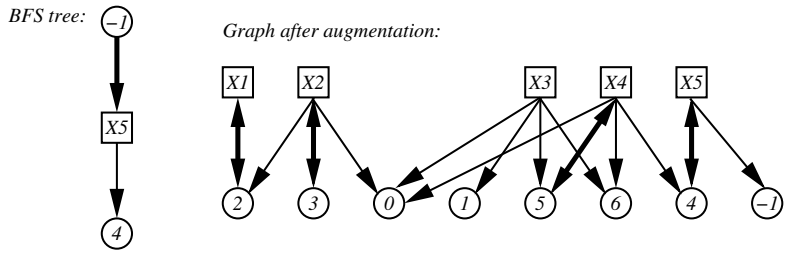


Figure 4.7: The BFS tree for the value -1, and the graph after the augmentation.

in Figure 4.8. Observe that the value 4 does not belong to the tree, although it is adjacent to X_4. This is because the previous `BFS` found only one augmenting path, and hence the marks were not reset. After the augmentation with the path ending in the value 6, we are done. Our matching $M = \{\{X_1, 2\}, \{X_2, 3\}, \{X_3, 6\}, \{X_4, 5\}, \{X_5, 4\}\}$ is a maximum weight $\mathcal{R}$-matching.

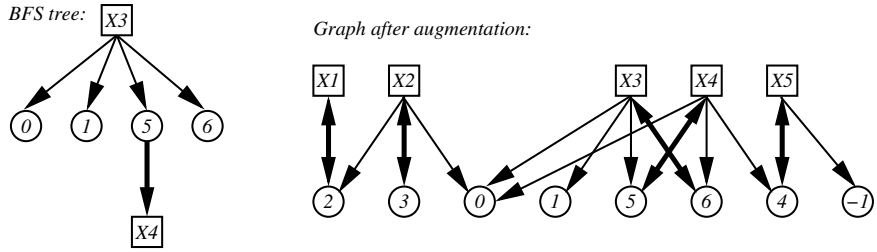


Figure 4.8: The BFS tree for X_3 and the final oriented value graph.

We conclude this section with an analysis of the running time of the algorithm. Let us recall some notation. We denote by m the sum of the cardinalities of all variable domains, i.e. $m = \sum_{i=1}^{n} |Dom(X_i)|$. Observe that m is equal to the number of edges in the value graph G. Furthermore, let $D = \bigcup_{i=1}^{n} Dom(X_i)$ and $d = |D|$ be the number of distinct values in the domains.

For our analysis we need to set up some assumptions on the environment in which the algorithm is embedded. The constraint programming system must allow the algorithm to scan the domain of a given variable. We suppose that the time needed to enumerate all values in a domain is linear in the size of the domain. Observe that we did not impose any restrictions on the types of the values in the domains. However, we must be able to associate some

constant size information with every value in D (like the current matching mate for example). We assume that accessing such an information takes constant time. One possibility to meet these requirements is to restrict the domains to sets of integers drawn from the range $[1..d]$. If one wants to allow arbitrary integers, one may consider perfect hashing [FKS84].

In the second phase of the algorithm, we process the matched values of M_{init} in increasing weight order. We assume that the algorithm is given a list L_{init} that contains these values in sorted order. Provided that two weights can be compared in constant time, we can build L_{init} in time $O(|M_{init}|\cdot\log|M_{init}|)$. We will discuss later how L_{init} can be maintained in an incremental setting such that this sorting step can be avoided.

We will show now that the total running time of the algorithm is bounded by $O(nm)$ (apart from the sorting step). The time for a single `BFS` is at most $O(m)$, because the domain of every variable is scanned at most once. Now we have to bound the total number of `BFS` calls. We make exactly one call for every value that is matched in M_{init} (in steps one and three of phase two), and one call for every variable that is free in M_{init} (in phase one and step two of phase two). As the number of matched values is equal to the number of matched variables, the total number of `BFS` calls is equal to the number of variables. This proves the following theorem:

Theorem 4.2 *Let G be the weighted value graph which is constructed from the domains of the assignment variables $X_1, \ldots, X_n$ and the value-weight table. Set $m = \sum_{i=1}^{n} |Dom(X_i)|$ and $d = |\bigcup_{i=1}^{n} Dom(X_i)|$. Then a maximum weight matching M in G can be computed in time $O(nm)$. And the space requirement is $O(n+d)$ (plus the space for storing the variable domains).*

Proof. Run the algorithm with $M_{init} = \emptyset$, thus no sorting is necessary. □

The analysis above is quite pessimistic. We can observe that the mark of a value node can only be reset from *reached* to *unreached* when an augmentation occurs. So the time between two successive augmentations is bounded by $O(m)$, no matter how many unsuccessful `BFS` calls we make in between. Thus if a denotes the total number of augmentations, the total running time is $O((a+1)m)$.

This observation will play an important role in the next section, where we deal with some aspects of integrating the algorithm in a constraint programming framework. We can record the following: If our initial matching M_{init} is nearly optimal, i.e. if there are not many augmentations necessary, then the algorithm is quite fast. In particular, if M_{init} is already a maximum weight $\mathcal{R}$-matching, the running time of the algorithm is $\Theta(m)$.

4.2.6 Integration of the algorithm into a CP framework

We discuss some issues concerning the integration of the algorithm into a constraint programming system. The situation is the following. Suppose a new *WPA*-constraint is posted, then the propagation computes a maximum weight $\mathcal{R}$-matching in the corresponding initial weighted value graph. During search and upon backtracking the domains of the variables involved in the constraint may be updated, which implicitly changes the weighted value graph as well. Because of the domain change the CP system wakes up the constraint again and reruns the propagation algorithm, which computes a matching in the current weighted value graph G. In most cases, G does not change much between two successive invocations of the matching algorithm. And hence, we expect that the corresponding matchings should be similar. Thus using the old matching as a starting point for the new matching should be more efficient than computing the new matching from scratch.

So the propagation algorithm proceeds as follows: When it is called for the first time, it computes $\mathcal{R}$ and invokes the matching algorithm with $M_{init} = \emptyset$ to obtain a maximum weight $\mathcal{R}$-matching $M^\star$, which is stored for future use. When the propagation algorithm is called again later, it recomputes $\mathcal{R}$ and uses $M^\star$ to construct a new initial matching M_{init} for Algorithm 4.2: For every edge $\{Var, val\}$ in $M^\star$, we check whether val is still contained in the current domain of Var. If so, we put the edge into M_{init}. The time for constructing M_{init} is $O(m)$.[2] Since we assume that the domains of the variables do not change too much between two successive invocations of the propagation algorithm, this should yield a good initial matching.

Incremental maintenance of L_{init}

In the second phase of Algorithm 4.2 we process the values matched in M_{init} in increasing weight order. We show how we can maintain these values in a sorted list L_{init} without increasing the asymptotic running time of $O((a + 1) \cdot m)$, where a is the number of augmentations. For the first run there is no problem, because M_{init} and hence L_{init} are empty. So let us assume that we have M_{init} and L_{init}. We show how to modify the algorithm such that it does not only compute a maximum weight matching $M^\star$ but also a list $L^\star$ that contains the matched values of $M^\star$ in increasing weight order.

As before, the algorithm maintains a current matching M which is augmented until it is optimal. In addition, we have a list L in which we store

[2] We still suppose that the time for enumerating a variable domain is linear in its cardinality.

all values that are matched in M but free in M_{init}. At the beginning L is empty. Whenever an augmentations occurs (see line 22 in Algorithm 4.1) such that *opt_end* is a (free) value different from *undef*, we append *opt_end* to L. And if the root x of the BFS tree is a (matched) value, then x is also matched in M_{init} but is free after the augmentation, and hence we remove[3] x from L_{init}. Thus at any time $L_{init} \cup L$ contains exactly the matched values of M. When phase two terminates with the matching $M^\star$, we construct $L^\star$ as follows. First we sort the values in L according to their weight, and then we merge the two sorted lists L_{init} and L.

Let us analyse the running time for this computation. Maintaining L and L_{init} requires only constant time per augmentation. Sorting L takes time $O(|L| \log |L|)$, and merging can be done in time $O(|L| + |L_{init}|)$. Since the cardinality of L is bounded by a and by m, the modified algorithm still runs in time $O((a+1)m)$.

Tailoring the algorithm for edge deletions

When the CP system searches a solution for a constraint program, it tries to narrow the variable domains until some propagation algorithm reports a failure or all domains have become singletons. Only if a failure occurs, which forces the system to backtrack, the domains can grow again. But in most of the cases the variable domains shrink between successive invocations of the matching algorithms. This means that edges are deleted from the weighted value graph, but no new edges are inserted.

So we study the following problem: Suppose that we have computed an *undef*-free maximum weight $\mathcal{R}^\star$-matching $M^\star$ in the weighted value graph $G^\star$. Then we prune a set P of edges from $G^\star$ and obtain a new graph G and possibly a new set $\mathcal{R}$. Our goal is to determine a maximum weight $\mathcal{R}$-matching in G. We show that we can modify Algorithm 4.2 such that its running time is $O(m \min(|P|, n))$ when it is applied to the initial matching $M_{init} = M^\star \setminus P$. (The modified algorithm still works for arbitrary initial matchings in arbitrary weighted value graphs in the same asymptotic running time as the original algorithm.)

Let us consider an $\mathcal{R}$-augmenting or weight-augmenting path $\vec{p}$ from a node x to a node y in $\vec{G}_{M_{init}}$. Since $\vec{p}$ is also a path in $\vec{G}^\star_{M^\star}$, but not an augmenting one, we conclude that some properties of x or y have changed due to the deletion. To be more precise, at least one of the following three statements holds:

[3] L_{init} and L are doubly linked lists, and for every matched value we store the address of the corresponding list-item, so we can delete an item in constant time.

1. x is in $\mathcal{R}$ but not in $\mathcal{R}^\star$, which is the case iff $\{x, \mathit{undef}\} \in P$.

2. x is a variable and free in M_{init}, but matched in $M^\star$.

3. The value y is free in M_{init}, but matched in $M^\star$.

So we say that a node z *has been affected by the pruning* iff z is in $\mathcal{R} \backslash \mathcal{R}^\star$ or z is matched in $M^\star$ and free in M_{init}. In the sequel we modify the matching algorithm to make sure that the following holds: Whenever we augment the current matching with a path p, then at least one end node of p has been affected by the pruning. This will allow us to bound the number of augmentations and obtain the claimed running time.

Let us make an important observation. If we look at the proof of Theorem 4.1, we see that the order of the `BFS` calls is not essential for the correctness of the algorithm. We could use another order as long as we make sure to reset the marks of all values if the weight threshold decreases between two successive calls. What matters for correctness is that `BFS` is called for the set S that contains every free variable and every matched value wrt. M_{init}. It will turn out useful to partition S in two sets A and $\bar{A}$ such that A contains those nodes in S which have been affected by the pruning. The modified algorithm processes S in two phases:

- Phase I: Process $\bar{A}$:
  $\bar{A}$ contains every value that is matched in M_{init} and every variable that is not $\mathcal{R}$ and free in both M_{init} and $M^\star$. We consider these nodes in weight increasing order, and hence, we have to reset the marks of the values only once at the beginning of the phase. Note that this phase is very similar to phase 2 of the original algorithm.

- Phase II: Process A:
  A can be decomposed in two sets again. In A_1, we have all variables in $\mathcal{R}$ which are free in M_{init}. Into A_2 we put all variables that are not in $\mathcal{R}$ and that are free in M_{init} and matched in $M^\star$. Observe that we must match the nodes in A_1 in order to turn M_{init} into an $\mathcal{R}$-matching, so that the weight threshold for the corresponding `BFS` calls is $-\infty$. For the nodes in A_2 the threshold is 0, because we are not required to match them. So if we process the nodes in A_1 before those in A_2, it suffices to reset the marks of the values at the beginning of the phase.

The following lemma analyses the running time of the modified algorithm:

Lemma 4.3 *Let P be a set of edges in a weighted value graph $G^\star$. If we have a maximum weight matching $M^\star$ in $G^\star$, then we can compute a maximum*

weight matching M in the graph $G = G^\star \setminus P$ in time $O(m \min(|P|, n))$, where n is the number of variables and m is the number of edges in G.

Proof. It suffices to show that the number of augmentations in each phase of the modified algorithm is at most $|P|$. The number of augmentations in phase II is bounded by $|A|$, because every augmenting path considered in this phase starts in a different node in A. Since every node in A has been affected by the pruning and each edge in P can affect at most one variable, we have $|A| \leq |P|$.

We may assume that $M^\star$ is *undef*-free, so *undef* is not affected by the pruning. For any matching M that is computed in phase I, we will prove the following invariant by induction: If q is a weight-augmenting path wrt. M in G, then at least one end node of q has been affected by the pruning. The base case $M = M_{init}$ has already been discussed.
So let us suppose that $M' = M \oplus p$, where p is an augmenting path computed in phase I, and M is a matching for which the invariant holds. Consider a weight-augmenting path q wrt. M'. If p and q are node-disjoint there is nothing to show. Otherwise we translate p to a directed path $\vec{p}$ from x to y in $\vec{G}_M$ and q to a path $\vec{q}$ in $\vec{G}_{M'}$ from u to v. If u has been affected by the pruning there is nothing to show. So let us assume it is not, which implies $u \in \bar{A}$ (by similar arguments as in the proof of Lemma 4.1). Since q is weight-augmenting, we have $\pi_M(u) \geq \mathit{weight}(v) > \mathit{weight}(u)$, and hence u has not been processed yet. Thus $\mathit{weight}(x) \leq \mathit{weight}(u)$, and we conclude $\mathit{weight}(x) < \mathit{weight}(v)$. The same argument as in the proof of Lemma 4.2 shows $v \in R_M(x)$. So there exists a weight-augmenting path q' wrt. M from x to v. Observing $x \in \bar{A}$ and applying the induction hypothesis, we conclude that v is affected by the pruning.

So every augmentation in phase I is made with the aid of a path p that ends in a free value y which is matched in $M^\star$ and free in M_{init}. As $y \neq \mathit{undef}$, y is matched again after the augmentation and remains matched till the end of the algorithm. Therefore the number of augmentations in this phase is bounded by $|P \cap M^\star|$. □

We conclude these considerations with an example which demonstrates that the original algorithm may introduce augmenting paths where both end nodes have not been affected by pruning. This shows that the arguments above only hold for the modified algorithm. On the left-hand side of Figure 4.9 we have depicted a graph $\vec{G}^\star_{M^\star}$ for a maximum weight matching $M^\star$. Suppose we prune the edges e and f and obtain the graph G and a matching M_{init}. Then the optimal augmenting path p for the variable X_1 ends in the (now) free value 5. After the augmentation $M = M_{init} \oplus p$, we have the situation shown on the right-hand side of the figure. There are three

weight-augmenting paths, where both end nodes have not been affected by the pruning: from 2 to 3, from X_4 to 3 and from X_4 to 1.

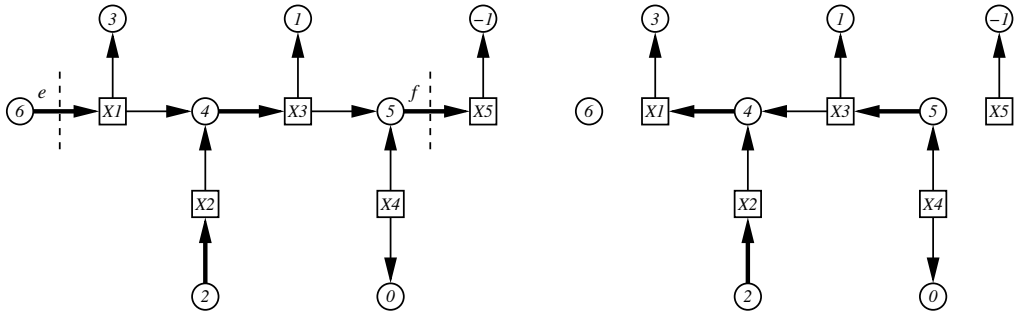


Figure 4.9: Example showing why Algorithm 4.2 is modified.

4.2.7 Computation of the upper regret

If we are given some variables $X_1, \ldots, X_n$ with their respective domains and a value-weight table T, we know how to compute the maximum weight w of a variable assignment. Suppose now, we pick one variable Var and a value $val \in Dom(Var)$, and we allow only assignments where Var is assigned the value val. We want to determine the maximum weight w' of such a restricted assignment. (If no such assignment exists, we set $w' = -\infty$.) We define the *upper regret* for the variable-value pair (Var, val) to be the difference $w - w'$ and denote it by $\overline{regret}(Var, val)$. So the upper regret tells us by which amount the maximum weight decreases, if we force the assignment of val to Var. This number is interesting because it allows us to prune the domain of Var with respect to the weight variable W: If $w - \overline{regret}(Var, val) < \underline{W}$, we can remove val from $Dom(Var)$.

We translate the problem to a matching problem in the value graph G. By Lemma 4.1, w is the weight of a maximum weight $\mathcal{R}$-matching M in G. And w' is the weight of a matching M' that has maximum weight among all $\mathcal{R}$-matchings containing the edge $e = \{Var, val\}$. It will turn out that we do not have to compute M' explicitly, but we are able to determine the upper regret with the aid of $\vec{G}_M$. We have to distinguish whether Var and val belong to the same SCCs of $\vec{G}_M$ or not.

Assume first that Var and val are in the same component, which implies that there is a simple cycle $\vec{c}$ in $\vec{G}_M$ that uses the edge (Var, val). If $\vec{c}$ consists only of two edges, then $\vec{c} = (Var, val) \circ (val, Var)$, which implies $e \in M$ and $\overline{regret}(Var, val) = 0$. Otherwise $\vec{c}$ translates to an alternating

cycle c in G wrt. M. Since M and $M \oplus c$ match the same nodes, we conclude $\overline{regret}(Var, val) = 0$.

We come to the second case: Var and val belong to different SCCs of $\vec{G}_M$. This implies that $e \notin M$ and that $M \oplus M'$ contains an acyclic alternating path p which uses e. The path p translates to a path $\vec{p}$ in $\vec{G}_M$ from a node x to a node y. Each of the two nodes is free in exactly one of the two $\mathcal{R}$-matchings M and M', and hence $x, y \notin \mathcal{R}$. With respect to M, x is a free variable or a matched value, and y is a matched variable or a free value (cf. Figure 4.10). Recalling that the weight of a variable is zero, we obtain $weight(M \oplus p) = weight(M) - weight(x) + weight(y)$. An analogous argument shows $weight(M') = weight(M' \oplus p) - weight(x) + weight(y)$. (Observe that $\vec{p}$ is also a path in $\vec{G}_{M' \oplus p}$.) Since M is a maximum weight $\mathcal{R}$-matching, we have $weight(M' \oplus p) \leq weight(M)$ and we can infer

$$\begin{array}{rcl} weight(M') & = & weight(M' \oplus p) - weight(x) + weight(y) \\ & \leq & weight(M) - weight(x) + weight(y) \\ & = & weight(M \oplus p) \leq weight(M') \end{array}$$

Thus $weight(M') = weight(M \oplus p)$, which means that we can assume $M' = M \oplus p$ from now on. In addition, we see $\overline{regret}(Var, val) = weight(M) - weight(M \oplus p) = weight(x) - weight(y)$, i.e. the upper regret depends only on the start and the end node of $\vec{p}$. We can decompose $\vec{p} = \vec{p}_{Var} \circ (Var, val) \circ \vec{p}_{val}$, where $\vec{p}_{Var}$ is a (possibly empty) path from x to Var and $\vec{p}_{val}$ is a (possibly empty) path from val to y.

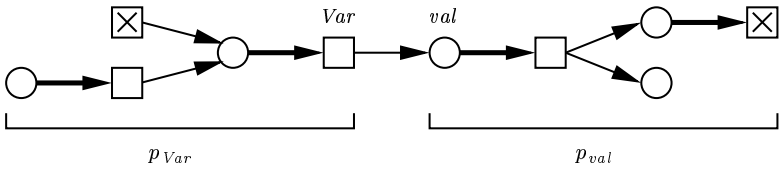


Figure 4.10: Possible start and end nodes for the path $\vec{p}$.

Let us assume that M is *undef*-free . We show that $weight(y) = \pi_M(val)$. Assume otherwise. If $\pi_M(val) > weight(y)$, then there is a path $\vec{q}$ in $\vec{G}_M$ from val to a free node $\tilde{y}$ with $weight(\tilde{y}) = \pi_M(val)$ (see Definition 4.3). Translating the path $\vec{p}_{Var} \circ (Var, val) \circ \vec{q}$ back to G, we obtain a path $\tilde{p}$. $M \oplus \tilde{p}$ is a matching containing e with $weight(M \oplus \tilde{p}) = weight(M) - weight(x) + weight(\tilde{y}) > weight(M')$, which is a contradiction.
So $\pi_M(val) \leq weight(y)$. If y is a value node, we have $\pi_M(val) \geq weight(y)$ by the definition of π_M. Otherwise y is a variable and y is not in $\mathcal{R}$, which means

that y is incident to the free value *undef*. Hence, $\pi_M(val) \geq weight(undef) = 0 = weight(y)$.

ρ-values

We will now define a function ρ_M such that $\rho_M(Var) = weight(x)$. We say that a node u is a *possible start node wrt. M* if u is a free variable or a matched value wrt. M. Note that this implies $u \notin \mathcal{R}$. For a node v we set $\rho_M(v) := \inf\{weight(u) \mid v \in R_M(u) \wedge u \text{ is a possible start node wrt. } M\}$.[4] We show that $\rho_M(Var)$ is indeed equal to $weight(x)$. From this definition we conclude $\rho_M(Var) \leq weight(x)$. Assume $\rho_M(Var) < weight(x)$. Then there is a path $\vec{r}$ in $\vec{G}_M$ from a node $\hat{x}$ to Var with $weight(\hat{x}) = \rho_M(Var) < weight(x)$. The path $\vec{r} \circ (Var, val) \circ \vec{p}_{val}$ translates to an alternating path $\hat{p}$ in G. $M \oplus \hat{p}$ contains e and $weight(M \oplus \hat{p}) = weight(M) - weight(\hat{x}) + weight(y) > weight(M')$, a contradiction.

The results from above can be summarized as follows: If Var and val belong to different SCCs of $\vec{G}_M$, then $\overline{regret}(Var, val) = \rho_M(Var) - \pi_M(val)$. If they belong to the same SCC, we have $\overline{regret}(Var, val) = 0$.
Now we are ready to outline an algorithm for computing the upper regret for any variable-value pair:

1. Compute the strongly connected components of $\vec{G}_M$:
   We assign a component number to every node such that two nodes receive the same number iff they belong to the same SCC.

2. Label every variable node Var with $l[Var] = \rho_M(Var)$.

3. Label every value node val with $l[val] = \pi_M(Var)$.

4. Compute the upper regret:
   For every pair (Var, val) with $val \in Dom(Var)$ we compute the upper regret as follows:
   $$\overline{regret}(Var, val) = \begin{cases} 0, & \text{if } Var, val \text{ belong to same SCC} \\ l[Var] - l[val], & \text{otherwise} \end{cases}$$

We will now discuss how to implement the algorithm in time $O(m)$, where m is the sum of the cardinalities of the variable domains as in the previous section. We assume that we have a sorting of the matched values according to their weights; recall that our extended maximum matching algorithm provides such a sorting. Computing the strongly connected components of a directed graph can be done in linear time. Some of these algorithms only scan the outgoing edges of a node, but not the incoming ones (see for example [CM96]).

[4] Here "inf S" denotes the infimum of S, which is equal to $\min(S \cup \{\infty\})$ if S is finite.

So the critical steps are the node labellings. We begin with the computation of the variable labels. Looking at the definition of ρ_M, we recognize the following: If u is a possible start node with minimum weight, then for all nodes $v \in R_M(u)$ we have $\rho_M(v) = \mathit{weight}(u)$. This observation leads to the following algorithm. First we label all variables with ∞. Then we consider all possible start nodes u in weight increasing order. We start a depth-first search (DFS) at each u and assign the label $\mathit{weight}(u)$ to all variables that u can reach and have not been reached before.
The pseudo-code for this approach is given in Algorithm 4.3. We observe that every variable domain is scanned exactly once, and hence the running time is $O(m)$. The correctness of the algorithm is implied by the following observation: After a node u has been processed in one of the for-loops in lines 2, 4 or 6, we have $l[\mathit{Var}] = \rho_M(\mathit{Var})$ for every variable $\mathit{Var} \in R_M(u)$.

Algorithm 4.3 Computation of the variable labels

Procedure: ComputeVarLabels(M)
1: initialize the labels of all variable nodes to ∞
2: **for all** values val matched in M with $\mathit{weight} < 0$ in incr. order **do**
3: VarDFS($M[\mathit{val}]$, $\mathit{weight}(\mathit{val})$)
4: **for all** variables Var free in M **do**
5: VarDFS(Var, 0)
6: **for all** values val matched in M with $\mathit{weight} \geq 0$ in incr. order **do**
7: VarDFS($M[\mathit{val}]$, $\mathit{weight}(\mathit{val})$)

Procedure: VarDFS(Var, w)
8: **if** $l[\mathit{Var}] = \infty$ **then**
9: $l[\mathit{Var}] \leftarrow w$
10: **for all** values $\mathit{val}' \in \mathit{Dom}(\mathit{Var})$ **do**
11: **if** val' has a matching mate Var' **then**
12: VarDFS(Var', w)
13: **end if**
14: **end if**

Now we discuss the labelling of the value nodes. One is tempted to use a similar approach as above: Process all free values in decreasing weight order, for each such node v label every unlabelled value that can reach v with $\mathit{weight}(v)$. The problem with this approach lies in the term "can reach v", i.e. the approach requires to scan the incoming edges of the nodes in $\vec{G}_M$. Since we do not want to build the graph explicitly, we would have to ask the constraint programming system questions of the following form: Given a

value val, tell us every variable Var such that $val \in Dom(Var)$. To the best of our knowledge there is no system that supports such a query efficiently.

So we take a different approach. It will turn out that we can use DFS again, but we have to apply it to an acyclic graph. We observe that two values that belong to the same strongly connected component can reach exactly the same nodes, which implies that their potentials are equal. And hence, we can use a well-known technique to make $\vec{G}_M$ acyclic: We shrink its SCCs. The shrunken graph $\vec{G}_M^{\mathrm{s}}$ contains a node for each component C of $\vec{G}_M$, and there is an edge (C_1, C_2) in $\vec{G}_M^{\mathrm{s}}$ iff there is an edge (x, y) in $\vec{G}_M$ with $x \in C_1$ and $y \in C_2$.

Let us consider what this means for our running example. On the left-hand side of Figure 4.11 we have marked the SCCs of $\vec{G}_M$ with dashed boxes. Observe that every free node forms a trivial component of its own, whereas the component of every matched node contains at least the mate. Note that shrinking the components collapses the edges $(X_3, 0)$ and $(X_4, 0)$ into the single edge (C_5, C_3).

Now we can state a recursive algorithm for computing the value labels of a component C: Assume first that C has no outgoing edges, then there are two possible cases for the label $l[C]$. If C consists of a single free value val, then $l[C] = weight(val)$. Otherwise C contains no free value, and hence, $l[C] = -\infty$. So if C has no outgoing edges, the recursion terminates immediately. Suppose now that C has outgoing edges, which implies that there are no free values in C. We initialize $l[C]$ with $-\infty$. For any edge (C, C') in $\vec{G}_M^{\mathrm{s}}$ we compute $l[C']$ recursively and set $l[C]$ to the maximum of its current value and $l[C']$. Note that this algorithm always terminates because the graph is acyclic.

The algorithm can be implemented with DFS. On the right-hand side of Figure 4.11, we have depicted the shrunken graph and for each component we indicate the time when the corresponding DFS call completes: for example, $C_2 : 3$ means that C_2 is the third component that gets completed. One can see that the source of an edge is always completed after the target. This holds for every DFS in an acyclic graph. And hence, whenever a component is completed by DFS, we can assign the correct label to it.

It is not necessary to construct $\vec{G}_M^{\mathrm{s}}$ explicitly. We can perform a DFS in the graph $\vec{G}_M$ (see Algorithm 4.4). The shrinking of the SCCs is done implicitly: We determine a single label for all values in an SCC. To prove the correctness of the algorithm, we introduce some terminology. We define the *root* of an SCC C to be the first value r in C that is discovered by the algorithm, where discovery means that the status changes from *unreached* to *active*. Since all nodes of C become DFS descendants of r, the root is the last

Algorithm 4.4 Computation of the value node labels

Procedure: ComputeValLabels(M,$\mathcal{R}$,SCC)
Require: $SCC[val]$ stores the SCC number for each value val and M is *undef*-free
1: initialize the labels of all SCCs with $-\infty$
2: initialize *mark* of all values as *unreached*
3: **for all** values val **do**
4: **if** $mark[val] = unreached$ **then**
5: ValDFS(val)

Procedure: ValDFS(val)
6: $mark[val] \leftarrow active$
7: **if** val is free **then**
8: $l[SCC[val]] \leftarrow weight(val)$
9: **else**
10: let Var denote the matching mate of val
11: **for all** values $val' \in Dom(Var)$ **do**
12: **if** $mark[val] = unreached$ **then**
13: ValDFS(val')
14: **end if**
15: $l[SCC[val]] \leftarrow \max(l[SCC[val]], l[SCC[val']])$
16: **end if**
17: $mark[val] \leftarrow completed$

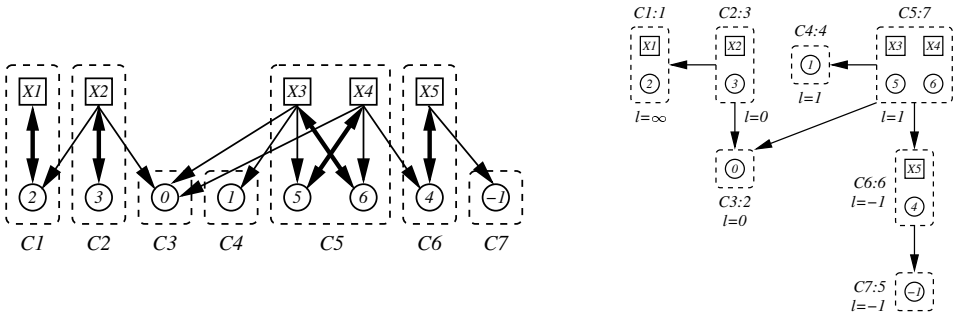


Figure 4.11: The strongly connected components of $\vec{G}_M$ and the completion times of a DFS in $\vec{G}_M^{\mathrm{s}}$.

node in C which is marked *completed*. We say that an edge (Var, val') in $\vec{G}_M$ is *finished*, when line 15 has been executed for val' during the enumeration of the domain of Var in line 11. The correctness of the algorithm follows immediately from the second statement of the following lemma:

Lemma 4.4 *During the execution of Algorithm 4.4 the following holds:*

1. *If an edge (Var, val') is finished and Var and val' belong to different SCCs, then all nodes in the component of val' are marked completed.*

2. *If the root of a component C is marked completed, then label of C is equal to the potential of all values in C.*

Proof. Suppose that the first claim is false. This means that an edge $e = (Var, val')$ which connects nodes in different SCCs gets finished, before all nodes in the component of val' are completed. In particular, the root r' of this component is not completed. Since `ValDFS(`val'`)` is called immediately before e is finished, we can conclude that r' is active when e gets finished. And hence, Var is a descendant of r' in the DFS tree, which implies that r' can reach Var. As Var can reach val' and val' can reach r', we have that Var is in the same SCC as r' and val', a contradiction.

We prove the second statement. It is easy to see that the labels assigned by the algorithm can never be higher than the correct labels. So we can ignore components with potential $-\infty$. Let r denote the root of a component with $\pi_M(r) > -\infty$. Thus there is a path $\vec{p}$ in $\vec{G}_M$ from r to a free value y with $weight(y) = \pi_M(r)$. We prove the statement by an induction on the number of components that are visited by $\vec{p}$. The base case for our induction is that r and y lie in the same SCC. Since y is a free value, its component is labelled correctly after the completion of the call DFS(y) (see line 8).

Now we come to the induction step. Assume that r and y reside in different SCCs, and let $e = (Var, val')$ be the first edge on $\vec{p}$ that connects a node in the SCC of r with a node outside. Consider the point in time when e is finished, this is clearly before the completion of r. By the first statement of the lemma, the root r' of the component of val' is marked completed at that time. So by the induction hypothesis (applied to the suffix of $\vec{p}$ from val' to y), the component of r' has the correct label $weight(y)$. Thus the component of r also receives the correct label (see line 15). □

Example. In Figure 4.12 we show all the node labels for our running example. Observe that the label of the value 2 is $-\infty$, which means that this value must be matched with X_1 in any $\mathcal{R}$-matching, because X_1 is the only variable in the SCC of 2. Moreover, we can see for example that $\overline{regret}(X_4, 4) = 5 - (-1) = 6$. And indeed, if we want to construct a maximum weight $\mathcal{R}$-matching containing the edge $\{X_4, 4\}$, we have to set value 5 free and match X_5 with -1. The latter is necessary because we are required to match X_5.

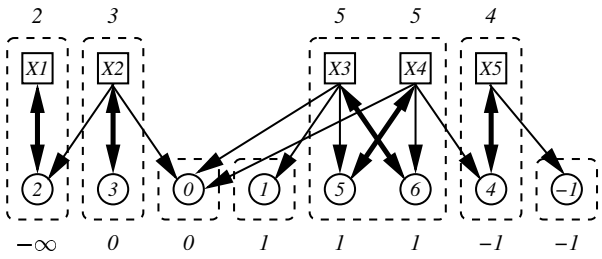


Figure 4.12: Example showing the computation of the upper regret.

4.2.8 Putting it all together

In this section we describe how to assemble the algorithms developed above to a propagation algorithm for the *WPA*-constraint. We will also prove some statements regarding consistency and the overall running time.
Before we describe Algorithm 4.5, we introduce the notion *lower regret*. It is defined in an analogous way as the upper regret: Let w_{min} denote the weight of a minimum weight $\mathcal{R}$-matching in the weighted value graph G. Consider an edge $e = \{Var, val\}$ and let w be the minimum weight of any $\mathcal{R}$-matching containing e. (If no such $\mathcal{R}$-matching exists set $w = \infty$.) We define $\underline{regret}(Var, val) = w - w_{min}$, i.e. the lower regret tells us by which amount the minimum weight increases if we force the assignment of val to

weight interval $I(e)$

Var. If w_{max} is the maximum weight of an $\mathcal{R}$-matching in G, then the weights of all $\mathcal{R}$-matchings containing e lie in the interval $I(e) := [w_{min} + \underline{regret}(Var, val); w_{max} - \overline{regret}(Var, val)]$. We call $I(e)$ the *weight interval* of e, and we will make sure that the following holds when our algorithm terminates: For any edge e that is not pruned we have $I(e) \cap Dom(W) \neq \emptyset$.

The algorithm works as follows. First we compute the set $\mathcal{R}$. Then we enter the main loop, which repeats the following steps: We determine a minimum weight matching M_{min} in the weighted value graph G and the corresponding lower regrets[5]. If M_{min} does not exist, or if $weight(M_{min})$ is greater than the upper endpoint of $Dom(W)$, the constraint has no solution and we report failure. With the lower regrets we can prune the domains of $X_1, \ldots, X_n$ according to the upper endpoint of $Dom(W)$. After that we compute a maximum weight matching M_{max} and the upper regrets, which allows us to perform similar pruning steps as before.

Unfortunately, it is not guaranteed that we reach a fixpoint after one iteration of the main loop. Hence, we repeat these steps until no edge deletion occurs anymore. Of course, we only recompute the matching M_{min} or M_{max} if this is necessary. We do not start from scratch, but rather use the old matching (minus the removed edges) as input for our matching algorithm. A recomputation becomes necessary, if an edge in the matching is deleted, or if a free variable enters $\mathcal{R}$. We observe that the lower regret of all edges in M_{min} is zero, and for every variable $X \notin \mathcal{R}$ which is free in M_{min} we have $\underline{regret}(X, undef) = 0$. And hence, the pruning according to the lower regrets can never invalidate M_{min}, but only M_{max}. A similar observation can be made for M_{max} and the upper regrets. Moreover, if the pruning according to the upper regrets deletes some edges, we do not have to recompute the upper regrets unless there are also edge removals due to the lower regrets. We will prove this in the following lemma.

Lemma 4.5 *Let M be a maximum weight $\mathcal{R}$-matching in the weighted value graph G. Suppose the upper regret of an edge e increases as we delete an edge d from G with $\overline{regret}(d) > 0$. Then $\overline{regret}(e) \geq \overline{regret}(d)$ holds before the deletion. An analogous statement can be made for the lower regrets. Hence, if Algorithm 4.5 removes d because its regret is too high, then it also removes e.*

Proof. We only have to prove the claim for the upper regrets. Let $d = \{Var_d, val_d\}$ and $e = \{Var_e, val_e\}$. In the sequel we will use the subscript "1" if we refer to the graph $\vec{G}_1 = \vec{G}_M$ and the subscript "2" for $\vec{G}_2 = \vec{G}_M \backslash d$. Since

[5] As we stated before, these quantities can be determined by multiplying all weights by -1 and computing a maximum weight matching and the upper regrets.

Algorithm 4.5 Propagation algorithm for *WeightedPartialAlldiff*

Procedure: PropagateWPA($X_1, \ldots, X_n$;*undef*;T;W)

compute $\mathcal{R} = \{X_i \mid \mathit{undef} \notin \mathit{Dom}(X_i)\}$
initialize M_{min} and M_{max} // maybe empty
recompute_min $\leftarrow$ *true*; *recompute_lregret* $\leftarrow$ *true*
recompute_max $\leftarrow$ *true*; *recompute_uregret* $\leftarrow$ *true*
while *recompute_lregret* or *recompute_uregret* **do**
  if *recompute_min* **then**
    compute min weight *undef*-free $\mathcal{R}$-matching M_{min}
    if M_{min} does not exist or $\mathit{weight}(M_{min}) > \overline{W}$ **then**
      report *failure* and terminate
    $\underline{W} \leftarrow \max(\underline{W}, \mathit{weight}(M_{min}))$
    recompute_min $\leftarrow$ *false*
  end if
  if *recompute_lregret* **then**
    compute the lower regrets (wrt. M_{min})
    for all variables X_i and **all** values $v \in \mathit{Dom}(X_i)$ **do**
      if $\mathit{weight}(M_{min}) + \overline{\mathit{regret}}(X_i, v) > \overline{W}$ **then**
        remove v from $\overline{\mathit{Dom}(X_i)}$; *recompute_uregret* $\leftarrow$ *true*
        if $v = \mathit{undef}$ **then** add X_i to $\mathcal{R}$
        if $\{X_i, v\} \in M_{max}$ or (X_i free in M_{max} and $v = \mathit{undef}$) **then**
          recompute_max $\leftarrow$ *true*
      end if
    recompute_lregret $\leftarrow$ *false*
  end if
  if *recompute_max* **then**
    compute max weight *undef*-free $\mathcal{R}$-matching M_{max}
    if $\mathit{weight}(M_{max}) < \underline{W}$ **then** report *failure* and terminate
    $\overline{W} \leftarrow \min(\overline{W}, \mathit{weight}(M_{max}))$
    recompute_max $\leftarrow$ *false*
  end if
  if *recompute_uregret* **then**
    compute the upper regrets (wrt. M_{max})
    for all variables X_i and **all** values $v \in \mathit{Dom}(X_i)$ **do**
      if $\mathit{weight}(M_{max}) - \overline{\mathit{regret}}(X_i, v) < \underline{W}$ **then**
        remove v from $\mathit{Dom}(X_i)$; *recompute_lregret* $\leftarrow$ *true*
        if $v = \mathit{undef}$ **then** add X_i to $\mathcal{R}$
        if $\{X_i, v\} \in M_{min}$ or (X_i free in M_{min} and $v = \mathit{undef}$) **then**
          recompute_min $\leftarrow$ *true*
      end if
    recompute_uregret $\leftarrow$ *false*
  end if
end while

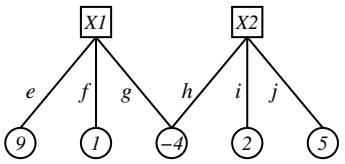
$\overline{regret}_1(d) > 0$, we have $d \notin M$, and M is a maximum weight $\mathcal{R}_2$-matching in G_2 (even if $val_d = undef$). Moreover, Var_d and val_d lie in different SCCs of $\vec{G}_1$. Hence, the SCCs of $\vec{G}_1$ and $\vec{G}_2$ are identical. As the upper regret of e increases by the deletion, Var_e and val_e also belong to different SCCs. Thus $\overline{regret}_1(d) = \rho_1(Var_d) - \pi_1(val_d)$ and $\overline{regret}_1(e) = \rho_1(Var_e) - \pi_1(val_e)$. (We assume w.l.o.g. that M is *undef*-free.) By definition, there is a path $\vec{p}$ in $\vec{G}_1$ from a possible start node x to Var_e with $weight(x) = \rho_1(Var_e)$ (see page 94); and there is a path $\vec{q}$ in $\vec{G}_1$ from val_e to a free node y with $weight(y) = \pi_1(val_e)$ (cf. Definition 4.3).
The deletion of d either increases the ρ-value of Var_e (if d lies on $\vec{p}$) or decreases the π-value of val_e (if d lies on $\vec{q}$). In any case x can reach Var_d. Thus $\rho_1(Var_d) \leq weight(x) = \rho_1(Var_e)$. Moreover, val_d can reach y so that $\pi_1(val_d) \geq weight(y) = \pi_1(val_e)$.
So $\overline{regret}_1(d) = \rho_1(val_d) - \pi_1(val_d) \leq \rho_1(val_e) - \pi_1(val_e) = \overline{regret}_1(e)$. □

Before we analyse the running time of the algorithm, we give an example which shows that it may be necessary to have several iterations of the main loop even if both matchings remain unchanged in one iteration. (This can happen because deleting an edge due to its upper regret may increase the lower regret of another edge.)

Example. We consider the graph G shown on the left-hand side of Figure 4.13, and we assume that $Dom(W) = [4; 5]$. We suppose that every value is equal to its weight. All variables must be matched, i.e. $\mathcal{R} = \{X_1, X_2\}$. We shall see that the algorithm must make 3 iterations, until it has reached a fixpoint. In the table on the right-hand side of the figure we depict the state of the algorithm after the execution of line 14 and line 31 in the respective iteration. An "x" in a column indicates that the respective edge has been pruned by the algorithm.

In the first iteration, g is deleted because of its upper regret. In the second row of the table we see that $weight(M_{max}) = 14$ and $\overline{regret}(g) = 13$. So every $\mathcal{R}$-matching containing g has weight at most 1, which is less than $\underline{W}$. Observe that g neither belongs to M_{min} nor to M_{max}, but the pruning of g affects the lower regrets. This causes the deletion of j in the second iteration. As j was in M_{max}, we have to recompute M_{max} and the upper regrets. At the end of this iteration we prune f, because its upper regret is now too high. Since f was a member of M_{min}, we have to recompute a minimum weight matching in the next iteration. As $weight(M_{min})$ increases to 5, we set $\underline{W}$ to 5. Moreover, we remove i due to its lower regret. Finally, we recompute M_{max} and terminate.

In the sequel we analyse the running time of Algorithm 4.5 and make



	weight		e	f	g	h	i	j
1	$min = -3$	lr	8	0	1	0	1	4
	$max = 14$	ur	0	8	**13**	9	4	0
2	$min = -3$	lr	8	0	x	0	6	**9**
	$max = 11$	ur	0	**8**	x	6	0	x
3	$min = 5$	lr	0	x	x	0	**6**	x
	$max = 5$	ur	0	x	x	0	x	x

Figure 4.13: An example computation of the pruning algorithm

some statements about the consistency it achieves. By Lemma 4.3, the total time for the computations of the matchings is bounded by $O((n+p)m)$, where p denotes the number of pruned variable-value pairs and m denotes the total size of the initial domains. A single iteration takes time $O(nm)$ in the worst case, but if we do not count the time for the matching computations, we obtain $O(m)$ per iteration. Since in all iterations, except for the last one, at least one value is pruned from some domain, Algorithm 4.5 runs in time $O((n + p)m)$. So when the algorithm runs for a very long time, this is compensated by a large amount of pruning.

With respect to consistency, we observe that upon termination of the algorithm we have $I(e) \cap Dom(W) \neq \emptyset$ for any edge e which has not been removed from G. This follows directly from the fact that both matchings and the corresponding regrets are valid upon termination. We summarize our observations in the following theorem:

Theorem 4.3 *Consider the constraint $WPA(X_1, \ldots, X_n; undef; T; W)$ and suppose we apply Algorithm 4.5 to it. Then its worst case running time is $O((n + p)m)$, where p is the number of pruned variable-value pairs. If it terminates without reporting failure, we have $I(\{X_i, v\}) \cap Dom(W) \neq \emptyset$ for any value v that occurs in the domain of some variable X_i.*

Observe that this does not imply that every value $v \in Dom(X_i)$ is consistent. We know that $weight(M) \in I(\{X_i, v\})$ for any $\mathcal{R}$-matching M containing $\{X_i, v\}$. But the converse is not true in general, there may be a value $w \in I(\{X_i, v\})$ such that there is no $\mathcal{R}$-matching M which has weight w and contains $\{X_i, v\}$.

This is shown by the following example: Suppose we have a *WPA*-constraint on three assignment variables X, Y, Z with domains $Dom(X) = \{-4, -2\}$, $Dom(Y) = \{0, 2\}$ and $Dom(Z) = \{4, 6\}$ and a weight variable W with $Dom(W) = \{3\}$. Assume that the weight of every value is equal to the

value itself. Then it is easy to verify that the minimum weight of an assignment is 0, the maximum weight is 6, and the lower or upper regret of any variable-value pair is at most 2. Thus the weight interval of any edge e in the weighted value graph satisfies $I(e) \supseteq [2;4] \supseteq \mathit{Dom}(W)$. Hence, the algorithm does not prune anything. But since every value has an even weight, there can be no solution to the constraint.

In the next lemma, we show that deciding solvability of a *WPA*-constraint is hard (if the weights appearing in its arguments have large absolute value):

Lemma 4.6 *Deciding whether the constraint* $\mathit{WPA}(X_1, \ldots, X_n; \mathit{undef}; T; W)$ *has a solution is NP-complete (in the weak sense).*

Proof. It is easy to see that the problem is in NP, because we can verify in polynomial time that a given variable assignment satisfies the constraint. To prove NP-completeness, we will give a reduction of an NP-complete set partitioning problem called "Subset Sum" (see [GJ79]):

Subset Sum
We have a finite set A, each element $a \in A$ has a weight $w(a) \in \mathbb{Z}^+$, and we have an integer $B \in \mathbb{Z}^+$. The problem is to decide whether there is a subset $A' \subseteq A$ such that $\sum_{a \in A'} w(a) = B$.

In order to encode an instance of the Subset Sum problem with $A = \{a_1, \ldots, a_n\}$ we use n assignment variables $X_1, \ldots, X_n$. We set $\mathit{undef} := 0$. For $i = 1, \ldots, n$ we choose $\mathit{Dom}(X_i) := \{0, a_i\}$ and $\mathit{weight}(a_i) := w(a_i)$. Finally, we set $\mathit{Dom}(W) := [B;B]$. A solution $(v_1, \ldots, v_n, B)$ of the constraint $\mathit{WPA}(X_1, \ldots, X_n; \mathit{undef}; T; W)$ corresponds to the subset $A' = \{v_i \mid v_i \neq \mathit{undef}\} \subseteq A$ of weight B and vice versa. (Observe that this reduction only uses the "weight"-part of the *WPA*-constraint, the "partial alldifferent"-part is satisfied automatically by construction.) □

We conclude the general analysis of the algorithm by showing that it is idempotent and monotonic (cf. page 14):

Lemma 4.7 *Algorithm 4.5 is idempotent and monotonic.*

Proof. It is obvious that the algorithm is idempotent, because the matchings and the regrets are valid upon termination.
To show monotonicity, we consider two applications of the algorithm that do not report failure. We assume that the weight table and the value *undef* are the same in both applications. In the sequel we will use the subscripts "1" and "2" to refer to the respective application. We use the superscript "0" to

denote the input domains and the superscript "$\star$" when we refer to the output domains. Suppose that the input domains satisfy $Dom_1^0(W) \subseteq Dom_2^0(W)$ and $Dom_1^0(X_i) \subseteq Dom_2^0(X_i)$ for $i = 1, \ldots, n$.

We show by induction that at any point in time during the second application the following holds: $Dom_1^\star(W) \subseteq Dom_2(W)$ and $Dom_1^\star(X_i) \subseteq Dom_2(X_i)$ for $i = 1, \ldots, n$. Clearly, the claim holds at the beginning of the second application.

There are four lines where Algorithm 4.5 updates a domain: 10, 17, 27 and 34. For the induction step we assume that the claim holds before such a domain update and prove that it still holds afterwards. So before the update step, the final weighted value graph $G_1^\star$ of the first application is a subgraph of the current weighted value graph G_2 of the second application, and $\mathcal{R}_1^\star \supseteq \mathcal{R}_2$. Hence, a minimum/maximum weight $\mathcal{R}_1^\star$-matching in $G_1^\star$ has at least/most the weight of a minimum/maximum weight $\mathcal{R}_2$-matching in G_2. Therefore the claim holds after an update in lines 10 and 27.
We come to the updates in lines 17 and 34. Consider an edge $e = \{X_i, v\}$ in $G_1^\star$. By Theorem 4.3, the following holds: $I_1^\star(e) \cap Dom_1^\star(W) \neq \emptyset$. So when line 16 or line 33 is executed for e, we have $I_2(\{X_i, v\}) \cap Dom_2(W) \neq \emptyset$, because $I_2(\{X_i, v\}) \supseteq I_1^\star(e)$ and $Dom_2(W) \supseteq Dom_1^\star(W)$. This implies that e is not pruned from G_2, i.e. v remains in $Dom_2(X_i)$.

Hence, when the second call terminates, we have $Dom_1^\star(W) \subseteq Dom_2^\star(W)$ and $Dom_1^\star(X_i) \subseteq Dom_2^\star(X_i)$ for $i = 1, \ldots, n$. □

Scenarios one and two

We can derive better results with respect to both consistency and running time, if we consider again the three scenarios from the beginning of this chapter (see page 71). Let us assume first that we are in the first scenario, where the constraint is used in a minimization problem, i.e. we impose only an upper bound on the weight variable W. Thus $\underline{W}$ is never greater than the weight of a minimum weight matching. By the lemma which we will prove below, this implies that our algorithm terminates after one iteration and that it achieves arc-consistency for the assignment variables:

Lemma 4.8 *Consider the constraint $WPA(X_1, \ldots, X_n; undef; T; W)$ such that $\underline{W}$ is not greater than the minimum weight of an $\mathcal{R}$-matching in the corresponding weighted value graph. If we apply Algorithm 4.5, then it terminates after one iteration, and hence the running time is $O(nm)$. It achieves arc-consistency for the variables $X_1, \ldots, X_n$, i.e. for every variable X_i and every value $v \in Dom(X_i)$ there is a solution $(v_1, \ldots, v_n, w)$ of the constraint*

with $v_i = v$. (The lower endpoint of $Dom(W)$ is made consistent, the upper endpoint may not be consistent.)

Proof. In order to show that the algorithm makes only one iteration, it suffices to prove that the condition of the "if"-statement in line 33 will never be true. Consider the execution of line 33 for a variable-value pair (X_i, v). By Lemma 4.5, we have $\underline{regret}(X_i, v) < \infty$, which implies $\overline{regret}(X_i, v) < \infty$. Thus there is an $\mathcal{R}$-matching M containing the edge $e = \{X_i, v\}$. We infer $weight(M_{max}) - \overline{regret}(X_i, v) \geq weight(M) \geq weight(M_{min})$. As $\underline{W} = weight(M_{min})$ holds after the execution of line 10, the "then"-part of the "if"-statement in line 33 is not executed.

Now we show that there is a solution where v is assigned to X_i. Let G be the weighted value graph at the end of the first iteration. G contains $e = \{X_i, v\}$, and the minimum weight matching and the lower regrets computed by the algorithm are valid for G. So we can find an $\mathcal{R}$-matching M' in G with $e \in M'$ and $weight(M') = weight(M_{min}) + \underline{regret}(X_i, v)$. Since v has not been pruned from $Dom(X_i)$, we have $weight(M') \leq \overline{W}$ (see line 16). Together with $\underline{W} = weight(M_{min})$ this implies $weight(M') \in Dom(W)$. By Lemma 4.1, M' corresponds to a solution where v is assigned to X_i. □

As a corollary from the previous proof, we observe that we can skip the pruning related to the upper regrets (lines 30–40) if $\underline{W} = weight(M_{min})$, because then the condition of the "if"-statement in line 33 will always be false. An analogous observation can be made for the lower regrets (lines 13–23) if $\overline{W} = weight(M_{max})$. However, these lines must not be skipped in the first iteration, because we have to prune the edges with regret ∞.

The situation of the second scenario is very similar to the case above. Recall that we assume in this scenario that the weight variable W is only bounded from below. In this case, we should compute M_{max} and the upper regrets before we compute M_{min} and the lower regrets. Otherwise we might need two iterations until we achieve arc-consistency for the assignment variables.

Scenario three

Now we come to the last scenario where all values (except for *undef*) have weight 1, and we make no assumptions about the lower or the upper endpoint of the domain of the weight variable W. This scenario is analysed in the following lemma:

Lemma 4.9 *Consider a constraint $WPA(X_1, \ldots, X_n; undef; T; W)$ such that all values different from undef have weight 1. Then Algorithm 4.5 achieves*

arc-consistency for all variables (including W). It makes at most two iterations, hence its running time is bounded by $O(nm)$.

Proof. We consider the first iteration and denote by $t^\star$ the point in time immediately after the execution of line 31 in the first iteration. Let $\mathcal{R}^\star$ be equal to the contents of the variable $\mathcal{R}$ and let $G^\star$ denote the weighted value graph at that time. M_{min} and M_{max} are minimum/maximum weight *undef*-free $\mathcal{R}^\star$-matchings in $G^\star$, and the lower and upper regrets computed by the algorithm are valid with respect to $G^\star$.

We show that any value w in $Dom(W)$ is consistent at time $t^\star$. We have $Dom(W) \subseteq [weight(M_{min}); weight(M_{max})]$. The set $M_{min} \oplus M_{max}$ consists of node-disjoint alternating paths and cycles $p_1, \ldots, p_k$. Let $M_0 = M_{min}$ and $M_i = M_{i-1} \oplus p_i$ for $i = 1, \ldots, k$. Each of these matchings is an $\mathcal{R}^\star$-matching, and $M_k = M_{max}$. As all value weights are 1 or 0 (for *undef*), we infer $weight(M_i) - weight(M_{i-1}) \in \{-1, 0, 1\}$ (cf. Figure 4.2 on page 77). Hence, for every $w \in Dom(W)$, there is a matching M_i in the sequence $M_0, \ldots, M_k$ with $weight(M_i) = w$. By Lemma 4.1, M_i corresponds to a solution of the constraint where w is assigned to W.

Let v be a value in the domain of an assignment variable X_i and assume that v is not pruned during the first iteration. Since $I(e) = [weight(M_{min}) + \underline{regret}(e); weight(M_{max}) - \overline{regret}(e)]$, this implies that there is some weight w in $I(e) \cap Dom(W)$ (see lines 16 and 33). By the definition of the lower and the upper regret, there are two $\mathcal{R}^\star$-matchings M_l and M_h in $G^\star$ such that both contain e and $I(e) = [weight(M_l); weight(M_h)]$. As every path in $M_l \oplus M_h$ avoids e, an analogous argument as above proves that there is an $\mathcal{R}^\star$-matching M in $G^\star$ with $e \in M$ and $weight(M) = w$. Hence, M corresponds to a solution where v is assigned to X_i.

As the domains of all variables are consistent after the first iteration, there can be no pruning in the second iteration. Only the matchings may change. Therefore the algorithm stops after the second iteration. □

4.3 Comparison with related work

To the best of our knowledge the *WPA*-constraint itself has not been treated before. In this section we will discuss some global constraints which are related with our work and summarize the results which have been obtained for them. As the name suggests, our constraint is strongly related with the classical *Alldiff*-constraint, which we discussed in Chapter 3. Of course, *WPA* is a generalization of *Alldiff*: If we choose for *undef* a value that is not contained in any variable domain, set W and all weights to zero,

then *WPA* becomes equivalent to an *Alldiff*-constraint on the assignment variables. There are several propagation algorithms which achieve different degrees of consistency (see [vH01a] for an overview). We compare our work with the results of Régin who proposed an arc-consistency algorithm [Rég94]. As before we denote by n the number of assignment variables and by m the sum of the cardinalities of their domains. Régin's algorithm, which is also based on matchings in bipartite graphs, has a worst case running time of $O(\sqrt{n}m)$. It is also incremental and has a best case running time of $\Theta(m)$. By Lemma 4.8, our propagation algorithm achieves the same consistency, however the worst case running time is $O(nm)$.

WPA is a generalization of a constraint called *alldiff_except_0*. The arguments of the constraint are n assignment variables and the constraint states that all variables must take pairwise distinct values except for those variables which are assigned the value 0. This constraint has been mentioned in [Bel00], there it was considered for applications like the ones we described in scenario three (on page 72). But as far as we know, no propagation algorithm has been proposed before. This constraint can be expressed by a *WPA*-constraint where *undef* is set to 0, and all weights are set to 1 (except for the value *undef*) and the initial domain of W is $[0; n]$. Using this translation, the user has the possibility to control the number of variables which are set to 0, by imposing constraints on W. By Lemma 4.9, we can achieve arc-consistency for the assignment variables in a worst-case running time of $O(nm)$.

Petit et al. [PRB01] also consider relaxations of the *Alldiff*-constraint. Their constraint involves n assignment variables $X_1, \ldots, X_n$ and a cost variable C. They discuss two different cost models. In the first model the cost of an assignment is defined to be n minus the number of distinct values. So if all values are pairwise distinct, the cost is 0. And the assignments $(1, 1, 2, 2)$ and $(2, 1, 2, 2)$ have both cost 2. The second model is based on a reformulation of the *Alldiff*-constraint as $\binom{n}{2}$ binary constraints of the form $X_i \neq X_j$ for $i \neq j$. And the cost in this model is simply the number of binary constraints which are violated by the assignment. If all variables take pairwise distinct values, the cost is 0 as before. The assignment $(1, 1, 2, 2)$ has cost 2, whereas the cost of $(2, 1, 2, 2)$ is now 3.

For both models they propose algorithms which prune the domains of the assignment variables according to a maximum cost. The algorithm for the first model is based on the computation of maximum cardinality matchings in the value graph. Its worst case running time is $O(\sqrt{n}m)$ and it achieves arc-consistency. The second model seems to be more difficult to handle. The algorithm for this model is based on flows. Its complexity is $O(n^2\sqrt{n}ms)$,

where s is the maximum cardinality of a variable domain. And it cannot guarantee arc-consistency.
We want to point out that we cannot emulate any of the two models with our constraint because we do not allow that any value different from *undef* is taken more than once in an assignment.

The next constraint that we discuss is called *SumOfWeightsOfDistinctValues* (abbreviated as *SWDV*). Beldiceanu et al. [BCT02] introduced this constraint. It takes as input n assignment variables $X_1, \ldots, X_n$, a value-weight table T and a weight variable W. In contrast to *WPA*, any value may be used several times in an assignment, but the weights may not be negative. The *SWDV*-constraint states that $W = \sum_{v \in \{X_1,\ldots,X_n\}} weight(v)$, i.e. W must be equal to the total weight of the variable assignment, but every value v contributes at most once to total weight, even if it is assigned to many X's.

It is easy to see that the maximum weight of a variable assignment is equal to the weight of a maximum weight matching in the corresponding weighted value graph. So the "upper side" of this constraint is very similar to *WPA*. In fact, the matching algorithm and the computation of the upper regrets that we have presented in the previous section have been derived from algorithms developed in [BCT02]. The "lower side" of *SWDV* is quite different from *WPA* and seems to be more difficult to handle (see [BCT02] for details).

In the sequel we compare with the constraint *MinWeightAllDiff* introduced by Caseau and Laburthe [CL97]. This constraint augments the classical *Alldiff*-constraints with costs. In addition to the n assignment variables, the constraint takes as input a cost table T and a cost bound β. The table T states for every variable X and every value $v \in Dom(X)$ the cost $c_{X,v}$ for assigning v to X. So in contrast to our constraint the cost of a value may depend on the variable to which it is assigned. The cost of an assignment $(v_1, \ldots, v_n)$ is defined as $\sum_{i=1}^{n} c_{X_i,v_i}$. *MinWeightAllDiff* holds if all assignment variables take pairwise distinct values and the cost of the assignment does not exceed β. So this constraint models a similar situation as in scenario one (see page 71). But it is not capable of modelling an *undef* value that may be used several times.[6]

Sellmann [Sel02] gave an arc-consistency algorithm for this constraint. It works on a weighted variable-value graph G, where the weights are associated with the edges. Checking feasibility of the constraint can be done by computing a minimum weight variable-perfect matching in G. And the pruning amounts to n single source shortest paths computations in an oriented

[6] As a work-around, one might introduce n distinct values $undef_1, \ldots, undef_n$ with weight 0 and add $undef_i$ to the domain of X_i.

variable-value graph. Thus the algorithm runs in time $O(n(m + k \log k))$, where $k = n + |\bigcup_{i=1}^{n} Dom(X_i)|$, i.e. k equals the number of nodes in G. The author mentions that it is not known how this algorithm can be implemented to run incrementally faster. So in a setting where the costs of the values are independent of the variables to which they are assigned, it is better to use the algorithms for the *WPA*-constraint, because they can achieve bound consistency in time $O(nm)$ (see Lemma 4.8), and they are incremental.

Finally, we want to mention the constraint *CostGCC* which has been described by Régin [Rég99]. It allows to state a cost bound for the global cardinality constraint (*GCC*), which we discussed in Section 3.2.3. Similar to *MinWeightAllDiff*, the input of the constraint consists of n assignment variables $X_1, \ldots, X_n$, a cost table T and a cost bound β. In addition, we have for every value v a lower and an upper capacity bound l_v and u_v. The constraint states each value v must be used at least l_v and at most u_v times in the assignment and that the cost of the assignment must not be more than β. Clearly, in the special case where all lower capacities are zero and all upper capacities are one, *CostGCC* is equivalent to *MinWeightAllDiff*.

Régin [Rég99] describes an arc-consistency algorithm for *CostGCC*. His algorithm is based on min-cost-flow (for feasibility checking) and single source shortest paths computations in a so-called residual graph (for pruning). It can be implemented to run in time $O(n(m + k \log k))$.[7]

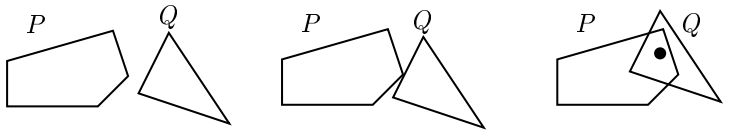
The constraint *CostGCC* can model scenario one as follows: we set all lower capacity bounds to zero, the upper capacity bound for the value *undef* is n and the upper bound for all other values is one. Régin's algorithm achieves the same pruning as our algorithms for the *WPA*-constraint, but the complexity is higher. There are cases where *CostGCC* is more general than *WPA*. But when this additional power is not needed, our algorithms give a better worst case complexity.

[7] So for the special case *MinWeightAllDiff*, Régin's algorithm has the same asymptotic complexity as the algorithm by Sellmann [Sel02].

Chapter 5

A non-overlapping constraint between convex polygons

This chapter discusses a non-overlapping constraint between two convex polygons[1]. We restrict our attention to the two-dimensional plane $\mathbb{R}^2$, but many statements can be generalized for higher dimensions. First, we want to explain what we mean by *non-overlapping*. Let P and Q be two sets of points (e.g. two polygons). If P and Q do not intersect at all (see left-hand side of Figure 5.1) or only their boundaries intersect (as in the middle of the figure), then P and Q are called *non-overlapping*. Only if P and Q have a common interior point (i.e. $\text{int}(P) \cap \text{int}(Q) \neq \emptyset$), we say that P and Q overlap (cf. right-hand side of the figure).



a) no intersection b) touching c) overlapping

Figure 5.1: Examples illustrating the definition of *overlapping*.

Determining whether two convex polygons intersect is a well-studied problem in computational geometry (see [dBvKOS00] for example). However, we want to deal with problems where the position of one or both polygons is not completely determined. We illustrate this with the example shown in Figure 5.2: We have two rectangles R_1 and R_2. The shape and the position of R_1 are fixed, the shape of R_2 is fixed too, but its position is not. The

[1] This and other relevant geometrical notions are formally defined in Section 2.3.

reference point of R_2, which is indicated by a dot in the figure, can be moved within a certain region. The coordinates of the reference point are given by two real-number[2] variables X_2 and Y_2 with domains $Dom(X_2) = [2, 5]$ and $Dom(Y_2) = [1, 3]$. Thus the reference point may be moved to any position in the rectangle $O_2 = Dom(X_2) \times Dom(Y_2)$. Our goal is to find out that it must not be placed in the interior of the shaded region of O_2, because this would cause R_1 and R_2 to overlap, i.e. we want to narrow the domain of X_2 to $[3, 5]$.

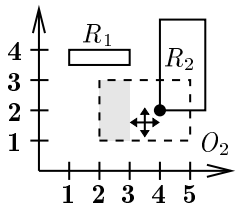


Figure 5.2: Example for a situation where one object is not fixed.

In general, the positions of both objects appearing in a *NonOverlapping*-constraint may be variable. Each object is modelled by a fixed shape polygon *Shp* and a (possibly) variable translation vector $t = (x, y) \in \mathbb{R}^2$. The actual object is obtained by applying the translation t to *Shp*, which we will denote by $t \oplus Shp$. For convenience, we assume that *Shp* contains the point $(0, 0)$, i.e. the origin of the coordinate system. In a translated copy $P = (x, y) \oplus Shp$, we call the point (x, y) the *origin of* P. So the relative position of the origin with respect to *Shp* is fixed. In our drawings we will mark this point by a dot (cf. R_2 in Figure 5.2). Geometrically, $(x, y) \oplus Shp$ is obtained by moving the dot into the point (x, y).

Our non-overlapping constraint has the following syntax:

$$NonOverlapping(Shp_1, (X_1, Y_1), OrgBnd_1; Shp_2, (X_2, Y_2), OrgBnd_2)$$

Each object is characterized by 4 parameters which have the following meaning:

- Shp_i is a convex polygon that describes the shape of the i-th object.
- (X_i, Y_i) is the translation vector for Shp_i, i.e. the position of its origin. Both X_i and Y_i are variables whose domains are closed intervals in $\mathbb{R}$.

[2] In the paper [BGT01] we considered finite domain integer variables. But since the coming theory is more easily formulated for real numbers, we chose domains of real numbers. Other domains will be discussed in Section 5.5.

- The origin boundary $OrgBnd_i$ is a convex polygon that (further) restricts the placement of the origin of the i-th object. We require that (X_i, Y_i) must be a point in $OrgBnd_i$.

Thus the origin of the i-th object must be a point in the *domain polygon* $Domain_i = (Dom(X_i) \times Dom(Y_i)) \cap OrgBnd_i$, which is convex again. The reader may wonder why we have introduced $OrgBnd_i$. The reason is that $Dom(X_i) \times Dom(Y_i)$ is always an axis-parallel rectangle, and hence, offering the origin boundary as additional restriction increases the modelling capabilities of the constraint.

The constraint holds if $(X_1, Y_1) \oplus Shp_1$ and $(X_2, Y_2) \oplus Shp_2$ do not overlap and the origins are contained in the respective domain polygons. The relation $\mathcal{S}$ of the above mentioned constrained contains all tuples (x_1, y_1, x_2, y_2) with the following properties:

- $\mathrm{int}((x_1, y_1) \oplus Shp_1) \cap \mathrm{int}((x_2, y_2) \oplus Shp_2) = \emptyset$
- $(x_i, y_i) \in Domain_i$ for $i \in \{1, 2\}$.

This constraint can be applied to all kinds of two-dimensional placement problems, e.g. designing a pattern for cutting cloth, or laying out parts on a sheet of metal [CF94]. The reader may wonder if these problems can be modelled well without allowing rotations of the shape polygon. Concerning the first example, we observe that cloth may be decorated with a pattern, thus the shapes that are to be cut out are often not allowed to be rotated, only to be translated. In the second example it may be possible in some cases to reduce the waste if rotations are possible. However, this may increase the time to solve the problem considerably. So in applications with limited computation time, a restricted model (without rotations) that yields good solutions quickly may be preferable. This might be the reason why rotations have not been taken into account in [CF94].

The work discussed in this chapter is based on the paper [BGT01], which is joint work with Nicolas Beldiceanu and Qi Guo. The chapter is organized as follows. We start with a rough overview of the propagation algorithm. Then we introduce the overlapping polygon, which will play a key role in the algorithm. We describe its main properties and discuss how to compute it. After that we present a narrowing algorithm for the variable domains, and we analyse its running time. Then we sketch some possible extensions of the constraint. Finally, we conclude with the discussion of related work.

5.1 Overview of the algorithm

In the sequel we develop a propagation algorithm that can narrow the domains of the origin variables to bound-consistency. Throughout this chapter we will use $\mathbb{R}$ as number type. But the results will also be valid for rational numbers. It will turn out that our algorithm needs to perform the following operations on numbers: the arithmetic operations "$+, -, \cdot, /$" and comparisons. In order to make the analysis easier, we assume that each of these basic number operations can be done in constant time. (We know that these assumptions are not realistic for exact computations with real or rational numbers, some of the issues will be discussed later in Section 5.5.) Given these assumptions we will show that the running time of the algorithm is linear in the total number of vertices of the input polygons.

As mentioned above, our algorithm does not deal with two fixed polygons but rather with two *families* $\mathcal{F}_1$ and $\mathcal{F}_2$ of polygons:

$$\mathcal{F}_i = \{(x, y) \oplus \mathit{Shp}_i \mid (x, y) \in \mathit{Domain}_i\}$$

Suppose we want to place the origin of the shape polygon of $\mathcal{F}_1$. Our algorithm can be outlined as follows:

- Compute Domain_1 and Domain_2:
  In order to determine Domain_i, we have to compute the intersection of the origin boundary OrgBnd_i with the axis parallel rectangle spanned by $(\underline{X}_i, \underline{Y}_i)$ and $(\overline{X}_i, \overline{Y}_i)$. This problem is well-known in computer graphics as "polygon clipping", an efficient algorithm for convex polygons was given by Sutherland and Hodgman [SH74]. (A very readable presentation of this algorithm together with other clipping algorithms can be found in [FvDFH90].)

- Compute the overlapping polygon $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)$:
  The overlapping polygon will be discussed in detail in Section 5.2. The crucial property of this polygon is that its interior points are exactly the forbidden placements of Shp_1 with respect to $\mathcal{F}_2$. A placement (x_1, y_1) is *forbidden* if $(x_1, y_1) \oplus \mathit{Shp}_1$ overlaps **every** member of $\mathcal{F}_2$.

- Examine $\mathit{Pl}_1 = \mathit{Domain}_1 \setminus \mathrm{int}(\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2))$:
  The set Pl_1 contains all admissible placements for the origin of Shp_1. With a sweepline algorithm (see Section 5.3) we can narrow the domains of X_1 and Y_1 to bound consistency.

In the sequel we will give some details of the algorithm. We will use the following example to illustrate it:

Example. The two objects that we want to place are described by the following parameters:

- $Shp_1 = \langle(4,-4),(-2,-2),(0,2)\rangle$,
  $Dom(X_1) = [5,9], Dom(Y_2) = [1,6]$,
  $OrgBnd_1 = \langle(5,4),(9,0),(11,4),(8,7)\rangle$

- $Shp_2 = \langle(0,0),(5,0),(7,2),(6,5),(0,3)\rangle$,
  $Dom(X_2) = [-2,5], Dom(Y_2) = [-1,4]$,
  $OrgBnd_2 = \langle(9,2),(1,4),(-1,-1),(1,-4)\rangle$

The shape polygons are depicted in Figure 5.3. The algorithm starts with clipping origin boundaries against the rectangles which are induced by the variable domains (see Figure 5.4). Clipping $OrgBnd_1$ against the rectangle with the corners $(5,1)$ and $(9,6)$ yields $Domain_1 = \langle(5,4), (8,1), (9,1), (9,6), (7,6)\rangle$. For the second family we obtain $Domain_2 = \langle(5,3), (1,4), (-1,-1), (5,-1)\rangle$.

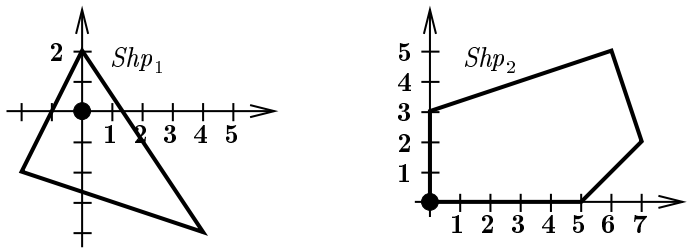


Figure 5.3: The shape polygons in our running example.

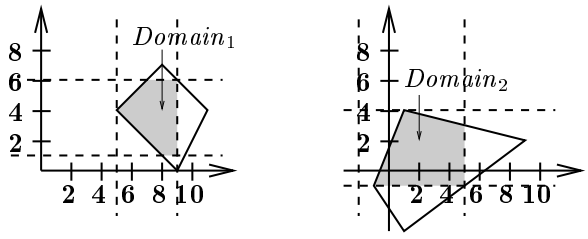


Figure 5.4: Clipping of the origin boundaries for our running example.

5.2 The overlapping polygon

In this section we tackle the problem of finding the forbidden placements for the origin of Shp_1 with respect to $\mathcal{F}_2$. We start with a slightly easier problem. We consider a fixed instance $P_2 \in \mathcal{F}_2$, and ask ourselves which placements are forbidden with respect to P_2. This problem is a subproblem in robot motion planning where one has to move a polygonal-shaped robot such that it avoids polygonal-shaped obstacles (see [dBvKOS00, Chapter 13]). Our "robot" is Shp_1 and our "obstacle" is P_2. But there is a subtle difference between the two problems: In robot motion planning the robot is not allowed to touch the obstacle, whereas Shp_1 may touch P_2 but not overlap. However, we shall see that the results from motion planning carry over to our problem with minor modifications.

Let $F = \{(x, y) \,|\, ((x, y) \oplus Shp_1) \cap P_2 \neq \emptyset\}$, i.e. F corresponds to all placements where Shp_1 and P_2 touch or overlap. Geometrically the boundary of F can be obtained by sliding Shp_1 along the boundary of P_2 and tracing the origin of Shp_1 (see Figure 5.5). Moreover, we see that the boundary of F (denoted by ∂F) consists of the placements where Shp_1 and P_2 just touch, and the interior of F (i.e. $\text{int}(F)$) contains the placements where they overlap.

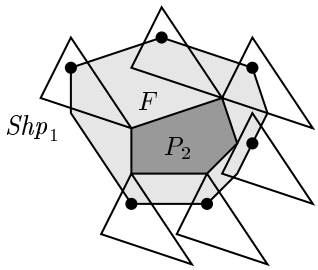


Figure 5.5: The forbidden placements for Shp_1 wrt. a fixed instance P_2.

In order to describe F in a way which will facilitate its computation, we use the notion of a *Minkowski sum*: The Minkowski sum of two sets $P, Q \subseteq \mathbb{R}^2$ is denoted by $P \oplus Q$ and defined as follows: $P \oplus Q = \{p+q \,|\, p \in P, q \in Q\}$. (Observe that $\{(x, y)\} \oplus Shp_1 = (x, y) \oplus Shp_1$.) Moreover, for $\lambda \in \mathbb{R}$ we define $\lambda \cdot P := \{\lambda \cdot p \,|\, p \in P\}$. It will turn out that $F = P_2 \oplus -Shp_1$, but before we show this, we list some fundamental properties of Minkowski sums and their interior points, which will be used to prove the subsequent lemmas.

Lemma 5.1 *For two sets $P, Q \subseteq \mathbb{R}^2$ the following holds:*

- $\text{int}(p \oplus P) = p \oplus \text{int}(P)$, $\forall\, p \in \mathbb{R}^2$.
- $\text{int}(\lambda \cdot P) = \lambda \cdot \text{int}(P)$, $\forall\, \lambda \in \mathbb{R} \setminus \{0\}$.
- $\text{int}(P \oplus Q) = \text{int}(P) \oplus \text{int}(Q)$, *if* P *and* Q *are convex with non-empty interior.*

For a convex set $Q \subseteq \mathbb{R}^2$, $p_1, \ldots, p_n \in \mathbb{R}^2$ *and* $\lambda_1, \ldots, \lambda_n \in [0,1]$ *with* $\sum_{i=1}^{n} \lambda_i = 1$ *the following holds:*

$$(\sum_{i=1}^{n} \lambda_i p_i) \oplus Q = \bigoplus_{i=1}^{n} \lambda_i (p_i \oplus Q)$$

Proof. The proof of the first two statements is straightforward. The last two statements are shown in Section 2.3 (cf. Lemmas 2.5 and 2.6). □

The following lemma characterizes the forbidden placements of a shape polygon with respect to a fixed polygon, i.e. we show $F = \text{int}(P_2 \oplus -Shp_1)$.

Lemma 5.2 *A member* $t \oplus Shp$ *of a family* $\mathcal{F}$ *of convex polygons overlaps a convex polygon* P *iff* $t \in \text{int}(P \oplus -Shp)$.

Proof.

$$\begin{array}{lll} \text{int}(t \oplus Shp) \cap \text{int}(P) \neq \emptyset & \iff & \exists\, s \in \text{int}(Shp), \exists\, p \in \text{int}(P) : t + s = p \\ & \iff & \exists\, s \in \text{int}(Shp), \exists\, p \in \text{int}(P) : t = p - s \\ & \overset{\text{L 5.1}}{\iff} & t \in \text{int}(P \oplus -Shp) \end{array}$$

□

Now we pick up our original problem. Recall that a placement t is forbidden with respect to a family $\mathcal{F}_2$ iff $t \oplus Shp_1$ overlaps every member of the family. By the Lemma above this is the case iff $t \in \bigcap_{P_2 \in \mathcal{F}_2} \text{int}(P_2 \oplus -Shp_1)$. The problem with this characterization is that $\mathcal{F}_2$ may have infinitely many members, which makes it hard to compute this intersection. In the sequel we will show that it suffices to consider only a finite number of members of $\mathcal{F}_2$. Every member of $\mathcal{F}_2$ can be written as $v \oplus Shp_2$ with $v \in Domain_2$. Let $\langle v_1, \ldots, v_n \rangle$ be the vertices of $Domain_2$. We call $v_1 \oplus Shp_2, \ldots, v_n \oplus Shp_2$ the *extreme members* of $\mathcal{F}_2$ and denote them by $Extr(\mathcal{F}_2)$. Using the convexity of the shape polygons and the fact that every point in $Domain_2$ can be written as a convex combination of its vertices, we will prove the lemma below. It states that $t \oplus Shp_1$ overlaps all members of $\mathcal{F}_2$ iff it overlaps all of its extreme members.[3]

[3] The lemma also holds for families with a non-convex domain polygon, but the shape polygon has to be convex.

Lemma 5.3 *A convex polygon S_1 overlaps all members of a family $\mathcal{F}_2$ of convex polygons iff S_1 overlaps all extreme members of $\mathcal{F}_2$.*

Proof. Suppose that S_1 overlaps all polygons in $Extr(\mathcal{F}_2)$, we will show that it overlaps every member $v \oplus Shp_2$ of $\mathcal{F}_2$. (The other direction of the statement is trivial.)
We can find extreme members $v_1 \oplus Shp_2, \ldots, v_k \oplus Shp_2$ of $\mathcal{F}_2$ such that $v = \sum_{i=1}^{k} \lambda_i v_i$ with $\lambda_1, \ldots, \lambda_k \in]0, 1]$ and $\sum_{i=1}^{k} \lambda_i = 1$. Since S_1 overlaps all extreme instances, there exists $s_i \in \mathrm{int}(S_1 \cap v_i \oplus Shp_2)$ for $i = 1, \ldots, k$. It is easy to see that $s = \sum_{i=1}^{k} \lambda_i s_i$ is in $\mathrm{int}(S_1)$, because s is a convex combination of interior points and S_1 is convex. Applying Lemma 5.1, we obtain $s \in \bigoplus_{i=1}^{k} \lambda_i \cdot \mathrm{int}(v_i \oplus Shp_2) = \mathrm{int}(\bigoplus_{i=1}^{k} \lambda_i \cdot (v_i \oplus Shp_2)) = \mathrm{int}((\sum_{i=1}^{k} \lambda_i v_i) \oplus Shp_2) = \mathrm{int}(v \oplus Shp_2)$. Thus S_1 and $v \oplus Shp_2$ overlap. □

Now we can define the overlapping polygon as a finite intersection of Minkowski sums: $Overlap(Shp_1, \mathcal{F}_2) := \bigcap_{P_2 \in Extr(\mathcal{F}_2)} (P_2 \oplus -Shp_1)$. The theorem below states that the interior points of the overlapping polygon are exactly the forbidden placements of Shp_1 with respect to $\mathcal{F}_2$.

Theorem 5.1 *A convex polygon $t \oplus Shp_1$ overlaps all instances of a family $\mathcal{F}_2$ of convex polygons iff $t \in \mathrm{int}(Overlap(Shp_1, \mathcal{F}_2))$.*

Proof.

$$
\begin{array}{rl}
 & t \in \mathrm{int}(Overlap(Shp_1, \mathcal{F}_2)) \\
\iff & t \in \mathrm{int}(\bigcap_{P_2 \in Extr(\mathcal{F}_2)} P_2 \oplus -Shp_1) \\
\iff & t \in \bigcap_{P_2 \in Extr(\mathcal{F}_2)} \mathrm{int}(P_2 \oplus -Shp_1) \\
\overset{\text{L 5.2}}{\iff} & t \oplus Shp_1 \text{ overlaps all extreme members of } \mathcal{F}_2 \\
\overset{\text{L 5.3}}{\iff} & t \oplus Shp_1 \text{ overlaps all members of } \mathcal{F}_2
\end{array}
$$

□

As an immediate consequence of the theorem above, we obtain the corollary below. We will use it to show that our narrowing algorithm achieves bound-consistency:

Corollary 5.1 *Let $Pl_1 = Domain_1 \setminus \mathrm{int}(Overlap(Shp_1, \mathcal{F}_2))$ and let $\mathcal{S}$ denote the set of all solutions of the non-overlapping constraint (cf. page 113). Define $\pi_{x_1,y_1}(\mathcal{S}) := \{(x_1, y_1) \mid \exists\, (x_1, y_1, x_2, y_2) \in \mathcal{S}\}$, i.e. $\pi_{x_1,y_1}(\mathcal{S})$ is the projection of $\mathcal{S}$ onto its first two components. Then $Pl_1 = \pi_{x_1,y_1}(\mathcal{S})$.*

Computation of the overlapping polygon

We discuss how to compute $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)$ efficiently for a convex shape polygon Shp_1 and a family $\mathcal{F}_2$ of convex polygons. If $v_1 \oplus \mathit{Shp}_2, \ldots, v_n \oplus \mathit{Shp}_2$ are the extreme members of $\mathcal{F}_2$, then the overlapping polygon looks as follows:

$$\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2) = \bigcap_{i=1}^{n}((v_i \oplus \mathit{Shp}_2) \oplus -\mathit{Shp}_1) = \bigcap_{i=1}^{n}(v_i \oplus (\underbrace{\mathit{Shp}_2 \oplus -\mathit{Shp}_1}_{=:S}))$$

The computation proceeds in two steps: First we compute the Minkowski sum $S := \mathit{Shp}_2 \oplus -\mathit{Shp}_1$, and then we intersect $v_1 \oplus S, \ldots, v_n \oplus S$. Observe that we deal with an intersection of translated copies of the same polygon, which – as we will see – is easier to compute than the intersection of n arbitrary polygons.

Computing the Minkowski sum of two convex polygons

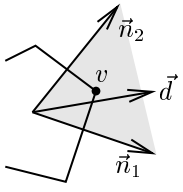
In the sequel we will present an algorithm that can compute the Minkowski sum of two convex polygons $P = \langle p_1, \ldots, p_h \rangle$ and $Q = \langle q_1, \ldots, q_k \rangle$ in time $O(h+k)$. Our presentation follows [dBvKOS00, Chapter 13.3], we include it here for the sake of completeness and to introduce some notions needed in the next section. Looking at Figure 5.6 we can make an important observation:

> An extreme point in direction $\vec{d}$ on $P \oplus Q$ is the sum of extreme points in direction $\vec{d}$ on P and Q, and vice versa.

As we can also see in the figure, there are two cases for the extreme points on a convex polygon in a direction $\vec{d}$: Either there is a single point, which is a vertex, or there is a whole edge, which happens if the outer normal of the edge has the same direction as $\vec{d}$. Thus adding the extreme points of P and Q in direction $\vec{d}$ either yields a vertex or an edge of $P \oplus Q$. And a vertex of the Minkowski sum is obtained iff both P and Q have a single extreme point in direction $\vec{d}$.

So in order to compute the vertices of $P \oplus Q$ we use a polar sweep algorithm. This means the following: We start with $\vec{d}$ pointing in the direction of the positive x-axis and then we rotate it counter-clockwise until we reach the positive x-axis again.

During the sweep we store the vertices of P and Q which are currently extreme in direction $\vec{d}$. As we can see on the right-hand side, a vertex is extreme for any direction in the interior of the cone spanned by the outer normals of its incident edges. And hence, the current extreme vertex changes whenever we sweep "over" an outer normal.



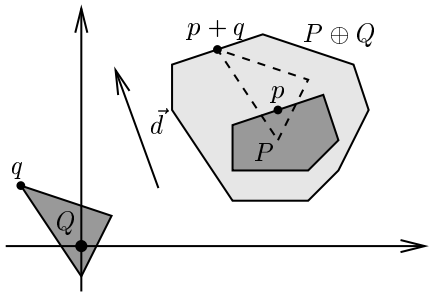
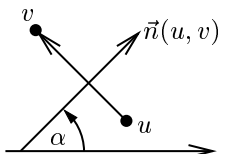


Figure 5.6: Minkowski sum of two convex polygons P and Q.



For two vertices $u = (u_x, u_y)$ and $v = (v_x, v_y)$ the outer normal vector $\vec{n}(u, v)$ of the directed edge from u to v is $(v_y - u_y, u_x - v_x)$. For a vector $\vec{d}$ we define $\angle \vec{d}$ to be the counter-clockwise angle between the positive x-axis and $\vec{d}$, and we require $\angle \vec{d} \in]0, 2\pi]$. (It may seem odd to exclude 0, but this is convenient for the algorithms we describe below.) If we visit the vertices of a convex polygon in counter-clockwise order and start with the lexicographically largest vertex, then the angles of the corresponding normals increase monotonously in the interval $]0, 2\pi]$. Algorithm 5.1, which computes the sum of two convex polygons is an immediate consequence of the ideas discussed above.

Algorithm 5.1 Minkowski sum $P \oplus Q$ of two convex polygons

Procedure: MinkowskiSum($P = \langle p_1, \ldots, p_h \rangle$, $Q = \langle q_1, \ldots, q_k \rangle$)
Require: P,Q are convex, the vertices in both lists are in counter-clockwise order with p_1 and q_1 having lexicographically largest coordinates

1: $p_{h+1} \leftarrow p_1$; $q_{k+1} \leftarrow q_1$
2: $i \leftarrow 1$; $j \leftarrow 1$ // we swept over the positive x-axis
3: **repeat**
4: add $p_i + q_j$ as vertex of $P \oplus Q$
5: **case 1:** $\angle \vec{n}(p_i, p_{i+1}) < \angle \vec{n}(q_j, q_{j+1})$
6: $i \leftarrow i + 1$ // we swept over $\vec{n}(p_i, p_{i+1})$
7: **case 2:** $\angle \vec{n}(p_i, p_{i+1}) > \angle \vec{n}(q_j, q_{j+1})$
8: $j \leftarrow j + 1$ // we swept over $\vec{n}(q_j, q_{j+1})$
9: **case 3:** $\angle \vec{n}(p_i, p_{i+1}) = \angle \vec{n}(q_j, q_{j+1})$ // parallel edges
10: $i \leftarrow i + 1$; $j \leftarrow j + 1$ // we swept over $\vec{n}(p_i, p_{i+1})$ and $\vec{n}(q_j, q_{j+1})$
11: **until** $i = h + 1$ and $j = k + 1$

We want to point out that the algorithm does not have to compute the angles of the two directions $\vec{d} = (d_x, d_y)$ and $\vec{e} = (e_x, e_y)$ explicitly in order to compare them. In most cases it suffices to check the sign of $e_x d_y - e_y d_x$, which is the third component of the cross product $\vec{e} \times \vec{d}$.[4] If the sign is positive (negative) then $\angle \vec{d}$ is greater (smaller) than $\angle \vec{e}$. Only if the sign is zero, which means the vectors point in the same or in opposite directions, we have to be careful. But then we test whether the two directions point into the same quadrant of the coordinate system.
Assuming that a basic numerical operation takes only constant time, we obtain a running time of $O(h + k)$ for the computation of the Minkowski sum.

Example. We give an example illustrating the computation of the Minkowski sum. We consider the two shape polygons from our running example:

$$\begin{array}{rcl} P & = & \mathit{Shp}_2 \\ Q & = & -\mathit{Shp}_1 \end{array} \begin{array}{l} = \langle (7,2),(6,5),(0,3),(0,0),(5,0) \rangle \\ = \langle (2,2),(-4,4),(0,-2) \rangle \end{array}$$

On the left-hand side of Figure 5.7, the two polygons and the outer normal vectors of their edges are depicted. In the middle we see a sphere that represents all possible two-dimensional directions. Our algorithm (conceptually) sweeps over all these directions in counter-clockwise order (starting with the direction of the positive x-axis) and looks at vertices of P and Q that are extreme in the current direction. The normal vectors divide the sphere into several sectors of directions. Within each sector the extreme vertices of both P and Q do not change, but when the sweep bypasses the border of a sector the extreme vertex of at least one of the two changes. Thus each sector corresponds to one vertex of the sum, and vice versa. (This correspondence is also shown in the middle part of the figure.) On the right-hand side of the figure, we see the resulting polygon $S = P \oplus Q$:

$$S = \langle (9,4),(8,7),(2,9),(-4,7),(-4,4),(0,-2),(5,-2),(7,0) \rangle$$

Computing the intersection

Now we know how to compute the vertices $s_1, \ldots, s_m$ of $S = \mathit{Shp}_2 \oplus -\mathit{Shp}_1$ in linear time. In the sequel we complete the computation of the overlapping polygon $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2) = \bigcap_{i=1}^{n} (v_i \oplus S)$. Since S is convex, it can be written as the intersection of half-planes $H_1, \ldots, H_m$. H_j lies to the left of the oriented line $\{s_j + \lambda(s_{j+1} - s_j) : \lambda \in \mathbb{R}\}$ (with $s_{m+1} := s_1$). And

[4] Here we see $\vec{d}$ and $\vec{e}$ as three-dimensional vectors with zero z-component.

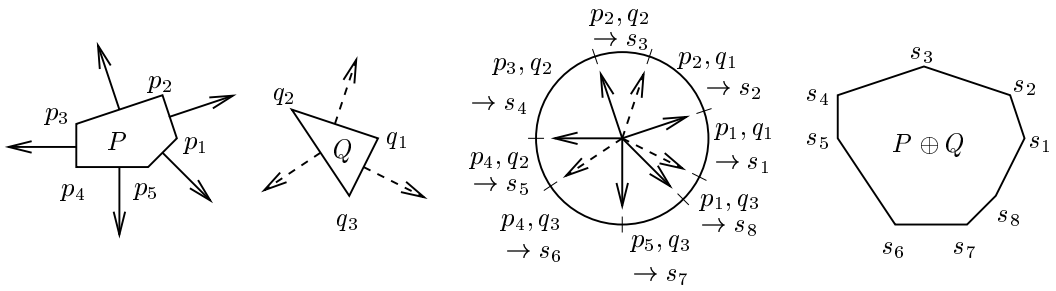


Figure 5.7: Example illustrating the computation of a Minkowski sum.

hence, $S = \bigcap_{j=1}^{m} H_j$, which implies $Overlap(Shp_1, \mathcal{F}_2) = \bigcap_{j=1}^{m} \bigcap_{i=1}^{n} (v_i \oplus H_j)$. Suppose we fix some j, then $v_1 \oplus H_j, \ldots, v_n \oplus H_j$ is a sequence of parallel half-planes. Thus there is some half-plane $v_{e_j} \oplus H_j$ which is contained in all half-planes in the sequence, i.e. $\bigcap_{i=1}^{n} (v_i \oplus H_j) = v_{e_j} \oplus H_j$. We call $R_j = v_{e_j} \oplus H_j$ a *relevant half-plane*. How do we determine v_{e_j}? Looking at Figure 5.8, we see that v_{e_j} can be found by sliding H_j in direction $-\vec{n}(s_j, s_{j+1})$, which is the inner normal of the edge $\overline{s_j s_{j+1}}$. The last vertex of the domain polygon $Domain_2$ that we hit during the slide is the desired vertex v_{e_j}. This means v_{e_j} is an extreme vertex in direction $-\vec{n}(s_j, s_{j+1}) = \vec{n}(s_{j+1}, s_j)$.

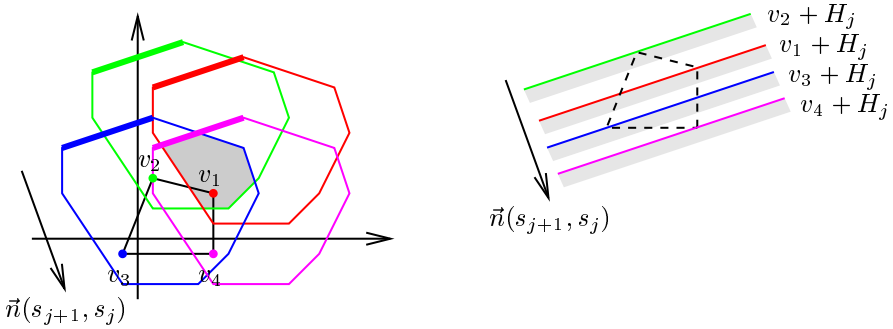


Figure 5.8: Finding the relevant half-planes of the intersection.

Algorithm 5.2, which computes the relevant half-planes, performs a polar sweep that is very similar to the computation of the Minkowski sum. We have to take care of the fact that we are looking at inner normal vectors of the edges of S, thus we have to start with the lexicographically smallest vertex of S. It is easy to see that the running time is $O(n+m)$, if we assume again that a basic number operation takes time $O(1)$.

Now we have reduced the intersection of $n \cdot m$ half-planes to an intersection

Algorithm 5.2 Computing the relevant half-planes

Procedure: RelevantHalfPlanes($D_2 = \langle v_1, \ldots, v_n \rangle$, $S = \langle s_1, \ldots, s_m \rangle$)
Require: D_2,S are convex, the vertices in both lists are in counter-clockwise order with v_1 and s_1 having lexicographically largest coordinates
1: $v_{n+1} \leftarrow v_1$; $s_{m+1} \leftarrow s_1$
2: $i \leftarrow 1$
3: **for all** vertices s_j of S (in ccw-order starting with the lex. smallest) **do**
4: **while** $\angle \vec{n}(v_i, v_{i+1}) < \angle \vec{n}(s_{j+1}, s_j)$ **do**
5: $i \leftarrow i + 1$
6: $v_{e_j} \leftarrow v_i$; $R_j \leftarrow v_{e_j} \oplus H_j$
7: **end for**

of m relevant half-planes: $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2) = \bigcap_{j=1}^{m} R_j$. It is well known that this can be computed in time $O(m \log m)$ (cf. [dBvKOS00, Chapter 4.2]). But we can take advantage of the fact that the half-planes $R_1, \ldots, R_m$ are ordered, because the angles of their outer normal vectors (which coincide with normal vectors of $H_1, \ldots, H_m$) increase (strictly) monotonously. We will compute the intersection iteratively, i.e. for $k = 2, \ldots, m$ we consider $I_k = \bigcap_{j=1}^{k} R_j$ and determine its boundary ∂I_k (if $\mathrm{int}(I_k) \neq \emptyset$). The boundary of the half-plane $R_j = v_{e_j} + H_j$ is the line $L_j = \{v_{e_j} + s_j + \lambda(s_{j+1} - s_j) \,|\, \lambda \in \mathbb{R}\}$. Each half-plane R_j can contribute at most one *boundary element* (i.e. a line segment or a ray) to the boundary of the intersection, and this contribution lies on L_j.

We represent ∂I_k by a list $\mathcal{B} = [B_1, \ldots, B_h]$ of boundary elements. Let us define the normal vector $\vec{n}(B)$ of a boundary element B to be the (outer) normal vector of the half-plane from which it originates. We will maintain the invariant that $\angle \vec{n}(B_1) < \ldots < \angle \vec{n}(B_h)$ and that successive boundary elements share a common endpoint. Moreover, if I_k is unbounded, then B_1 and B_h are (non-intersecting) rays and $B_2, \ldots, B_{l-1}$ are line segments; and if I_k is bounded, then it is a polygon and the elements in $\mathcal{B}$ are its edges.

The boundary of I_2 consists of two rays tailed at the intersection point of L_1 and L_2. Suppose that we want to compute the boundary of I_k and that the boundary of I_{k-1} is represented by $\mathcal{B} = [B_1, \ldots, B_h]$. In order to determine the position of a boundary element B in $\mathcal{B}$ relative to the half-plane R_k, we orient the line L_k (i.e. ∂R_k) such that $\mathrm{int}(R_k)$ lies to the left and the complement of R_k lies to the right. We direct L_k from $p = v_{e_k} + s_k$ towards $q = v_{e_k} + s_{k+1}$. The orientation of a point r with respect to L_k is

computed as follows:

$$\operatorname{sgn}\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{cases} +1 & : & r \text{ lies to the left} \\ 0 & : & r \text{ lies on } L_k \\ -1 & : & r \text{ lies to the right} \end{cases}$$

It suffices to look at the relative position of L_k and B (cf. Figure 5.9):

1. B (except for one endpoint possibly) lies to the right of L_k:
   Then $B \cap R_k$ is either empty, or it contains one endpoint of B that is also an endpoint of another boundary element B' in $\mathcal{B}$. Thus we can discard B. (Observe that if B' also lies to the right of L_k, then all boundary elements do and I_k is just a single point, which means that its interior is empty.)

2. B (except for one endpoint possibly) lies to the left of L_k:
   Then $B \subset R_k$, and hence B is contained in ∂I_k, and we keep it unchanged.

3. B lies on L_k:
   Since $\vec{n}(B)$ and $\vec{n}(R_k)$ cannot point in the same direction, they must be anti-parallel, thus $I_k = B$, i.e. $\operatorname{int}(I_k)$ is empty.

4. B and L_k intersect properly, i.e. in a single point, which is not an endpoint of B: Then we replace B with the part of B to the left of L_k.

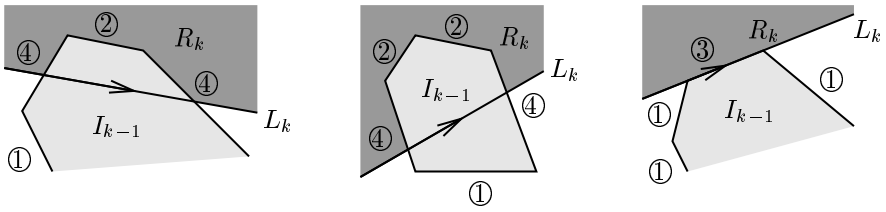


Figure 5.9: The possible cases when computing the boundary of $I_{k-1} \cap R_k$. (Next to each boundary element we indicate the case that applies.)

In order to update $\mathcal{B}$ we do not have to test all its elements against L_k, but we can proceed as follows: As long as the first or the last element of $\mathcal{B}$ lies to the right of L_k, we discard it. If $\mathcal{B}$ becomes empty or we encounter an element lying on L_k, we report that the interior of the overlapping polygon is empty and terminate. Let $[B_l, \ldots, B_r]$ be the sequence of the remaining elements. For both B_l and B_r we have that at least one endpoint lies to the

left of L_k. Observing that the angles of the normals of the elements increase monotonously and that $\vec{n}(R_k)$ is contained in the cone spanned by $\vec{n}(B_r)$ and $\vec{n}(B_l)$, we can conclude that $B_{l+1}, \ldots, B_{r-1}$ lie to the left of L_k. So we do not have to test them. For B_l and B_r we distinguish three cases:

- Both B_l and B_r lie to the left of L_k, which can only happen if I_{k-1} is bounded, then we are done.
- If B_l and B_r are distinct and L_k intersects both of them properly in p_l and p_r, we update B_l and B_r. Moreover, we append to $\mathcal{B}$ the segment $\overline{p_r p_l}$, which is the contribution of R_k to the boundary of I_k.
- If I_k is unbounded, we may have only one proper intersection p. Considering the arrangements of the normal vectors, it is easy to see that L_k must intersect B_r in that case. We update B_r, and append to $\mathcal{B}$ a ray that starts in p and has the same direction as L_k.

What remains to show is that the algorithm runs in time $O(m)$. With a constant number of arithmetic operations "$+, -, \cdot, /$" we can determine the relative position of a boundary element and an oriented line, and we can compute the intersection of an element and a line. The algorithm makes $m-1$ iterations. In every iteration it tests and discards some elements, it tests and possibly modifies at most two elements, which are not discarded, and at most one new element is added. So except for the time spent for removing elements each iteration takes constant time. As there can be at most m removals in total, the time bound of $O(m)$ follows.

Example. We take up our running example again and discuss the computation of the overlapping polygon $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)$. The different steps are illustrated in Figure 5.10. The boundary of I_2 consists of two rays that are tailed in the same point, the second ray is intersected by L_3 (see part 1). Thus the boundary of I_3 is obtained by updating the second boundary element and adding a ray that corresponds to the contribution of L_3. The computation for I_4 proceeds in analogous way, because L_4 properly intersects the last element of ∂I_3 (see part 2).
In part 3, we see that the last element of the boundary of I_4 lies to right of L_5, and hence it is discarded. Its predecessor element intersects L_5, so that it must be updated. Moreover, the first boundary element, which is a ray, also intersects L_5, which implies that I_5 is bounded (cf. part 4).
L_6 intersects the first and the last element of ∂I_5, and hence it contributes a line segment to the boundary of I_6 (see part 5). The same can be said about L_7 and the boundary of I_6. The last part of the figure shows I_7 and L_8. As can be verified by an easy computation, I_7 lies (slightly) to the left of L_8.

Therefore $I_8 = I_7$ and no boundary element needs to be updated.
So we obtain the following overlapping polygon:

$$\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2) = \langle (7,6), (4,7), (1.\overline{54}, 6.\overline{18}), (4.\overline{3}, 2), (6,2), (7.75, 3.75) \rangle$$

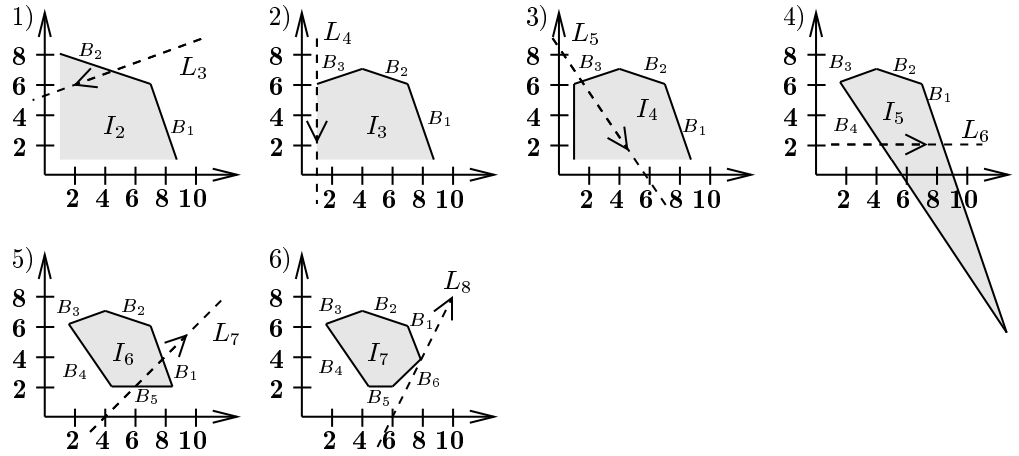


Figure 5.10: Computation of $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)$.

5.3 Narrowing the bounds of the origin variables

In this section we describe how to narrow the bounds of the origin variables of $\mathcal{F}_1$. In fact, we focus on the lower endpoint of X_1, but the procedures for the upper endpoint and for the endpoints of Y_1 are similar. So far we have determined Domain_1, it contains all placements of the origin of Shp_1 which are possible in principle, i.e. without regarding $\mathcal{F}_2$. Moreover, we have computed $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)$, whose interior consists of all placements that are forbidden because Shp_1 would overlap every member of $\mathcal{F}_2$. Thus $Pl_1 = \mathit{Domain}_1 \setminus \mathrm{int}(\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2))$ contains exactly the placements of Shp_1 which fulfil the constraint. In other words, Pl_1 is the projection of the solutions of the constraint on the first two components (see Corollary 5.1). Our task is to check that Pl_1 is non-empty, and if so, we narrow the lower endpoint of the domain of X_1, i.e. we determine $x_{min} = \min\{x \mid (x,y) \in Pl_1\}$. (Observe that we do not want to compute all the vertices of Pl_1.)

We use a plane sweep algorithm: Conceptually we move a vertical sweepline continuously from $\underline{X}_1$ to $\overline{X}_1$ until it hits a point p in Pl_1 for the first time. (As p is a leftmost admissible placement, x_{min} is equal to the x-coordinate of p.) We denote the sweepline at position x by L_x. During the sweep we maintain the edges of $Domain_1$ and $Overlap(Shp_1, \mathcal{F}_2)$ that intersect the current sweepline L_x. Let us assume first that there are no vertical edges, we will later discuss how to handle them. Since each of the two polygons is convex, L_x intersects each boundary in at most two points. Thus we can represent the status of the sweep with four edge pointers: *D_lower*, *D_upper*, *O_lower*, *O_upper*. We suppose that every (non-vertical) edge is half-open: its left vertex belongs to it, but the right one does not (it belongs to the other incident edge unless the right vertex is a rightmost vertex of the polygon).

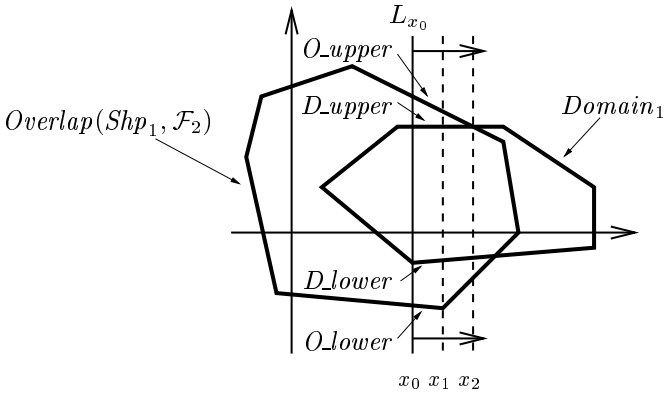


Figure 5.11: Status after moving the sweepline past x_0 (edge change event). (The next two events x_1 and x_2 are also indicated, where the latter is an intersection event.)

The initialization of our algorithm looks as follows: We determine x_{start}, which is the minimum x-coordinate of a vertex of $Domain_1$ (and may be larger than $\underline{X}_1$), and the two (non-vertical) edges *D_lower* and *D_upper* of $Domain_1$ that intersect $L_{x_{start}}$. Then we compute the edges *O_lower* and *O_upper* of the overlapping polygon intersecting $L_{x_{start}}$. If they do not exist or the overlapping polygon has a vertical edge at x_{start}, then $L_{x_{start}}$ does not intersect the interior of $Overlap(Shp_1, \mathcal{F}_2)$. Hence, we can stop immediately and report that $x_{min} = x_{start}$. Otherwise, we consider the intersections of $L_{x_{start}}$ with the four edges and compare their y-coordinates: We can terminate the sweep if $D\_lower|_{x_{start}} \leq O\_lower|_{x_{start}}$ or $D\_upper|_{x_{start}} \geq O\_upper|_{x_{start}}$. If this is not the case, then all points of $Domain_1$ with x-coordinate x_{start} are

contained in the interior of the overlapping polygon.

So we have to start the sweep in order to find the first position where the domain polygon leaves the interior of the overlapping polygon. This can only be a position where the two boundaries intersect. Thus our algorithm considers two types of events:

- intersection event:
  This event occurs when the sweepline L_x hits an intersection between *D_lower* and *O_lower* or between *D_upper* and *O_upper*. (Observe that we do not have to check for an intersection between a lower and an upper edge.) When this event occurs, we stop immediately and report $x_{min} = x$.

- edge change event:
  Whenever we sweep over a common vertex of two edges, we have to update the corresponding edge pointer: We let it point to the other incident edge of the vertex. Then we check whether this edge lies to the right of the sweepline. If not, we have reached the maximum x-coordinate of the polygon and we can terminate the sweep (see below).

With the four edge pointers we are able to determine easily the position x of the next event and to handle all the events that occur there. As long as no intersection occurs, we go on with the sweep until we reach a rightmost vertex of one of the two polygons. Let x_{end} denote the current position of the sweep line when this happens. First we consider the case that $Overlap(Shp_1, \mathcal{F}_2)$ ends at x_{end}. This means that *O_lower* and *O_upper* have a common vertex lying on $L_{x_{end}}$ or a vertical edge connecting *O_lower* and *O_upper* lies on the line. So in any case there is no interior point of the overlapping polygon on $L_{x_{end}}$. As $Domain_1$ does not end before x_{end}, we know that $Pl_1 \cap L_{x_{end}} \neq \emptyset$ and we report $x_{min} = x_{end}$. The other case is that the domain polygon ends strictly before the overlapping polygon, which implies that $Domain_1$ is contained in the interior of the overlapping polygon. Therefore, Pl_1 is empty and we report failure.

This suggests to handle the events at position x in the following order: first intersection events, then edge changes of the overlapping polygon, and finally edge changes of the domain polygon. So when we recognize the end of the domain polygon, we know that the overlapping polygon ends strictly later without checking this explicitly. Moreover, we want to point out that due to convexity a vertical edge is always extreme to the left or to the right. So we do not have to consider vertical edges after the initialization phase.

We want to analyse the running time of this sweepline algorithm. In the initialization phase we scan the vertex list of $Domain_1$ for the leftmost vertex

(or vertices if there is a parallel edge on the left-hand side), which allows us to set up *D_lower* and *D_upper*. This can be done in time $O(|Domain_1|)$. The initialization of *O_lower* and *O_upper* requires (in the worst case) to compute the intersection of $L_{x_{start}}$ with every edge of the overlapping polygon. This requires $O(|Overlap(Shp_1, \mathcal{F}_2)|)$ time. For the actual sweep we observe that handling an event and determining the next one can be done in constant time. Moreover, each vertex of either polygon gives rise to at most one edge change event; and there can be only one intersection event in total. Thus the overall running time of the sweep is $O(|Domain_1| + |Overlap(Shp_1, \mathcal{F}_2)|)$.

Example. We return to our running example and discuss how the sweep algorithm computes the narrowed lower endpoint for the variable X_1. The leftmost vertex of $Domain_1$ is $(5, 4)$, so we start the sweep at position $x_{start} = 5$. And the edges *D_lower* and *D_upper* are initialized to be the two edges incident to that vertex (see Figure 5.12). Then we scan the edge list of $Overlap(Shp_1, \mathcal{F}_2)$, and find out that $L_{x_{start}}$ intersects two of its edges, which provides us initial values for *O_lower* and *O_upper*. Moreover, the vertex $(5, 4)$ is nested between the intersections of $L_{x_{start}}$ with *O_lower* and *O_upper*, and it does not lie on a vertical edge of the overlapping polygon. Thus the vertex is contained in $\text{int}(Overlap(Shp_1, \mathcal{F}_2))$.
So we have to move the sweepline in order to determine x_{min}. The first event occurs at $x_1 = 6$. It is an edge change event, we have to update *O_lower*. The next event occurs because *O_lower* and *D_lower* intersect in the point $p = (6.5, 2.5)$ (see position x_2 in the Figure). Observe that p lies on the boundary of both polygons, and hence it is not an interior point of the overlapping polygon. Thus p is an admissible placement for the origin of Shp_1, and we can report $x_{min} = 6.5$.

On the right-hand side of Figure 5.12 we show that the placement p for the origin of Shp_1 is indeed admissible, i.e. there is a placement $q \in Domain_2$ such that $q \oplus Shp_2$ and $p \oplus Shp_1$ do not overlap. (In the example they just touch.)

5.4 Summary and total running time

In this section we briefly summarize the steps of the full propagation algorithm in order to analyse its total running time. In order to simplify notation let $n_i = |Shp_i|$ and $m_i = |OrgBnd_i|$. We will show now that the domain endpoints of X_1 and Y_1 can be narrowed to bound consistency in time $O(n_1 + n_2 + m_1 + m_2)$. By symmetry, the same result holds for X_2 and Y_2. We now consider the different steps of the algorithm:

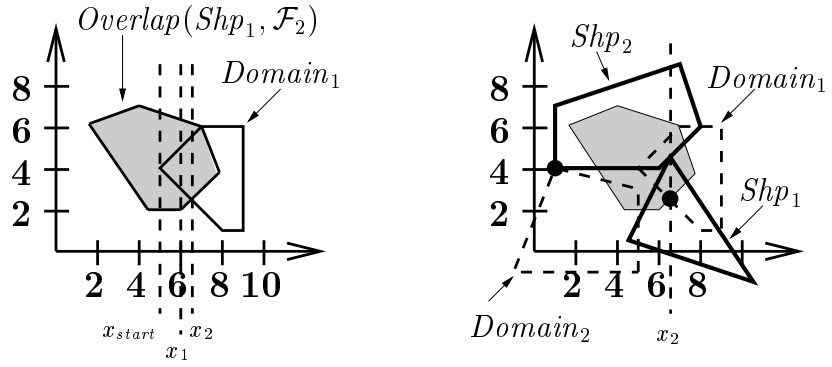


Figure 5.12: The left-hand side shows the events in the sweep that computes x_{min}; and the right-hand side demonstrates that an admissible placement is obtained at $x_{min} = x_2$.

- Computation of $Domain_i$:
  Since $OrgBnd_i$ is a convex polygon, it can be clipped against a rectangle in time m_i (see [SH74]). We observe that there can be at most two intersections between $OrigBnd_i$ and an edge of the clipping rectangle. It is easy to see that $|Domain_i| \leq m_i + 4$.

- Computation of $Overlap(Shp_1, \mathcal{F}_2)$:

  - Minkowski sum $S = Shp_2 \oplus -Shp_1$:
    As we have seen, S can be computed in time $O(n_1 + n_2)$, and we have $|S| \leq n_1 + n_2$.
  - Extremal vertices of $Domain_2$ wrt. the edges of S:
    These vertices (and hence the relevant half-planes for the overlapping polygon) can be determined in time $O(n_1 + n_2 + m_2)$
  - Intersection of the relevant half-planes:
    The computation takes time $O(n_1 + n_2)$.

  So the total time needed to calculate the overlapping polygon is $O(n_1 + n_2 + m_2)$, and this polygon has at most $n_1 + n_2$ vertices.

- Determining the narrowed endpoints (by sweeping over $Domain_1 \setminus \text{int}(Overlap(Shp_1, \mathcal{F}_2))$:
  We make four sweeps, each of them has a worst case running time of $O(m_1 + n_1 + n_2)$.

Putting everything together yields an overall running time for the propagation algorithm of $O(n_1 + n_2 + m_1 + m_2)$.

5.5 Extensions

Non-convex polygons

We begin with the case that both Shp_1 and Shp_2 are still convex, but at least one origin boundary polygon is not. As before we start with clipping the origin boundary, i.e. we compute $Domain_i = OrgBnd_i \cap \mathrm{Rect}(\underline{X}_i, \underline{Y}_i, \overline{X}_i, \overline{Y}_i)$. Since $OrgBnd_i$ is not convex, this is not so simple anymore, but it can still be done in time $O(m_i \log m_i)$ (see [dBvKOS00, Chapter 2.4]), where $m_i = |OrgBnd_i|$. Observe that the result of clipping is not necessarily single polygon, but may consist of several polygonal chains (see Figure 5.13). As every edge of $OrgBnd_i$ can intersect the clipping rectangle at most twice, we have $O(m_i)$ vertices after clipping.

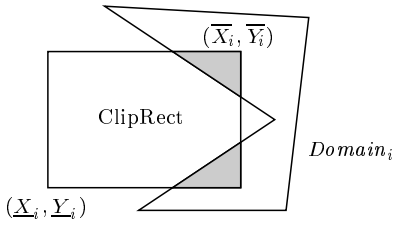


Figure 5.13: Clipping a non-convex polygon against a rectangle.

It will turn out that for computing $Overlap(Shp_1, \mathcal{F}_2)$ it suffices to adapt the definition of the extreme members of a family for non-convex domain polygons: We consider the convex hull of $Domain_2$ and define $Extr(\mathcal{F}_2)$ to be the members which are obtained by placing Shp_2 into the vertices of $\mathcal{CH}(Domain_2)$. The crucial observation is that with this definition Lemma 5.3 carries over literally. The proof is also the same too, because every point in $Domain_2$ is a convex combination of vertices of $\mathcal{CH}(Domain_2)$. So we determine $\mathcal{CH}(Domain_2)$ in time $O(m_2 \log m_2)$ (cf. [dBvKOS00, Chapter 1.1]), it is a polygon with $O(m_2)$ vertices. After that we can compute the overlapping polygon in the same way as described above.

What remains is the final step. We discuss how to examine the set of admissible placements for Shp_1 with a sweepline algorithm. Recall that $Pl_1 = Domain_1 \setminus Overlap(Shp_1, \mathcal{F}_2)$. The idea is basically the same as before, but the problem is now that a sweepline L_x may intersect more than two edges of $Domain_1$, because this polygon is in general not convex. So we cannot represent the status of the sweepline with four edge pointers any longer, but we have to use a more complicated data structure: We use a balanced binary tree that stores all edges that currently intersect L_x ordered according to the

y-coordinate of the intersection point. Since there are $O(m_1)$ such edges, we can perform an update on the tree in time $O(\log m_1)$.
Moreover, finding the next event is not as easy as before. The edge change events that are caused by the edges of the overlapping polygon can be found as before, because this polygon is still convex. In order to keep track of the edge change events that arise from edges of Domain_1, we sort the vertices of this polygon lexicographically. We will not go into detail how intersection events are determined, this is described for example in [dBvKOS00, Chapter 2.1]. What is important for us is that future intersection events are generated on-the-fly during the sweep. But since we can stop the sweep when the sweepline hits an intersection for the first time, we only have to keep the nearest intersection event even if several such events are discovered during the sweep. As $|\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)| = n_1 + n_2$, the whole sweep can be done in time $O((m_1 + n_1 + n_2) \log m_1)$. And hence, the overall running time for the propagation algorithm is $O((m_1+m_2+n_1+n_2) \log(m_1+m_2))$. So we only have to pay a logarithmic factor if the origin boundary polygons are not convex. Moreover, it is clear that for this case we also achieve bound-consistency.

The situation becomes more difficult when we consider non-convex shape polygons. We can observe that many of our statements do not hold any more in that case. What we can do is the following: We divide each shape polygon into convex polygons (for example by triangulating them), i.e. $\mathit{Shp}_i = \bigcup_{j=1}^{k_i} \mathit{Shp}_{ij}$. For each pair $(\mathit{Shp}_{1j}, \mathit{Shp}_{2j'})$ we have a non-overlapping constraint. Moreover, we need constraints stating that for each i the origins of Shp_{i1}, Shp_{i2}, ... are equal. We want to point out that this approach may lead to poor propagation and bound-consistency cannot be guaranteed.

A note on the variable domains

In the discussion above we have assumed that the domains of the variables are closed intervals of real numbers. Moreover, we have supposed that our algorithms can perform the basic arithmetic operations "$+, -, \cdot, /$" and comparisons on these numbers in constant time. If the hardware-supported floating point numbers are used for the computations, then these time assumptions hold. Clearly, floating point numbers cannot represent all reals and due to rounding errors the computations are not exact. Thus the algorithm may prune the variable domains too much, i.e. it can remove solutions of the constraint.

The accuracy problem can be solved by using rational numbers: Each number is represented by a numerator and a denominator, which are integers of arbitrary length. Then we can perform exact computations (as long as

all results fit into memory), but a single operation cannot be performed in constant time. The running time of an operation depends on the size of its arguments, i.e. the number of bits that are required to represent the arguments. Observe that applying an arithmetic operation to two rational numbers of size s_1 and s_2 respectively may yield a result of size $\Theta(s_1 + s_2)$. And hence, rational numbers can slow down the computation considerably.

A constraint programming system might offer the user two implementations of the propagation algorithm, one with floating point and one with rational numbers. Thus the user can choose between speed and accuracy. (Holzbaur has chosen this approach for his clp(Q,R) library [Hol95], which provides – among other things – support for linear equations over real and rational valued variables in Prolog.)

Finally, we address some issues related to finite integer domains, i.e. each domain is a finite set of integers. A possible way to deal with these domains is to treat them as if they were continuous, which means every discrete domain $D = \{d_1, \ldots, d_k\}$ is replaced by the interval $D' = [\min D, \max D]$. Then we apply our propagation algorithm to the interval domains, which may narrow D' to the interval $D'' = [a, b]$. We can prune every value d in the original D that violates $\lceil a \rceil \leq d \leq \lfloor b \rfloor$. This is the basic approach suggested in [BGT01]. Clearly, no solution is lost this way, but no consistency can be guaranteed.

We discuss how to achieve better pruning. Assume that each domain D is a range of integers, i.e. $D = [a..b] = \{a, a+1, \ldots, b\}$ with $a, b \in \mathbb{Z}$. Observe that $\mathit{Domain} = (\mathit{Dom}(X) \times \mathit{Dom}(Y)) \cap \mathit{OrgBnd}$ is not a polygon as in the continuous case, but a finite set of points. In order to compute the overlapping polygon we look for the extreme points in Domain.
We give an example[5]. Assume $\mathit{OrgBnd} = \langle (0,0), (19,0), (19,12) \rangle$, $\mathit{Dom}(X) = [1..20]$ and $\mathit{Dom}(Y) = [0..15]$. The example is depicted on the left-hand side of Figure 5.14. Domain consists of all points on the integer grid that lie in OrgBnd to the right of the line $x = 1$. The extreme points of Domain are marked by circles: $(19,0), (19,12), (8,5), (5,3), (2,1), (1,0)$. (Observe that $(11,7)$ does lie in OrgBnd because it lies above the line $-12x + 19y = 0$.) The extreme points in Domain are the vertices of the so-called *integer hull* (see [Har99]) of $S = \mathit{OrgBnd} \cap \{(x,y) \in \mathbb{R}^2 \mid x \geq 1\}$, which is the convex hull of all the integer points in S.
Assume that OrgBnd has m vertices with rational coordinates and that the absolute value of the numerator and the denominator of each coordinate is bounded by A_{max}. Then the extreme vertices of Domain can be computed in time $O(m \log A_{max})$ (cf. [Har99]).

[5] This example stems from Figure 4.1 in [Har99].

Suppose now we have computed the overlapping polygon. Observe that its vertices may have rational coordinates even if the coordinates of all shape polygons are integers. The next critical step in the pruning algorithm is the plane sweep which examines $Pl_1 = Domain_1 \setminus Overlap(Shp_1, \mathcal{F}_2)$ and narrows the variable domains. Suppose the situation is as shown in Figure 5.14 and we want to prune the lower endpoint of the domain of X_1. Observe that not all points in Pl_1 are admissible, only those points which lie on the integer grid are allowed. In the example in the figure, the leftmost point in Pl_1 is $p = (2.75, 4.25)$, but every leftmost point with integer coordinates has 4 as its x-coordinate. Clearly, computing the leftmost (rightmost) point in Pl_1 and rounding up (down) its x-coordinate yields a narrowing algorithm for X_1. But – as the example shows – this algorithm does not achieve bound-consistency.

In [BGT01], the authors suggest a modification of the sweep algorithm which can increase the pruning power in some cases. The idea is to generate a check event for every integer x_i in $Dom(X_1)$, which makes the sweep pause at position x_i. Thus it is possible to test whether Pl_1 contains a point (x, y) with $x = x_i$. But one has to pay a price for the better pruning: The total running time increases by a linear term in the size of the variable domains. (An asymptotically better running time can be achieved by using results from integer linear programming in two dimensions [EL03].)

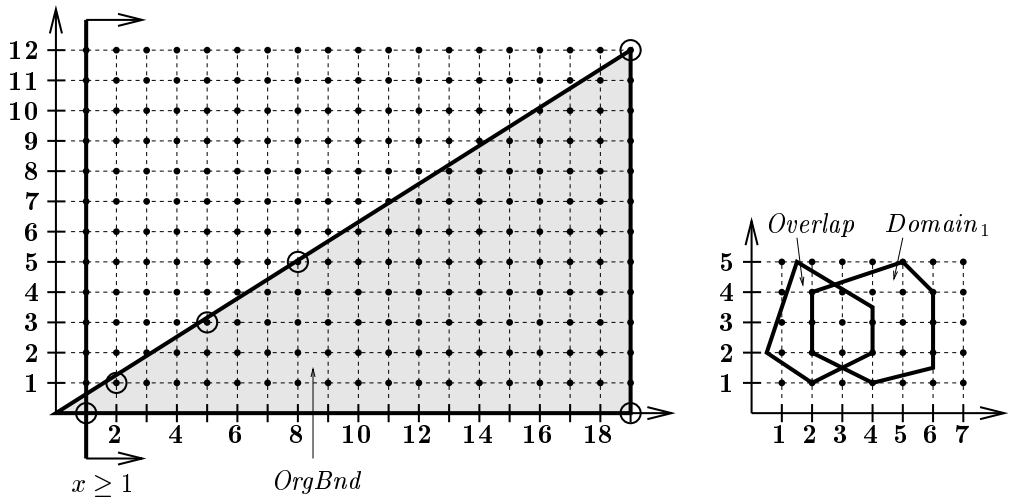


Figure 5.14: Examples depicting problems with integer domains.

More than two objects

Suppose we have a non-overlapping constraint on n objects, whose placement is in general not fixed. So with the ith object we associate a family $\mathcal{F}_i$ of convex polygons, which is described by a shape polygon Shp_i, origin variables X_i, Y_i, and an origin boundary polygon $OrgBnd_i$. Suppose we want to prune the origin variables of the first object. A straight-forward extension of the algorithm for two objects works as follows: We compute $O_{1,i} = Overlap(Shp_1, \mathcal{F}_i)$ for $i = 2, \ldots, n$. Then we examine $Pl_1 = Domain_1 \setminus (\bigcup_{i=2}^{k} \mathrm{int}(O_{1,i}))$ with a sweepline algorithm. Observe that the union of the overlapping polygons does not have to be computed explicitly, this can be done on-the-fly during the sweep. Of course, this makes the sweep more complicated as before, and one has to pay a logarithmic factor in running time.

Let us compare this global approach with a setting where we have $n-1$ independent binary non-overlapping constraints between the first object and the remaining objects. In general, we achieve a better running time because we perform only one sweep instead of $n-1$ sweeps. Sometimes we also achieve better pruning, as the following example shows: We want to place three objects, the shape of each object is a 3×3 square (with its lower left corner at $(0,0)$). The placement of the first object is not fixed, suppose $Dom(X_1) = Dom(Y_1) = [0,5]$ and $OrgBnd_1 = [0,5] \times [0,5]$. The other two objects are fixed with $(X_2, Y_2) = (0,0)$ and $(X_3, Y_3) = (0,3)$. The situation is depicted on the left-hand side of Figure 5.15.
Let us assume first that we have three binary non-overlapping constraints, one for each pair of objects. Since the constraint between the first and the second object is not aware of the third object, it will "think" that the first object could be placed on top of the second one. Thus it will not narrow the domain of X_1. A similar argument shows that the constraint between the first and the third object also leaves the domain of X_1 unchanged.
However, it is clear that if we want to place the first object such that there is no overlapping, then X_1 must be at least 3. Consider now the case that there is only one global non-overlapping constraint between all the three objects. Suppose that the pruning algorithm overlays the overlapping polygons $O_{1,2}$ and $O_{1,3}$ (as described above). Then it detects that the first object cannot be placed at a point with x-coordinate less than 3 (cf. right-hand side of Figure 5.15).

In the example above, the approach worked well. But if we change the example slightly, we obtain an instance with three objects, where our algorithm does not detect failure, although there is no solution. In our next example each object has identical parameters: For $i = 1, 2, 3$ Shp_i is the 3×3 square (with lower left corner at $(0,0)$), $Dom(X_i) = [1,5]$, $Dom(Y_i) = \{2\}$

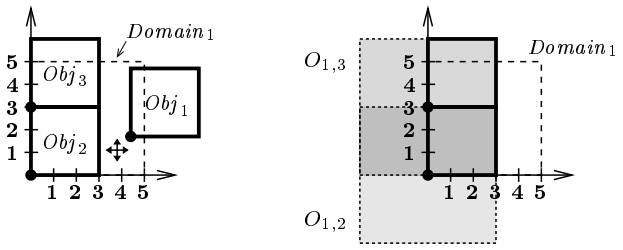


Figure 5.15: An example where bound-consistency is achieved for a ternary non-overlapping constraint.

and $OrgBnd_i = [0,5] \times [0,5]$. Looking at Figure 5.16 it is easy to see that a ternary non-overlapping constraint on the three objects has no solution. Since all three objects have identical parameters, all overlapping polygons are identical, too. The common overlapping polygon O is also depicted in the figure. As all x-coordinates of the points in O are between 2 and 4, we see that the algorithm does not narrow the domains of X-variables and it also does not detect failure.

In general we can make the following observation: If we have n identical objects, then the algorithm sketched above does not achieve more pruning than the algorithm for the binary case, because all overlapping polygons will be identical.

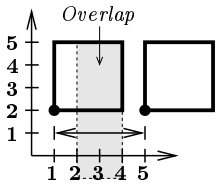


Figure 5.16: Two squares can be placed without overlapping, but there is no room for a third one.

The last example makes our algorithm look bad, but we will show below that there is no efficient algorithm which can decide solvability of non-overlapping constraints of arbitrary arity (unless P=NP). (For this we restrict the domain endpoints of the variables and the coordinates of the vertices to be rational numbers.) We show that deciding solvability is NP-complete in the strong sense (cf. [GJ79, Chapter 4.2]). This means that the problem is even hard, when we only consider instances with "small" numbers as denominators and numerators.

Theorem 5.2 *Deciding satisfiability for non-overlapping constraints (of arbitrary arity) is NP-complete in the strong sense.*

Proof. It is easy to see that the problem is in NP, because we can verify in polynomial time that a given variable assignment satisfies the constraint. To prove NP-completeness in the strong sense, we will give a (pseudo-polynomial) reduction from a scheduling problem called "Sequencing with release times and deadlines"[6] (see [GJ79, problem SS1, pg. 236]), which is known to be NP-complete in the strong sense:

Sequencing with release times and deadlines
An instance of the problem consists of a set T of tasks and, for each task $t \in T$, a length $l(t) \in \mathbb{Z}^+$, a release time $r(t) \in \mathbb{Z}_0^+$, and a deadline $d(t) \in \mathbb{Z}^+$. The question is, if there is a one-processor schedule for T that satisfies all release time constraints and meets all the deadlines, i.e. an injective function $\sigma : T \mapsto \mathbb{Z}_0^+$, with $\sigma(t) > \sigma(t')$ implying $\sigma(t) \geq \sigma(t') + l(t')$, such that, for all $t \in T$, $\sigma(t) \geq r(t)$ and $\sigma(t) + l(t) \leq d(t)$.

Let $T = \{t_1, \ldots, t_n\}$ be a set of tasks. For each task t_i we introduce a rectangular object. The shape of this object is the axis-parallel rectangle with width $l(t_i)$ and height 1, i.e. $Shp_i = [0, l(t_i)] \times [0, 1]$. The domains of the origin variables are defined as follows: $Dom(X_i) = [r(t_i), d(t_i) - l(t_i)]$ and $Dom(Y_i) = \{0\}$. The origin boundary is chosen such that it does not impose further restrictions: $OrgBnd_i = Dom(X_i) \times [0, 1]$. Clearly, the X-variables correspond to the starting times of the tasks. It easy to see that a schedule σ for the instance of the sequencing problem corresponds to a solution of the constraint, we simply assign to X_i the value $\sigma(t_i)$ for $i = 1, \ldots, n$. On the other hand a solution to the constraint allows us to construct a schedule, we set $\sigma(t_i) = \lfloor X_i \rfloor$. The other properties of a pseudo-polynomial reduction (as defined at [GJ79, Section 4.2.2]) are also fulfilled. □

Three-dimensional convex polytopes

We extend our result to two convex polytopes in the three-dimensional space $\mathbb{R}^3$. Thus each object that we place is described by a convex shape polytope, a convex origin boundary polytope and the origin of the shape polytope is determined by three variables X, Y and Z. It is easy to see that all arguments from above also hold in higher dimensions than two. And hence, the outline of the algorithm to narrow the bounds of the variables stays the same as

[6] This problem can be viewed as a one-dimensional non-overlapping constraint.

in Section 5.1. However, the computations involved in the different steps become more demanding as the dimension increases.

Now we examine the different steps of the algorithm. As before let $n_i = |Shp_i|$ and $m_i = |OrgBnd_i|$.

- Find combinatorial representations of $Shp_1, Shp_2, OrgBnd_1, OrgBnd_2$:
  Computing a combinatorial representation from a pure vertex representation of a polytope P can be done in time $O(|P| \log |P|)$ (see Section 2.3).

- Determine $Domain_1$ and $Domain_2$:
  We compute the intersection of $OrgBnd_i$ with the three-dimensional box $Dom(X_i) \times Dom(Y_i) \times Dom(Z_i)$. This takes time $O(m_i \log m_i)$ (see [HMMN84]). There is also a linear time algorithm ([Cha92]), but it is rather complicated. Observe that $|Domain_i| = O(m_i)$.

- Compute the overlapping polygon $Overlap(Shp_1, \mathcal{F}_2)$:
  As before we start by computing the Minkowski sum $S = Shp_2 \oplus -Shp_1$. We use the algorithm by Guibas and Seidel [GS87] with running time is $O(n_1 + n_2 + |S|)$. We will not give the details of their algorithm but we sketch some basic ideas. In the two-dimensional case we map each polygon onto the unit circle (cf. Figure 5.7) such that every vertex corresponds to an arc on the circle (which consists of all directions where the vertex is extreme) and every edge corresponds to a point on the circle (the outer normal vector of unit length). Computing the Minkowski sum basically amounts to overlaying the mappings of the two involved polygons. In the three-dimensional case each polytope is mapped to the unit sphere (Guibas and Seidel call this a direction map): every vertex corresponds to a surface patch (consisting of the directions where it is extreme), every edge is mapped to a great arc of the circle, and every facet corresponds to a point on the sphere (the outer normal vector of unit length). In order to determine the Minkowski sum, one can compute the overlay of the two maps.
  The problem is that S may be very complex: As Guibas and Seidel point out, $|S|$ can vary from $\Theta(n_1 + n_2)$ to $\Theta(n_1 \cdot n_2)$.

  After that we determine for each facet f of S a vertex of $Domain_2$ that is extreme in the direction of the inner normal of f (denoted by $\vec{n}_{\text{in}}(f)$). This means we want to maximize $\vec{n}_{\text{in}}(f)^T p$ subject to $p \in Domain_2$. Observing that $Domain_2$ can be written as the intersection of half-spaces where each half-space corresponds to a facet, we see that we have to solve a linear program for each facet f. However, the constraints of

the program are always the same, only the objective function varies. Therefore we can use an algorithm by Guibas et al. [GSC87] that finds the extreme vertices for all facets of S in time $O((m_2+|S|)\log m_2)$. This algorithm builds the direction map of $Domain_2$ (see again Figure 5.17) and constructs a data structure for locating points in this map. This solves the problem because a vertex of $Domain_2$ is extreme for all points (i.e. directions) in its corresponding surface patch.

Finally, we compute $Overlap(Shp_1, \mathcal{F}_2)$ as the intersection of $|S|$ half-spaces. (Each half-space corresponds to a facet translated by an extreme vertex of $Domain_2$, as in the two-dimensional case). This computation can be done in time $O(|S|\log|S|)$ (see [GO97, Chapter 19] or [PM79]). Thus the total time needed for the computation of the overlapping polygon is $O(n_1+n_2+(|S|+m_2)\log(|S|+m_2))$. Moreover, we observe that $|Overlap(Shp_1, \mathcal{F}_2)| = O(|S|)$.

- Examine $Pl_1 = Domain_1 \setminus \mathrm{int}(Overlap(Shp_1, \mathcal{F}_2))$:
  In the two-dimensional case we used an algorithm that moves a sweep-line across the plane in order to examine Pl_1. Hertel et al. [HMMN84] show that in the three-dimensional case boolean operations on convex polytopes can be computed by moving a sweepplane through the space. Their algorithm is quite elaborated and not just a straight-forward generalization of the two-dimensional sweep, so we omit the details here. They achieved a running time of $O(t\log t)$ where t is the total number of vertices of both polygons. So in our case $t = O(m_1 + |S|)$. In order to determine the lower and upper endpoints of the variable domains we simply scan all vertices of Pl_1.

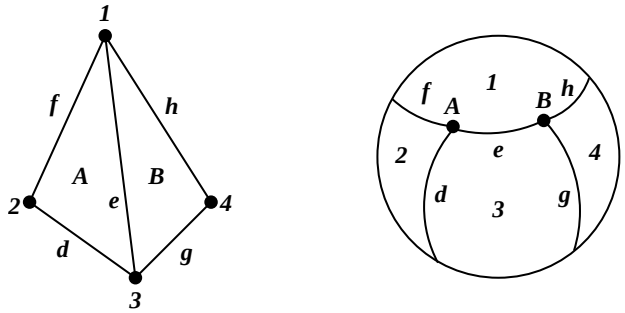


Figure 5.17: Mapping a 3D polytope onto the unit sphere (reproduced from [GSC87]).

To summarize, we obtain an algorithm for the three-dimensional case

which achieves bound-consistency. The overall running time is $O(s \log s)$ with $s = n_1 + n_2 + m_1 + m_2 + |S|$. And hence, the total running time heavily depends on the complexity of $S = \mathit{Shp}_2 \oplus -\mathit{Shp}_1$. This can be as small as $\Theta(n_1 + n_2)$ and then we pay only a logarithmic factor compared to the two-dimensional case. But it can also be $\Theta(n_1 \cdot n_2)$, which makes the algorithm impractical for large values of n_1 and n_2, because it computes S completely.

One might ask what happens if we consider d-dimensional objects with $d > 3$. The lemmas and theorems that we discussed in this chapter hold in higher dimensions. So one might try to apply basically the same algorithm as before. But we do not think that this is practical. One can show for example that a d-dimensional polytope with k vertices may have up to $\Theta(k^{\lfloor d/2 \rfloor})$ facets (see [GO97, Chapter 19]). Therefore, we do not pursue this issue any further.

5.6 Comparison with related work

In constraint programming non-overlapping constraints have been studied for a long time. Lahrichi and Gondran [LG84] introduced the notion of *compulsory part* for a family (of rectangles), which is the intersection of all members of that family. The overlapping polygon that has been defined in our work can be seen as a generalization. However, it is easy to see that $\mathit{Overlap}(\mathit{Shp}_1, \mathcal{F}_2)$ may be non-empty although the compulsory part of $\mathcal{F}_2$ is empty. And hence, an approach that is based on compulsory parts does not yield a bound-consistency algorithm.
Beldiceanu and Contejean [BC94] provided a global constraint called *diffn*. It allows to state a non-overlapping constraint between several d-dimensional axis-parallel boxes.

To the best of our knowledge, all shapes that have been considered so far are axis-parallel rectangles (and axis-parallel boxes in higher dimensions). More complex shapes are approximated by sets of rectangles (cf. [CF94]), the origin of every rectangle is linked explicitly to the origin of its shape, and there is a non-overlapping constraint for any pair of rectangles that belong to different shapes. This approach suffers from poor propagation. Our approach allows for the first time to model complex convex shapes directly. But non-convex shapes still have to be decomposed into convex shapes.

Chapter 6

Dominance graphs

The second part of this thesis deals with a problem from the field of computational linguistics. Roughly speaking, the problem is to assemble some given tree fragments into a tree T such that some given constraints are satisfied. These constraints have the form "node u should *dominate* node v", where *dominate* means that u is a (not necessarily proper) ancestor of v. The problem is given to us as a so-called *dominance graph* D, which will be formally defined later. Such a graph represents both the tree fragments and the dominance requirements for T.

Dominance based tree descriptions have been investigated in different areas for a long time. They were used in automata theory in the sixties [TW67], rediscovered in computational linguistics in the early eighties [MHF83], and investigated from a logical point of view in the early nineties [BRVS95]. Since then, there have been several applications in computational linguistics: They have been used for grammar formalisms [VS92, RVSW95, DT99, Per00], in natural language semantics [Mus95, ENRX98], and for discourse analysis [GW98].

The details of the tree descriptions vary over the different applications, but there is a framework called *dominance constraints* [KNT01], which is general enough to be applied to a variety of problems. However, Koller et al. [KNT01] showed that deciding solvability of dominance constraints is an NP-complete problem. From a practical point of view, there were doubts whether these constraints are a useful tool. In fact, the solvers that existed at that time were not efficient enough.
The doubts were removed by the work of Althaus et al. [ADK+01, ADK+03]. They identify the subclass of *normal* dominance constraints. This subclass is sufficiently large for many practical applications and solvability can be decided in polynomial time. The algorithms developed by Althaus et al. actually work on dominance graphs. They describe a back-and-forth trans-

lation between dominance graphs and normal dominance constraints, which makes their graph algorithms applicable to the logical language of dominance constraints.

The work in the following chapters is based on the papers [ADK+01, ADK+03], which are joint work with Ernst Althaus, Denys Duchier, Alexander Koller, Kurt Mehlhorn and Joachim Niehren. But we also present some results which improve upon this work and have not been published yet. In particular, the solvability test is by a factor of n faster and the enumeration algorithm outperforms our old algorithm by a factor of n^2, where n is the number of nodes of the graph.

The second part of the thesis is organized as follows: In the remainder of this chapter we discuss an example from computational linguistics to motivate the subsequent work and we introduce some basic definitions. In Chapter 7 we develop a linear time algorithm which can check if a given dominance graph D has a solved form. We show in Chapter 8 how to enumerate all N minimal solved forms of D in time $O(m + N \cdot nm)$, where n is the number of nodes and m is the number of edges of D. Finally, we discuss related work in Chapter 9. In particular, we describe the relationship between dominance graphs and normal dominance constraints.

6.1 Motivation

As an example for an application of dominance graphs in computational linguistics we will give a brief introduction to *scope underspecification* [EKN01, AC92, Rey93, Bos96]. This application examines ambiguous sentences with respect to the scope of quantifiers. Here is an example:

> Every scientist speaks a language.

This sentence has two possible readings which can be determined by the following continuations:

1. ... But not all of them speak the same one.

2. ... This world language of science is English.

In the first reading, no two scientists (necessarily) speak the same language. But in the second one, there is one certain language that is common to all scientists. The difference between the two readings and also the term scope ambiguity become clear when one looks at the representations of the two readings as logic formulas.

1. $\forall x$:(scientist$(x) \rightarrow \exists y$:(lang$(y) \wedge$ speak(x, y)))
2. $\exists y$:(lang$(y) \wedge \forall x$:(scientist$(x) \rightarrow$ speak(x, y)))

In the first formula, the existential quantifier is in the scope of the universal quantifier, whereas in the second formula it is vice versa. The two readings of the sentence also correspond to two different parse trees T_1 and T_2 shown in Figure 6.1. In T_1 the node $n_\forall$ labelled with the universal quantifier is an ancestor of the node $n_\exists$ labelled with the existential quantifier, i.e. $n_\forall$ dominates $n_\exists$. In T_2 the dominance relation between the two quantifiers is opposite.

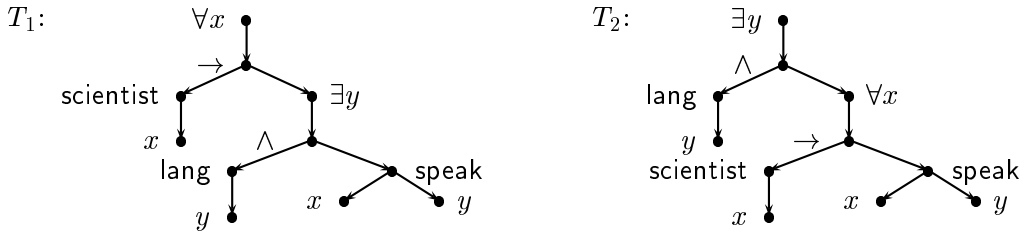


Figure 6.1: Trees corresponding to the readings of "Every scientist speaks a language".

Let us now focus on the similarities of the two readings and how they are reflected in the two formulas and trees respectively. We begin with the logic formulas. Both of them are composed of the following three parts which correspond to the representations of the "semantic material" like "every scientist", "a language" and "speak":

- $\forall x$:(scientist$(x) \rightarrow \ldots$)
- $\exists y$:(lang$(y) \wedge \ldots$)
- speak(x, y)

The ellipses "..." in a part are place-holders where another part has to be plugged in. Furthermore, we know that in any reading the part for "speak" is within the scope of both quantifiers.

We can decompose the parse trees in an analogous way and obtain the tree fragments that are shown in Figure 6.2 (ignoring the dashed edges for the moment). The fragment of each quantifier contains an unlabelled leaf node, which is a place-holder where the root of another fragment can be plugged in. Such a leaf is called a *hole*. The graph in the figure contains two dashed edges which are directed from the holes to the root of the "speak"-fragment.

They indicate that the respective hole dominates the "speak"-fragment in any possible parse tree for the sentence. We say that the dashed edges are *dominance edges* and the solid edges are *tree edges*.

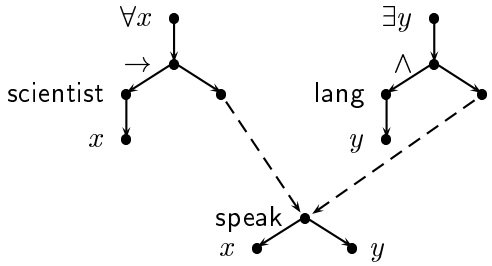


Figure 6.2: A graph representation of "Every scientist speaks a language".

Graphs like the one in Figure 6.2 are useful to model the semantics of an ambiguous sentence. Instead of committing to a particular reading (which is the case if we use a parse tree), we can maintain all possible readings in one model. For such a model two questions arise naturally:

1. Can we assemble the fragments to a parse tree that fulfils all the dominance requirements?

2. Can we enumerate all the trees that are a solution for the model?

Ambiguity is an important problem in language processing, because the number of readings of a sentence grows quickly with the number of quantifiers and scope ambiguity may interact with other sources of ambiguity. In the remainder of this section we give some examples which illustrate this.

The following example has already 56 readings, its corresponding graph is shown in Figure 6.3.

> John says that some representative of every department in a company saw a sample of each product.

There are even more striking examples. The following sentence is due to Hobbs [Hob83] and has about 200 readings:

> Many people feel that most sentences exhibit too few quantifier scope ambiguities for much effort to be devoted to this problem, but a casual inspection of several sentences from any text should convince almost everyone otherwise.

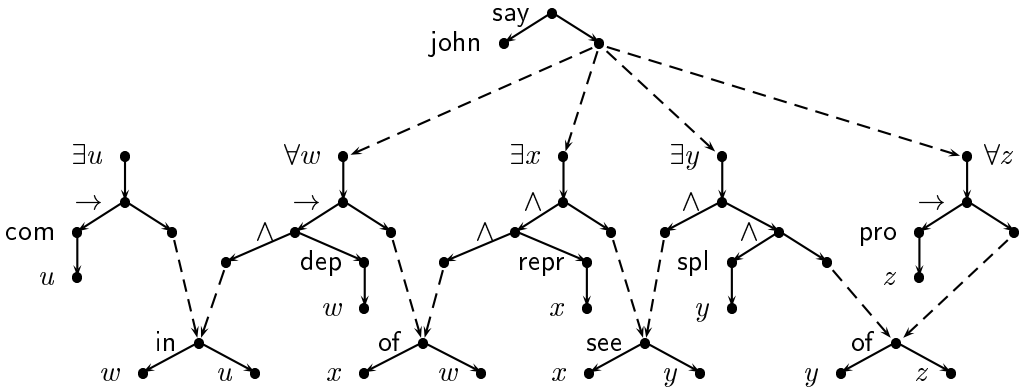


Figure 6.3: A graph representation of a sentence with 56 readings.

6.2 Definitions

In this section we will give the definition of a *dominance graph* and define some related notions. Informally, a dominance graph is a collection of rooted tree fragments and some dominance requirements between them. A dominance requirement is given as a directed edge from a leaf of one fragment to the root of another fragment. So if one deletes all the node labels in Figure 6.3, one obtains a dominance graph. In the formal definition below we allow w.l.o.g. only tree fragments of height one, because this simplifies the succeeding arguments. (On page 194, we discuss how to eliminate this restriction.)

Definition 6.1 (dominance graph) *A* dominance graph D *is a directed graph* (V, E) *with two partitions* $V = V_r \dot\cup V_l$ *and* $E = E_t \dot\cup E_d$. V *is partitioned into* root nodes V_r *and* leaf nodes V_l; E *is partitioned into* tree edges E_t *and* dominance edges E_d. *The following must be satisfied:* $E_t \subseteq V_r \times V_l$ *and* $E_d \subseteq V_l \times V_r$, *i.e. tree edges are directed from roots to leaves, and dominance edges point from leaves to roots. Moreover,* $(V_r \dot\cup V_l, E_t)$ *is a forest where each tree has height one.*
We write $D = (V_r \dot\cup V_l, E_t \dot\cup E_d)$ *to denote the dominance graph.*

In Figure 6.4 we depict three example dominance graphs. We draw roots as squares, leaves as circles, tree edges as solid darts and dominance edges as dashed darts; the different tree fragments are indicated by dotted silhouettes.

Let us compare the graphs D_1 and D_2 in Figure 6.4: Both graphs have the same tree fragments, only their dominance edges are different. We observe that all dominance requirements of D_1 are also encoded in D_2. E.g., the fact that the node b should be an ancestor of f is expressed explicitly in D_1 by

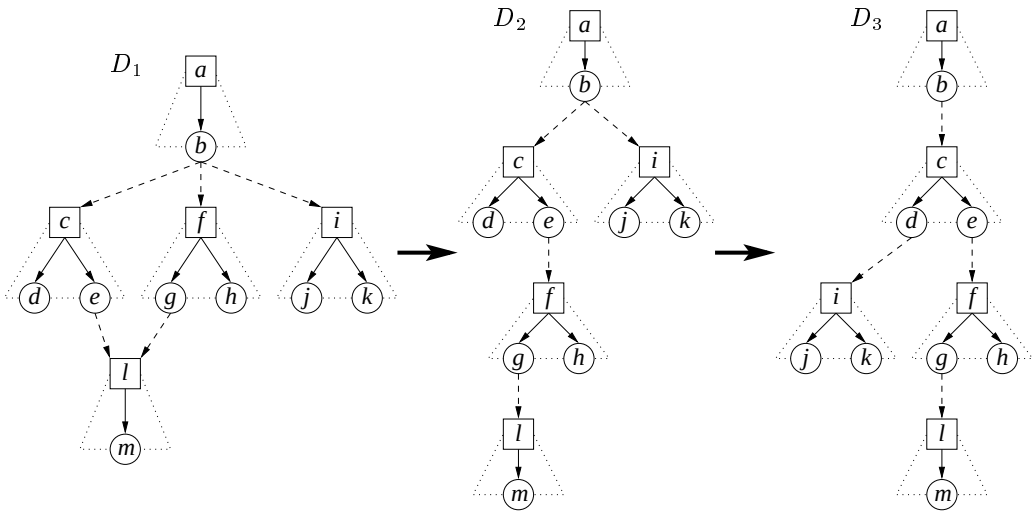


Figure 6.4: Examples to illustrate the definitions of a dominance graph and related notions.

a dominance edge, in D_2 this is encoded implicitly because there is a path from b to f. We see that D_2 is more restricted than D_1, for D_2 fixes f to be a descendant of c, whereas D_1 leaves the relation between these two nodes open. We say that D_2 is a (strict) *amplification* of D_1. In order to describe this notion formally, we define the *reachability relation* $\mathrm{Reach}(D)$ of a directed graph $D = (V, E)$: $\mathrm{Reach}(D)$ is the transitive closure of E (interpreted as binary relation over V), i.e. a tuple (u, v) of nodes is contained in $\mathrm{Reach}(D)$ iff there is a non-empty path from u to v in D.

Definition 6.2 (amplification) *Let* $D = (V_r \dot\cup V_l, E_t \dot\cup E_d)$ *and* $D' = (V_r' \dot\cup V_l', E_t' \dot\cup E_d')$ *be two dominance graphs. We say that* D *is an* amplification of D' *if the following holds:* $V_r = V_r'$, $V_l = V_l'$, $E_t = E_t'$ *and* $\mathrm{Reach}(D) \supseteq \mathrm{Reach}(D')$. *If we have* $\mathrm{Reach}(D) \supset \mathrm{Reach}(D')$, *then* D *is a* strict *amplification of* D'.

Let us look at the examples in Figure 6.4 again. We see that D_3 is a strict amplification of D_2, and hence also of D_1. Recall that our goal is to assemble the tree fragments to a tree such that its ancestor-descendant relation fulfils all dominance requirements imposed by the respective dominance graph. Any tree which is a solution for D_3 is also a solution for D_1 and D_2, because D_3 is an amplification. We observe that D_3 has a nice property: Every node is incident to at most one dominance edge, i.e. the dominance edges match some leaves and roots. Thus the dominance edges tell us how to assemble the tree

fragments: We plug c into b, i into d, f into e and l into g. (One can imagine the plugging process as contracting each dominance edge and identifying its two incident nodes.) We call D_3 a *configuration*, which is defined formally below:

Definition 6.3 (configuration) *A dominance graph $D = (V_r \dot\cup V_l, E_t \dot\cup E_d)$ is called a* configuration *if D is a forest and E_d is a matching, i.e. each node is incident to at most one dominance edge. A configuration D which is an amplification of a dominance graph D' is called a* configuration of D'.

We can now formalize the problems from Section 6.1:

1. Decide whether a given dominance graph D has a configuration.
2. Enumerate all configurations of a dominance graph D.

Let us reconsider the dominance graph D_2 in Figure 6.4. D_2 is a tree, but it is not a configuration, because the leaf b has two outgoing dominance edges. As D_3 is an amplification of D_2 which in turn is an amplification of D_1, we can see D_2 as an "intermediate stage" between the original problem encoded in D_1 and the solution represented by D_3. This stage is called *solved form* and defined below.

Definition 6.4 (solved form) *A dominance graph $D = (V_r \dot\cup V_l, E_t \dot\cup E_d)$ is in* solved form *if D is a forest. A solved form D which is an amplification of a dominance graph D' is called a* solved form of D'. *We say that D is a* minimal solved form of D' *if there is no solved form D'' of D' with* $\mathrm{Reach}(D'') \subset \mathrm{Reach}(D)$.

By definition, every configuration is also a solved form, but in general not a minimal solved form. In our example, D_2 and D_3 are solved forms of D_1. Since D_3 is a strict amplification of D_2, it is not a minimal solved form of D_1. D_2, however, is a minimal solved form of D_1. This can be seen as follows: Consider a solved form D with $\mathrm{Reach}(D_1) \subset \mathrm{Reach}(D) \subseteq \mathrm{Reach}(D_2)$. Since $\mathrm{Reach}(D_2) = \mathrm{Reach}(D_1) \cup \{(e, f), (e, h), (e, g)\}$, $\mathrm{Reach}(D)$ contains at least one of the three tuples $(e, \cdot)$. As f, h, and g belong to the same tree fragment, $\mathrm{Reach}(D)$ must contain all of them.

In the sequel we will show that the term "solved form" is justified, i.e. every dominance graph in solved form has a configuration. In the proof in the lemma below we describe how a configuration can be constructed from a solved form.

Lemma 6.1 *A dominance graph D in solved form has a configuration.*

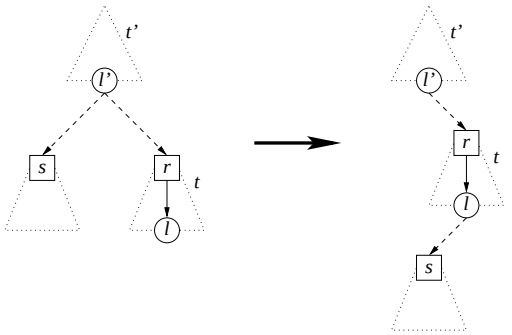


Figure 6.5: Application of Transformation Rule 1: The dominance edge from l' into s is shifted down to l.

Proof. By the definition of solved form, D is a forest. If it is not a configuration, then there exists a leaf with more than one outgoing dominance edge. So we can apply the following transformation to D and obtain a new graph D':

Transformation Rule 1 *Let l' be a leaf with at least two outgoing dominance edges (l', r) and (l', s). Choose an arbitrary leaf l in the fragment with root r and replace (l', s) by (l, s), see Figure 6.5.*

Clearly, D' is in solved form. Moreover, it is an amplification of D because the edge (l', s) in D is replaced by a path from l' to s in D'. As D is a forest, we have $(l, s) \notin \mathrm{Reach}(D)$ (cf. left-hand side of Figure 6.5). Thus D' is a strict amplification. This means the following: By applying the rule above repeatedly, we can generate a series of strict amplifications in solved form until we obtain a configuration of D. Each application increases the size of the reachability relation, and hence there can be at most n^2 applications, where n is the number of nodes of D. □

The lemma above implies that configurability and solvability (i.e. deciding whether a dominance graph has a configuration or a solved form respectively) are equivalent problems. Concerning our second problem, the enumeration problem, it also suffices to focus on solved forms: We will show that all configurations of a dominance graph D can be obtained by applying Transformation Rule 1 exhaustively to its minimal solved forms. Let us assume that D is connected[1] so that all its solved forms are trees. Then our claim follows immediately from the lemma below:

[1] Observe that this property can always be achieved: Add to D a dummy fragment with a root r and a single leaf l, then add dominance edges from l to every root different from r.

Lemma 6.2 *Let D_1 and D_2 be two dominance graphs, which are trees. If D_2 is an amplification of D_1, then D_2 can be obtained from D_1 by at most n^2 applications of Transformation Rule 1, where n is the number of nodes in D_1.*

Proof. If $D_1 \neq D_2$, then there is a dominance edge (l, r) which is in D_2 but not in D_1. Let α denote the lowest common ancestor of l and r in D_1. We have $\alpha \neq r$, for otherwise r would be an ancestor of l in both D_1 and D_2; this would imply that D_2 contains a cycle. Moreover, $\alpha \neq l$. Assume otherwise, then there is a path from l to r in D_1. Since (l, r) is not an edge of D_1, this path must visit a node x different from l and r. As D_2 is an amplification of D_1, there is a path from l to r in D_2 that visits x. But since D_2 is a tree containing the edge (l, r), this is impossible.

From $l \neq \alpha \neq r$ we conclude that in D_1 the nodes l and r are descendants of different children s_l and s_r of α (see left-hand side of Figure 6.6). In D_2, l and hence s_l are ancestors of r. The fact that s_l and s_r are ancestors of r in D_2 but siblings in D_1 implies that α is a leaf and s_l and s_r are roots. In D_2, s_l must be an ancestor of s_r or vice versa (cf. right-hand side of Figure 6.6). Let us assume the former (the other case is symmetric), and let (s_l, h) be the first edge on the path from s_l to s_r. Clearly, h is a leaf in the fragment of s_l. Now we apply Transformation Rule 1 and replace in D_1 the dominance edge (α, s_r) by the dominance edge (h, s_r). Thus we obtain a new tree D with $\text{Reach}(D_1) \subset \text{Reach}(D) \subseteq \text{Reach}(D_2)$. And hence, we can transform D_1 into D_2 by at most n^2 applications of the transformation rule. (Observe that two trees with the same reachability relation are equal.) □

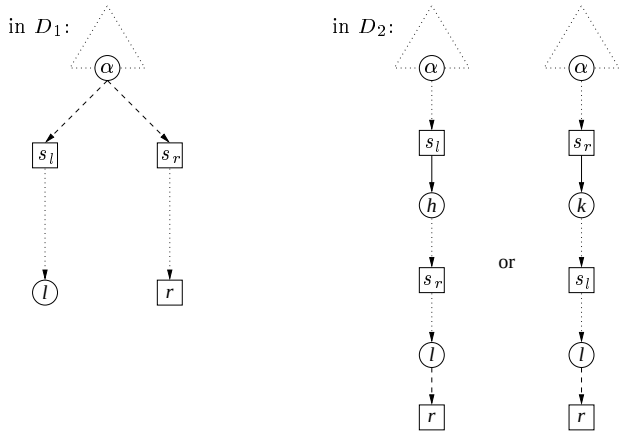


Figure 6.6: Situation in the proof of Lemma 6.2. (Dotted darts represent paths in the trees.)

Chapter 7

Deciding solvability

In this chapter we show that solvability of a given dominance graph D can be decided in linear time. We obtain this result as follows: First we prove that solvability of D is equivalent to the absence of certain cycles (called *harmful cycles*) in an undirected version of D. Then we develop a linear-time algorithm which can check whether a graph contains a harmful cycle or not.

Although the enumeration of solved forms is the topic of the next chapter, we start this chapter with the presentation of an enumeration algorithm. This brute-force algorithm is inefficient, but it assist us in proving the correctness of the harmful cycle criterion.

7.1 A brute-force enumeration algorithm for minimal solved forms

We discuss an algorithm for enumerating all minimal solved forms of a dominance graph D, which is due to Althaus et al. [ADK+03]. Let us denote the set of all minimal solved forms of D by $\mathcal{S}(D)$. How can we find a solved form of D? By definition, a solved form is a forest, i.e. it is acyclic and every node has at most one incoming edge. If D contains a cycle, then it has no solved form, and we can stop our search. So assume that D is acyclic. If it is not in solved form, then there must be a node s in D with two or more incoming edges. Recalling the definition of a dominance graph, we can infer that s must be a root node and the incoming edges are dominance edges. In the sequel we describe two transformation rules that allow us to reduce the indegree of a root while – in some sense – preserving the set of solved forms.

Looking at the dominance graph D in Figure 7.1, we see that the dominance edge d from l to s is superfluous, because there is an alternative path

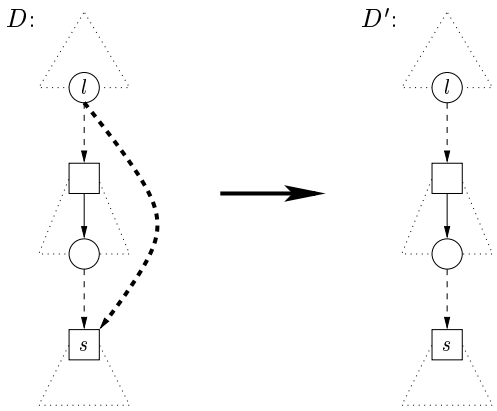


Figure 7.1: Eliminating a redundant edge in D to obtain D'.

from l to s. Thus the dominance requirement d is implied by transitivity. If we delete d from D and obtain D', we have $\mathrm{Reach}(D) = \mathrm{Reach}(D')$, and hence $\mathcal{S}(D) = \mathcal{S}(D')$. We call a dominance edge $d = (l, s)$ *redundant* in a dominance graph D if there is a path from l to s in $D \setminus d$. This gives rise to the following transformation rule:

Transformation Rule 2 (Redundancy Elimination) *Remove every redundant dominance edge.*

Our definition of redundancy coincides with the definition of *transitive redundancy* introduced by Aho et al. [AGU72].[1] We can apply their algorithm to make our dominance *reduced*, i.e. to remove all redundant dominance edges.

But even a reduced dominance graph D may contain a root s with two incoming dominance edges (l_1, s) and (l_2, s) (see the left-hand side of Figure 7.2). Consider a solved form D_s of D. Since both l_1 and l_2 dominate s in D_s and D_s is a tree, we conclude that l_1 is an ancestor of l_2 or vice versa. Assume the former, then l_1 must also be an ancestor of the root r_2 of the fragment containing l_2. Thus D_s is also a solved form of the graph $D_1 = D \cup (l_1, r_2)$ (cf. right-hand side of Figure 7.2). But if l_2 is an ancestor of l_1 in D_s, then D_s is a solved form of $D_2 = D \cup (l_2, r_1)$, where r_1 is the root of the fragment of l_1. Clearly, D_s cannot be a solved form of both D_1 and D_2. (Otherwise, $\mathrm{Reach}(D_s)$ would contain both (l_1, l_2) and (l_2, l_1), which cannot be in a tree.) This means that $\mathcal{S}(D) = \mathcal{S}(D_1) \dot{\cup} \mathcal{S}(D_2)$, which justifies our next transformation rule:

[1] Observe that a tree edge cannot be transitively redundant, because it is the only incoming edge of its incident leaf.

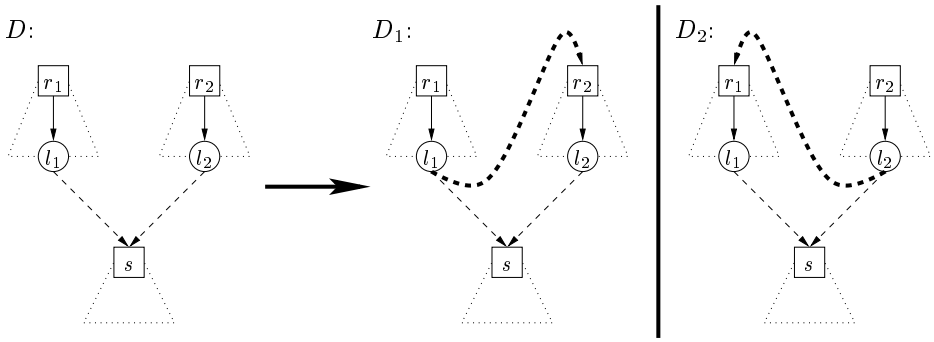


Figure 7.2: D_1 and D_2 are generated by applying the choice rule to D.

Transformation Rule 3 (Choice) *Let s be a root with at least two incoming dominance edges (l_1, s) and (l_2, s) and let r_1 and r_2 be the roots of the fragments containing the leaves l_1 and l_2, respectively. Generate two amplifications D_1 and D_2 by adding either (l_1, r_2) or (l_2, r_1) to D (see Figure 7.2).*

After an application of the choice rule, we have to work on two dominance graphs, but what do we gain? Looking at D_1, we see that the dominance edge (l_1, s) is redundant. We can remove it and reduce the indegree of s. Moreover, since by our assumption (l_1, s) is not redundant in D, we know that D_1 is a strict amplification of D. An analogous observation can be made for D_2. So whenever we apply the choice rule to a reduced dominance graph, the two generated graphs have a bigger reachability relation than the original one. This property will guarantee the termination of our enumeration algorithm.

The algorithm to enumerate the minimal solved forms of a dominance graph D is straightforward (see Algorithm 7.1): First we check whether the graph is acyclic, if not we can terminate. After that we eliminate all redundant dominance edges. Then we look for a root s with two incoming dominance edges. If no such root exists, then D is a forest and we report D as a minimal solved form. So assume we find such a root s. Then we apply the choice rule to s and process the generated instances D_1 and D_2 recursively.

We prove correctness and termination of the algorithm. First we show that any minimal solved form D_s of D is reported by the algorithm. We have seen that an application of the transformation rules above does not "lose" any solved forms. So it is easy to see by induction that D_s is an amplification of some solved form D' which is reported by the algorithm. Thus $\text{Reach}(D') \subseteq \text{Reach}(D_s)$. Since D_s is minimal, we get $\text{Reach}(D_s) = \text{Reach}(D')$, which implies $D_s = D'$, because both graphs are trees.

Algorithm 7.1 Enumerating the minimal solved forms of D (brute force)

Procedure: Enum-BruteForce(D)
1: **if** D contains no (directed) cycle **then**
2: eliminate all redundant dominance edges
3: **if** D has a root s with at least two incoming dominance edges **then**
4: apply the choice rule and generate two new instances D_1 and D_2
5: Enum-BruteForce(D_1); Enum-BruteForce(D_2)
6: **else** // D is in solved form
7: report D
8: **end if**
9: **end if**

It is clear that all reported dominance graphs are solved forms of D, but we have to convince ourselves that they are indeed minimal. Suppose that the algorithm reports a solved form D' of D which is not minimal. Then there exists a minimal solved form D_s of D such that D' is a strict amplification of D_s. This implies that D_s is also reported by the algorithm. Thus the algorithm has made an application of the choice rule which "separated" D' and D_s, i.e. the application generated two graphs D_1 and D_2 such that $D' \in \mathcal{S}(D_1)$ and $D_s \in \mathcal{S}(D_2)$. As we have shown, $\mathcal{S}(D_1)$ and $\mathcal{S}(D_2)$ are disjoint. And hence, D' cannot be an amplification of D_s, a contradiction.

Now we prove termination. Since the choice rule is always applied to reduced graphs, the reachability relation strictly increases every time. And hence, the recursion depth is bounded by the maximum size of this relation, which is n^2, where n is the number of nodes of D. Observe that the running time of this algorithm is exponential in general, because every call to the procedure `Enum-BruteForce` may spawn two recursive calls. If we apply the algorithm to an unsolvable graph, it may have to generate many intermediate dominance graphs until it finds out that all branches of the computation finally produce graphs with a cycle.

7.2 Harmful cycles

We will now develop a criterion which allows us to decide efficiently whether a given dominance graph has a solved form or not. The following presentation is based on [ADK+03]. So far we have only seen examples which have a solved form. It is time to study some unsolvable examples. Let us look at Figure 7.3 and convince ourselves that the three graphs depicted there are unsolvable. For D_1, this is easy, because it contains a directed cycle. Concerning D_2,

we see that the two leaves b and c in the topmost fragment both want to dominate the root j, because there is a path from b to j and one from c to j. But since b and c are siblings in a tree fragment, this is impossible. It is not obvious that D_3 has no solved form. Let us apply the choice rule to the root g: This generates two graphs $D_3' = D_3 \cup (b, d)$ and $D_3'' = D_3 \cup (e, a)$ (cf. Figure 7.4). Due to symmetry, it suffices to look at D_3'. We see that the two siblings b and c want to dominate i, which implies unsolvability.

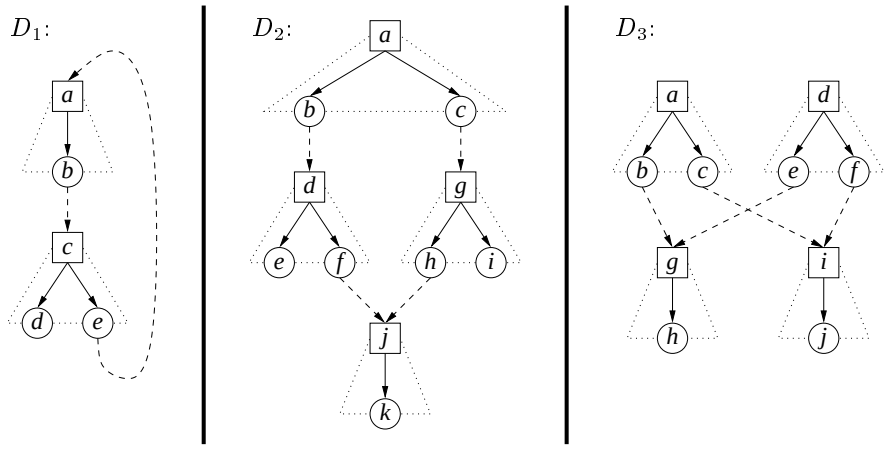


Figure 7.3: Three unsolvable dominance graphs.

Is there a property that all the examples in Figure 7.3 have in common? If we ignore the arrowheads of the edges, we observe that the three unsolvable dominance graphs contain a cycle. Unfortunately, a solvable dominance graph D^+ may also contain an "undirected" cycle. In Figure 7.5 we have a synopsis of an unsolvable graph D^- and a solvable graph D^+. (D^- is the graph in the middle of Figure 7.3, D^+ is a subgraph of the solvable graph on the left-hand side of Figure 6.4 on page 146.) D^- and D^+ look very similar, and they both contain undirected cycles C_1 and C_2 respectively.

What is the crucial difference between the two cycles? If we translate C_1 back to the directed graph D^-, we get two directed paths from a to i. Since both paths start with tree edges, they prove that two different children of a (namely b and c) both want to dominate the node i, which is impossible. So C_1 is a proof for the fact that D^- is not solvable.
Translating the cycle C_2 back to D^+ we obtain two directed paths from b to l. But this time both paths start with dominance edges. Thus there is no problem, because in a solved form c may be an ancestor of f or vice versa, thus both of them can dominate l.

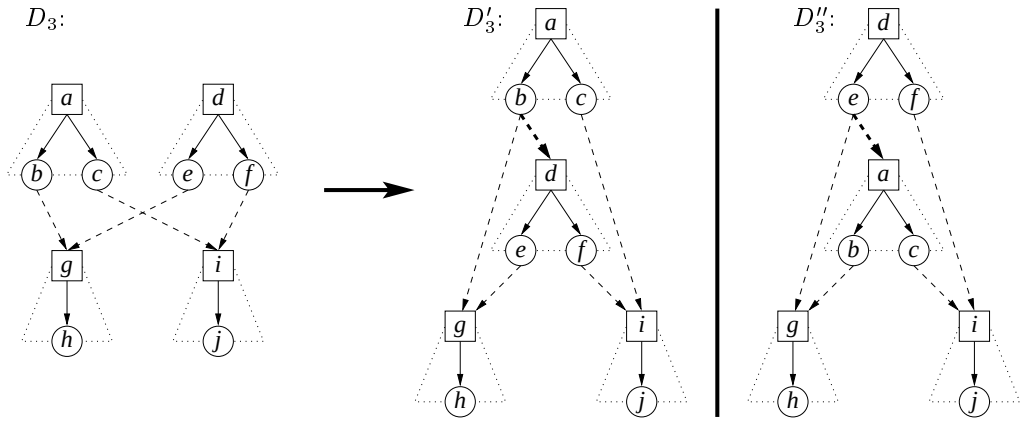
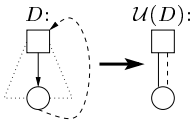


Figure 7.4: Applying the choice rule to D_3 to prove unsolvability.

We call a cycle which can prove unsolvability a *harmful cycle*[2]. The examples we have seen indicate that such a cycle must not visit two dominance edges that are incident to the same leaf. The remainder of this section is organized as follows: First we formally define harmful cycles, and then we prove that solvability is equivalent to the absence of harmful cycles.

As we have seen above, the directions of the edges do not matter when we look for harmful cycles. So instead of the original dominance graph D we search the underlying undirected dominance graph $\mathcal{U}(D)$.
Informally, this graph is obtained by deleting the arrowheads of all edges of D, the partition of the nodes into roots and leaves and the partition into tree and dominance edges are kept. As we can see on the right-hand side, $\mathcal{U}(D)$ can be a multigraph (see Definition 2.7).



This is reflected by our definition of an undirected dominance graph:

Definition 7.1 (undir. dom. graph) *An* undirected dominance graph U *is an undirected multigraph* (V, E, inc) *with two partitions* $V = V_r \dot\cup V_l$ *and* $E = E_t \dot\cup E_d$. *V is partitioned into* root nodes V_r *and* leaf nodes V_l*; E is partitioned into* tree edges E_t *and* dominance edges E_d. *The following must hold: Every edge $e \in E$ is incident to one root and to one leaf, and every leaf is incident to exactly one tree edge.*
We write $U = (V_r \dot\cup V_l, E_t \dot\cup E_d, inc)$ to denote the undirected dominance graph.

The underlying undirected dominance graph *of a (directed) dominance graph*

[2]In the papers [ADK+01, ADK+03] these cycles are called *hypernormal*.

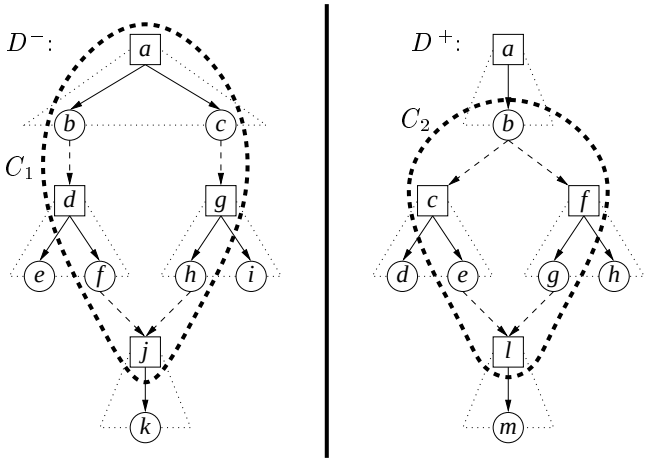
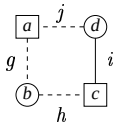


Figure 7.5: Two dominance graphs with undirected cycles (indicated by the dashed lines). D^- is not solvable, but D^+ is.

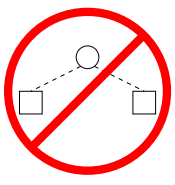
$D = (V_r \dot\cup V_l, E_t \dot\cup E_d)$ is the graph $\mathcal{U}(D) = (V_r \dot\cup V_l, E_t \dot\cup E_d, inc)$, where inc is defined as follows: For an edge $e = (u, v)$ of D we set $inc(e) := \{u, v\}$.

In order to simplify notation we write $e = \{u, v\}$ as an abbreviation for $inc(e) = \{u, v\}$, although there can be several edges connecting u and v. Observe that there can be at most one tree edge between u and v in an undirected dominance graph, but there may be more than one dominance edge. We allow this on purpose, because our algorithm may add dominance edges that are parallel to existing ones. (It will turn out that additional dominance edges do not harm, because they will basically be ignored.)

We need to introduce some notions to talk about undirected cycles. The cycle on the right side visits four nodes, we say that it makes four bends. A *bend* is a triple $\langle e, v, f\rangle$ which consists of two distinct edges e and f that are incident to the node v. Our example has the following bends: $\langle j, a, g\rangle$, $\langle g, b, h\rangle$, $\langle h, c, i\rangle$ and $\langle i, d, j\rangle$. Recalling the example in Figure 7.5, we see that the bend $\langle g, b, h\rangle$ is exactly the kind of bend that we want to avoid, because b is a leaf and g and h are both dominance edges. This motivates the following definition:



Definition 7.2 (harmful cycle) *Let C denote a cycle in an undirected dominance graph U. A bend $\langle e, v, f\rangle$ on C is called* forbidden *if v is a leaf and e and f are dominance edges; otherwise we say that $\langle e, v, f\rangle$ is* admissible. *The cycle C is called* harmful *if C is simple and all its bends are admissible.*



In the sequel we will prove that solvability of a dominance graph D is equivalent to the absence of harmful cycles in $\mathcal{U}(D)$. For this aim, we show two lemmas, which – in some sense – express the fact that an application of a transformation rule does not affect the existence of harmful cycles. (For the choice rule, this means that *both* generated instances contain a harmful cycle if and only if there is a harmful cycle in the original graph.) First, we prove that the removal of redundant edges does not affect the existence of a harmful cycle.

Lemma 7.1 *Let D' be a dominance graph which is obtained by applying Transformation Rule 2 to a dominance graph D. Then $\mathcal{U}(D)$ contains a harmful cycle iff $\mathcal{U}(D')$ does.*

Proof. Suppose that redundancy elimination has removed the dominance edge $e = (l, s)$ from D. Since the edges of D' are a subset of those of D, we only have to show the following: If $\mathcal{U}(D)$ contains a harmful cycle C which uses the edge $\{l, s\}$, then there is also a harmful cycle in $\mathcal{U}(D')$.

As C is harmful, the bend at l must be admissible, i.e. C must use the tree edge $\{r, l\}$ incident to l. So we may suppose that C starts with $\{r, l\}$ and then uses $\{l, s\}$ as second edge. Since e is redundant in D, there is a simple (directed) path P from l to s in D'. Let x denote the last node on C which is also visited by P ($x \neq l$, and possibly $x = s$). Let P_x denote the prefix of P from l to x. If $x = r$, then $P_x \circ (r, l)$ is a directed cycle in D', which translates to a harmful cycle in $\mathcal{U}(D')$. So assume $x \neq r$. Thus the suffix C_x of C from x to r is not empty. Identifying P_x with the corresponding undirected path in $\mathcal{U}(D')$, we obtain the simple cycle $C' = \{r, l\} \circ P_x \circ C_x$. Since $x \neq l$, we conclude that C' does not use the edge $\{l, s\}$, and hence, it is a cycle in $\mathcal{U}(D')$. We have to check that any bend of C' is admissible. For the bend at the root r and any bend on C_x this is obvious. As P_x is directed, any bend of P_x is admissible, and if x is a leaf, P_x must end with a tree edge. So the bend at x is also admissible. And hence, C' is harmful. □

Now we prove a similar lemma for the choice rule:

Lemma 7.2 *Suppose an application of Transformation Rule 3 to a dominance graph D generates two instances D_1 and D_2. Then $\mathcal{U}(D)$ contains a harmful cycle iff both $\mathcal{U}(D_1)$ and $\mathcal{U}(D_2)$ do.*

Proof. Let us assume that we apply the choice rule to two dominance edges (l_1, s) and (l_2, s) in D. This generates two amplifications $D_1 = D \cup (l_1, r_2)$ and $D_2 = D \cup (l_2, r_1)$, where r_1 and r_2 are the roots of the fragments containing l_1 and l_2 respectively (see Figure 7.2 on page 153). Since the edges of D are

a subset of the edges of D_1 and of the edges of D_2, we only have to prove that $\mathcal{U}(D)$ contains a harmful cycle if $\mathcal{U}(D_1)$ and $\mathcal{U}(D_2)$ do.

We consider a harmful cycle C_1 in $\mathcal{U}(D_1)$. If C_1 does not use the new edge $\{r_2, l_1\}$, then it is also a cycle in $\mathcal{U}(D)$. So we may suppose $C_1 = \{r_2, l_1\} \circ \{l_1, r_1\} \circ P_1$, because the bend at l_1 must be admissible. Similarly, we assume that $\mathcal{U}(D_2)$ contains a harmful cycle $C_2 = \{r_1, l_2\} \circ \{l_2, r_2\} \circ P_2$.

If P_1 or P_2 visits s, we can construct a harmful cycle in $\mathcal{U}(D)$: Suppose for some $i \in \{1, 2\}$ we have $P_i = P' \circ P''$ such that P' ends in s. Then $P' \circ \{s, l_i\} \circ \{l_i, r_i\}$ is a harmful cycle because P' avoids l_i and $\{l_i, r_i\}$ is a tree edge.

Hence, we may assume that both C_1 and C_2 avoid s. Let x denote the first node on P_1 different from r_1 that also lies on P_2. If $x = r_2$, then P_1 and P_2 have no common inner node, and hence $P_1 \circ P_2$ is a simple cycle. Since the endpoints of P_1 and P_2 are roots, we see that all bends are admissible, and we are done.

Now consider the case $x \neq r_2$. For $i \in \{1, 2\}$ we decompose P_i such that $P_i = Q_i \circ R_i$, Q_i ends at x and R_i starts at x (see Figure 7.6). Since P_i avoids l_i, we conclude that Q_i does not visit l_i. In particular we have $l_1 \neq x \neq l_2$. We want to prove that Q_1 also avoids l_2. Suppose P_1 visits l_2, otherwise there is nothing to show. Since the simple path P_1 ends at r_2 and all bends of P_1 are admissible, we can infer that the last edge on P_1 is the tree edge $\{l_2, r_2\}$. As Q_1 is a prefix of P_1 that ends in x and x is different from both l_2 and r_2, we conclude that l_2 is an inner node of R_1. So Q_1 avoids l_2. An analogous argument shows that Q_2 does not visit l_1. (Observe that $x \neq r_1$ by the choice of x.)

By construction, Q_1 and Q_2 have no common node but x. Denote the reversal of Q_1 by Q_1^{rev}. Then the cycle $C = \{r_1, l_1\} \circ \{l_1, s\} \circ \{s, l_2\} \circ \{l_2, r_2\} \circ Q_2 \circ Q_1^{\text{rev}}$ (see again Figure 7.6) is simple. If C is not harmful, then the bend at x – where Q_2 and Q_1^{rev} join – is forbidden. This means x is a leaf, and both Q_1 and Q_2 end with a dominance edge. Since the bend at x in C_2 is admissible, we have that R_2 starts with the tree edge incident to x. Therefore every bend on the cycle $Q_1 \circ R_2$ is admissible. By the choice of x, this cycle is also simple. And hence, it is harmful. □

We are ready to prove our characterization of solvability. The proof is based on the correctness of the enumeration algorithm from the previous section.

Theorem 7.1 *A dominance graph D is solvable iff $\mathcal{U}(D)$ does not contain a harmful cycle.*

Proof. Suppose first, $\mathcal{U}(D)$ contains a harmful cycle and we run our enumeration algorithm on D. Consider any instance D' which is generated

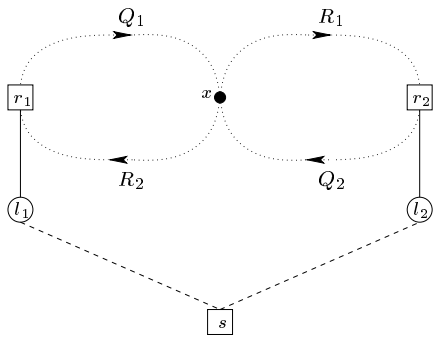


Figure 7.6: Situation in the proof of Lemma 7.2. (Observe that R_1 may visit l_2, hence $Q_1 \circ R_1 \circ \{r_2, l_2\} \circ \{l_2, s\} \circ \{s, l_1\} \circ \{l_1, r_1\}$ is in general not a simple cycle. An analogous observation can be made for R_2 and l_1.)

during this run by an application of a transformation rule. From the two previous lemmas we can conclude that $\mathcal{U}(D')$ contains a harmful cycle. And hence, D' cannot be a forest. Thus, `Enum-BruteForce(`D`)` will not report any solved form, which implies that D is not solvable.

Assume now that $\mathcal{U}(D)$ does not contain a harmful cycle. Again, we run our enumeration algorithm on D and trace its computations. Lemma 7.2 states that if the choice rule is applied to a graph without harmful cycle, then at least one of the two generated instances has no harmful cycle. And hence, we can inductively identify a branch in the computation of the enumeration algorithm where all generated dominance graphs are free of harmful cycles. Clearly, this branch cannot end with the discovery of a directed cycle. Thus the branch terminates when a solved form of D is found. □

7.3 An efficient harmful cycle test

In this section we will give a linear time algorithm which can test whether a given undirected dominance graph contains a harmful cycle. By Theorem 7.1, this implies that deciding solvability for a dominance graph can be done in linear time. This improves upon the results in [ADK+03].

The harmful cycle test will be developed gradually. We start with the well-known depth first search algorithm which can look for arbitrary cycles in undirected graphs. Then we make a simple modification which guarantees that the algorithm reports only harmful cycles. Unfortunately, this algorithm may miss harmful cycles, i.e. it may report that no harmful cycle exists,

although there is one. After another modification that takes care of this problem, we obtain a correct harmful cycle test.

DFS in undirected graphs

We forget about dominance graphs for the moment and consider arbitrary undirected multigraphs. We recall the depth-first search algorithm (DFS), which can be used to explore an undirected multigraph G in a systematic way. Then we explain how this algorithm can be augmented for cycle detection. We want to point out that we accept arbitrary cycles, i.e. we do not impose any constraints on the bends. When DFS explores G, it maintains for each node v its *status*, which can be *unreached*, *active* or *completed*: *unreached* means that v has not been discovered yet, *active* indicates that the exploration of v is in progress, *completed* marks the end of the exploration.

The procedure `DFS` (without cycle detection) works as follows: At the beginning the status of all nodes is set to *unreached*. Then `DFS` iterates over all nodes. Whenever it discovers a node v with status *unreached*, it calls the procedure `DFS-visit` to explore v. This procedure changes the status of v to *active* and scans every edge $e = \{v, w\}$ incident to v. Whenever it encounters an *unreached* node w during the scan, it makes a recursive call `DFS-visit`(w) to explore w. When the scan of v is finished, the status of v becomes *completed* and the call `DFS-visit`(v) returns.

We discuss a well-known modification of this basic scheme which allows to find cycles (see Algorithm 7.2). We use an array called *dfs_inedge*: For every recursive call `DFS-visit`(w), it stores the edge which caused the call (see line 10). If the respective call has been a top-level call (see line 4), then *dfs_inedge* is set to *none*.
We make a crucial observation: At any time the *active* nodes lie on a simple path P, and the edges on P are the *dfs_inedge*s of these nodes. We make this more precise. Every active node corresponds to a call of `DFS-visit` that has not returned yet. So we can order the active nodes $v_0, \ldots, v_k$ according to the call stack of `DFS-visit` such that `DFS-visit(`v_0`)` is the top-level call and `DFS-visit(`v_i`)` has made a recursive call to `DFS-visit(`v_{i+1}`)` for $i = 0, \ldots, k-1$. Then $P = [v_0, \mathit{dfs\_inedge}[v_1], v_1, \ldots, \mathit{dfs\_inedge}[v_k], v_k]$. Note that v_k is the node whose edges are currently scanned. Suppose now that we scan an edge $e \neq \mathit{dfs\_inedge}[v_k]$ (cf. line 7), and assume that e connects v_k with another active node v_i. Then we have discovered a simple cycle $C = P' \circ e$, where P' is the subpath of P from v_i to v_k. This explains lines 11 and 12 of the algorithm.

Let us apply the algorithm to the example on the left-hand side of Figure 7.7. Assume that we explore the node a first, and that edges are scanned

Algorithm 7.2 Finding arbitrary cycles with standard DFS

Procedure: DFS(G)
1: initialize the status of all nodes to *unreached*
2: **for all** nodes v of G **do**
3: **if** $status[v] = unreached$ **then**
4: $dfs\_inedge[v] \leftarrow none$; DFS-visit($v$)
5: report "no cycle found"

Procedure: DFS-visit(v)
6: $status[v] \leftarrow active$
7: **for all** edges e incident to v s.th. $e \neq dfs\_inedge[v]$ **do**
8: let w be the node adjacent to v via e
9: **case 1:** $status[w] = unreached$
10: $dfs\_inedge[w] \leftarrow e$; DFS-visit($w$)
11: **case 2:** $status[w] = active$
12: report "cycle found" and terminate
13: **otherwise:** do nothing
14: **end for**
15: $status[v] \leftarrow completed$

from left to right. On the right-hand side, we visualize the state of the algorithm at the time when it discovers the cycle: The node b is already completed (indicated by the thick circle), the nodes a, c, d and e are active (depicted by the double-circles) and the node f is still unreached. e is the last node on the path of active nodes, which means that the call `DFS-visit(`e`)` is currently executed. (This is why e is marked by two solid circles while the outer circles of the other active nodes are dashed.) For each node v with $status[v] \neq unreached$, we have marked $dfs\_inedge[v]$ by an arrowhead pointing towards v. The cycle is detected when the edge h is scanned, which leads from e back to the active node c.

We make another observation. Let us call a node v *reached* iff its status is *active* or *completed*. The *explored subgraph* of G is the subgraph which is induced by the reached nodes. It is easy to see that the algorithm computes a spanning forest $\mathcal{F}$ of the explored subgraph. The edges of the *DFS forest* $\mathcal{F}$ are the dfs_inedges of the reached nodes. We view $\mathcal{F}$ as a directed graph, for each node v in the forest $dfs\_inedge[v]$ is oriented towards v. Thus an edge (u, v) in $\mathcal{F}$ indicates that the call `DFS-visit(`u`)` has spawned a recursive call `DFS-visit(`v`)`. We say that u is a *DFS father* of v, and we call v a *DFS child* of u. If v is an ancestor of w in the DFS forest, then there is a unique

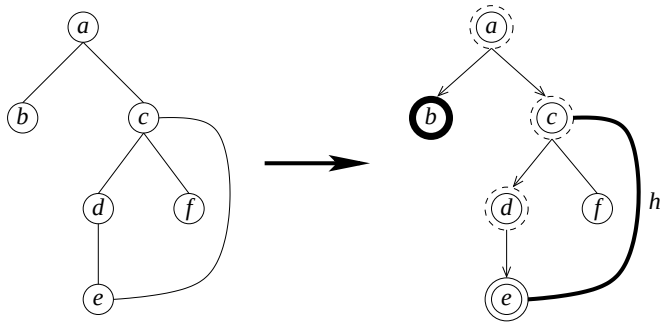


Figure 7.7: Discovering a cycle with standard DFS

path from v to w in $\mathcal{F}$, which we denote by $v \xrightarrow{\text{DFS}} w$. We freely identify the directed path $v \xrightarrow{\text{DFS}} w$ in $\mathcal{F}$ with the corresponding undirected path from v to w in G.

Naive harmful cycle test

Suppose now that we have an undirected dominance graph U. We want to modify Algorithm 7.2 such that only harmful cycles are reported. This means that a cycle C is only reported if all bends of C are admissible. The algorithm we obtain (see Algorithm 7.3) will be sound but not complete, i.e. it may overlook harmful cycles. (In the next section we will show an algorithm which is complete.)

Our first modification (in line 7 of Algorithm 7.3) ensures that the path of active nodes has only admissible bends. Consider a call `Naive-HC-visit(`v`)` (which corresponds to `DFS-visit(`v`)` in the standard algorithm). After v has been made active, the path P of active nodes ends in v. Whenever an edge e incident to v leads to a recursive call, then the path of active nodes is extended to $P' = P \circ e$. If we are dealing with a top-level call, then P is empty and P' consists of a single edge, which implies that it does not have any bend. So we do not have any restrictions on e in this case. If we are faced with a recursive call, then P ends with $dfs\_inedge[v]$, so P' has one more bend than P, namely $\langle dfs\_inedge[v], v, e\rangle$. Therefore we should scan an edge e only if $\langle dfs\_inedge[v], v, e\rangle$ is an admissible bend.[3]
Recall that we are in a top-level call iff $dfs\_inedge[v] = none$. Thus we can combine both cases into one by defining $\langle none, v, e\rangle$ to be admissible for any edge e incident to v.

[3]We want to point out that $\langle dfs\_inedge[v], v, dfs\_inedge[v]\rangle$ is not a bend, and hence $e = dfs\_inedge[v]$ is not admissible.

Algorithm 7.3 Naive harmful cycle test

Procedure: Naive-HC-Test(U)
1: initialize the status of all nodes to *unreached*
2: **for all** roots r of U **do**
3: **if** $status[r] = unreached$ **then**
4: $dfs\_inedge[r] \leftarrow none$; Naive-HC-visit($r$)
5: report "no harmful cycle found"

Procedure: Naive-HC-visit(v)
6: $status[v] \leftarrow active$
7: **for all** edges e incident to v s.th. $\langle dfs\_inedge[v], v, e\rangle$ is admissible **do**
8: let w be the node adjacent to v via e
9: **case 1:** $status[w] = unreached$
10: $dfs\_inedge[w] \leftarrow e$; Naive-HC-visit($w$)
11: **case 2:** $status[w] = active$ and $\langle e, w, \text{first edge of } w \overset{\text{DFS}}{\longrightarrow} v\rangle$ admissible
12: report "harmful cycle found" and terminate
13: **otherwise:** do nothing
14: **end for**
15: $status[v] \leftarrow completed$

But this modification is not enough. We also have to check the bend which is generated when we "close" the cycle (see line 11). This becomes clear when we look at the example in Figure 7.8. It shows a dominance graph containing a cycle, and it visualizes the state of the algorithm when it scans the edge k which closes the cycle. The bends at c and d are admissible because they are bends of the path of active nodes. The bend at e is also admissible, otherwise k would not have been scanned (cf. line 7). But the bend $\langle k, b, h\rangle$ is forbidden, because b is a leaf and k and h are dominance edges. This is captured by the condition "$\langle e, w, \text{first edge of } w \overset{\text{DFS}}{\longrightarrow} v\rangle$ admissible" in line 11. Thus our algorithm does not report a harmful cycle when applied to this example.

How can we check this condition efficiently? In order to determine the first edge f on the path $w \overset{\text{DFS}}{\longrightarrow} v$, we could trace the reverse path from v to w by means of the *dfs_inedge*s. But this may take a long time. We can do it in constant time: If w is a root, then the bend is admissible, no matter what f is. Otherwise, we look at $dfs\_inedge[w]$ and distinguish three cases: If $dfs\_inedge[w]$ is a tree edge (cf. node b in Figure 7.8), then f is a dominance edge, because there is only one tree edge incident to the leaf w (by Definition 7.1). Thus the bend is forbidden in that case. If $dfs\_inedge[w]$ is a dominance edge (see node d in Figure 7.8), then f must be the tree edge

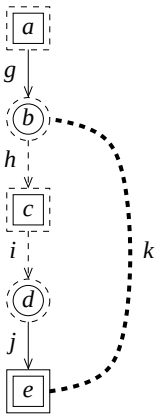


Figure 7.8: Example demonstrating why a second modification of Algorithm 7.2 is necessary.

incident to w, for $\langle dfs\_inedge[w], w, f\rangle$ is a bend of the path of active nodes. We conclude that the bend is admissible then.
But what can we do if $dfs\_inedge[w] = none$? The solution is to rule out this case by a small modification (see line 2): We make top-level calls only for root nodes. Since every leaf is adjacent to a root (by a tree edge), it is still guaranteed that every node of the graph is explored.

From the discussion above it should be clear that the algorithm is sound: When it reports a harmful cycle, then it has discovered a simple cycle and it has checked that every bend is admissible. Unfortunately, there exist examples like the dominance graph U in Figure 7.9. Although U contains a harmful cycle (indicated by the thick edges), Algorithm 7.3 *may* not find it. We say "may", because the outcome of `Naive-HC-Test` depends on the order in which the edges are scanned in line 7.

We trace a computation which fails to discover a harmful cycle. Assume the algorithm explores the root a first, i.e. it makes a top-level call `Naive-HC-visit(`a`)`. Then it scans the edge $\{a, b\}$ and makes a recursive call for b. Suppose this call scans $\{b, f\}$ first, which leads to a recursive call for f. The edge $\{f, e\}$ gives rise to a recursive call for e, the state of the algorithm at that time is shown as state ① in Figure 7.9.
Since $dfs\_inedge[e]$ is a dominance edge, the dominance edge $\{e, g\}$ is not scanned during that call. The path of active nodes can only be extended by the tree edge $\{e, d\}$, which causes a recursive call for d. During this call the edge $\{d, b\}$ is scanned (cf. state ② in the figure). This closes a cycle, but as the bend at b is forbidden, the cycle is ignored.
Then the recursive calls to d, e and f finish, and the algorithm returns to

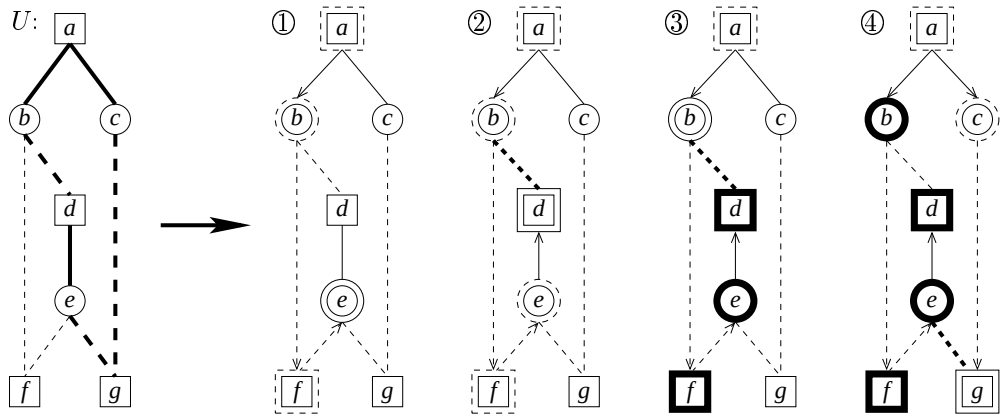


Figure 7.9: Example illustrating that Algorithm 7.3 may fail. (On he right-hand side four intermediate states of a failing computation are depicted.)

b. It scans the edge $\{b, d\}$ (see state ③), but it takes no action, because d is completed. The algorithm backtracks to a and makes a recursive call `Naive-HC-visit(`c`)`, which in turn spawns a recursive call for g. Then the edge $\{g, e\}$ is scanned for the first time (state ④). Since e is already completed, nothing happens. After that all calls of `Naive-HC-visit` return, and the algorithm reports that it has not found a harmful cycle.
We want to point out that a harmful cycle would have been found if $\{b, d\}$ had been scanned before $\{b, f\}$ in the call `Naive-HC-visit(`b`)`.

Correct harmful cycle test

We modify the naive harmful cycle test from the previous section such that we obtain a correct algorithm. Let us reconsider the example in Figure 7.9, in particular the state labelled ③: The algorithm is scanning the edge $\delta = \{b, d\}$ during the call `Naive-HC-visit(`b`)`. As d is already completed at that time, no action is taken. Observe that δ is an edge on the harmful cycle C in U, which suggests that δ should not be ignored. There is another edge on C that has been ignored so far: $\{e, g\}$ has not been scanned although e is already completed. Before we discovered δ, we knew only one path from b to e, namely $b \xrightarrow{\text{DFS}} e$. Since this path ends with a dominance edge, we skipped $\{e, g\}$ when e was active. With the discovery of the so-called *detour* δ there is an alternative path $Q = \delta \circ \{d, e\}$ from b to e ending with a tree edge. Thus $Q' = Q \circ \{e, g\}$ contains only admissible bends. As Q' is a subpath of C, we should do something to make the algorithm aware of it. The idea is to add a single dominance edge $\{b, g\}$ to the graph as a *short-cut* for Q'.

Before we formalize this idea, we discuss how the addition of $\{b, g\}$ influences the computation of the algorithm (see Figure 7.10). Suppose we are in state ③, discover the detour $\{b, d\}$ and add the short-cut $\{b, g\}$ (state ④′). Before b is declared completed, the short-cut[4] is scanned and a recursive call for g is invoked (state ⑤′). In this call the edges $\{g, e\}$ and $\{g, c\}$ are explored. The former causes no action, because e is completed. The latter leads to a recursive call for c (state ⑥′). The algorithm scans $\{c, a\}$ and reports a harmful cycle.

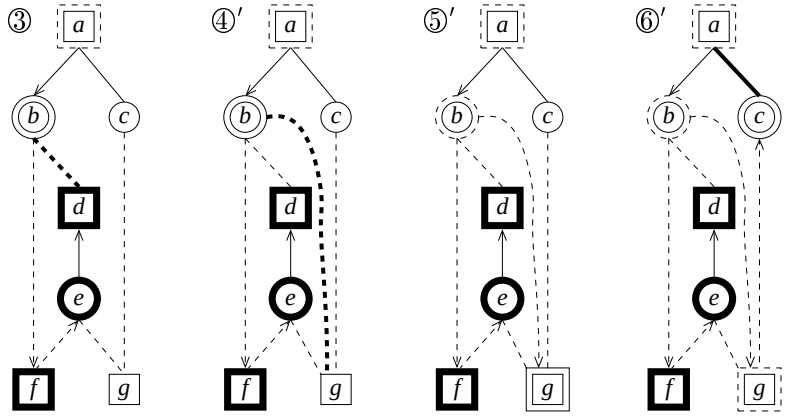


Figure 7.10: The computation of the harmful cycle test after adding the short-cut $\{b, g\}$.

In order to discover and handle detours we modify the harmful cycle test as follows: When we scan an edge $e = \{v, w\}$ in a call `HC-visit(`v`)`, we distinguish a third case (see lines 15 and 16 of Algorithm 7.4). If w is a completed root, we invoke the procedure `Collect`. As we shall see later, e is a detour and w is a descendant of v in the DFS forest. The task of `Collect` is to check the nodes on $v \xrightarrow{\text{DFS}} w$ for incident dominance edges that have not been scanned so far and to add the corresponding short-cuts. As the graph is altered by `Collect`, we create a working copy U of the original input U_{in} at the beginning of the algorithm (see line 1).

[4] We assume that the short-cut is appended to the adjacency lists of b and g, which guarantees that it will be scanned before b is declared completed.

Algorithm 7.4 The correct harmful cycle test

Procedure: HC-Test(U_{in})
1: $U \leftarrow$ copy of U_{in}
2: **for all** nodes v of U **do**
3: $\quad status[v] \leftarrow unreached$; $mark[v] \leftarrow false$ // *mark* is used by `Collect`
4: **for all** roots r of U **do**
5: $\quad$ **if** $status[r] = unreached$ **then**
6: $\quad\quad dfs\_inedge[r] \leftarrow none$; HC-visit($r$)
7: report "U_{in} contains no harmful cycle"

Procedure: HC-visit(v)
8: $status[v] \leftarrow active$
9: **for all** edges e incident to v s.th. $\langle dfs\_inedge[v], v, e\rangle$ is admissible **do**
10: $\quad$ let w be the node adjacent to v via e
11: $\quad$ **case 1:** $status[w] = unreached$
12: $\quad\quad dfs\_inedge[w] \leftarrow e$; HC-visit($w$)
13: $\quad$ **case 2:** $status[w] = active$ and $\langle e, w,$ first edge of $w \overset{\text{DFS}}{\longrightarrow} v\rangle$ admissible
14: $\quad\quad$ report "U_{in} contains harmful cycle" and terminate
15: $\quad$ **case 3:** $status[w] = completed$ and w is a root
16: $\quad\quad$ Collect(e) // collects previously forbidden dom. edges
17: $\quad$ **otherwise:** do nothing
18: **end for**
19: $status[v] \leftarrow completed$

Before we describe the procedure `Collect` we define the notion of a detour:

Definition 7.3 (detour) *An edge δ incident to a leaf v and a root w is called a* detour *if the following holds:*

- *The node v is an ancestor of w in the DFS forest. All nodes on $v \xrightarrow{\text{DFS}} w$ except for v are completed.*
- *The edge δ is a dominance edge, $dfs\_inedge[v]$ is a tree edge, and $\delta \neq dfs\_inedge[w]$. (Thus δ does not belong to the DFS forest.)*

Suppose `Collect` is called for a detour δ incident to a leaf v and a root w. The procedure traverses the nodes on $v \xrightarrow{\text{DFS}} w$ in reverse order, i.e. it starts in w and walks up in the DFS forest until it reaches v. This can be done with the aid of the *dfs_inedge*s, because $dfs\_inedge[x]$ connects a node x with its father in the DFS forest.
Every node x (different from v) on $v \xrightarrow{\text{DFS}} w$ is checked for incident dominance edges that have not been scanned so far. By Definition 7.3, x is completed. So if x is a root, or if x is a leaf and $dfs\_inedge[x]$ is a tree edge, then all edges have already been scanned. But if x is a leaf and $dfs\_inedge[x]$ is a dominance edge, then only the tree edge incident to x has been scanned (cf. Figure 7.11). For every dominance edge $e = \{x, y\}$ incident to x with $e \neq dfs\_inedge[x]$, we add the dominance edge $e' = \{v, y\}$ to the graph, i.e. we append e' to adjacency lists of both v and y. We call e the *origin* of e'. Observe that e' is a short-cut for the path $\delta \circ (x \xrightarrow{\text{DFS}} w)^{\text{rev}} \circ e$, which contains only admissible bends (in particular at x).

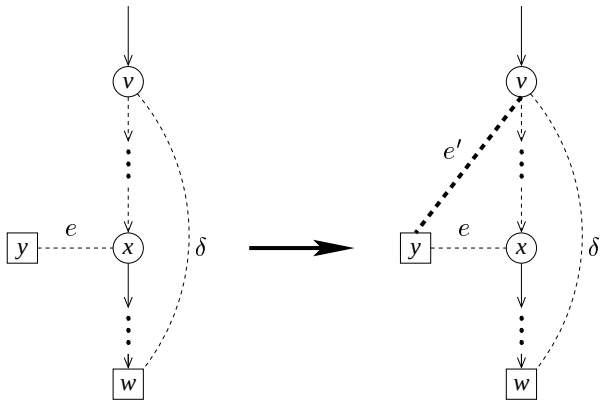


Figure 7.11: Addition of a short-cut e' for the path $\delta \circ (x \xrightarrow{\text{DFS}} w)^{\text{rev}} \circ e$.

It will turn out that it suffices to process every node in U at most once by `Collect`. Therefore we mark every node x that is examined (see line 5)

Algorithm 7.5 Collect previously forbidden dominance edges

Procedure: Collect(δ)
Require: δ is a detour
1: let v be the leaf and w be the root incident to δ
2: $x \leftarrow w$
3: **repeat**
4: **if** $mark[x] = false$ **then**
5: $mark[x] \leftarrow true$
6: **if** x is a leaf and $dfs\_inedge[x]$ is a dominance edge **then**
7: **for all** dom. edges $e = \{x, y\}$ inc. to x s.th. $e \neq dfs\_inedge[x]$ **do**
8: add the dominance edge $e' = \{v, y\}$ to U
9: $origin[e'] \leftarrow e$
10: **end for**
11: **end if**
12: $x \leftarrow$ father of x in the DFS forest
13: **until** $x = v$
14: remove δ from U // only needed for the proofs

and we skip marked nodes (cf. line 4).
Before the procedure terminates, it deletes the detour δ from U. This is only needed for the correctness proof. At the time of its removal, δ has already been scanned twice (first by `HC-visit(`w`)` and later by `HC-visit(`v`)`). Hence, it would not be scanned again anyway. So in a practical implementation of the algorithm we can keep δ, and we do not have to store the origins of the short-cuts (see line 9), as this information is only used in the proofs.

Correctness

Now we prove the correctness of the harmful cycle test. We begin by showing that the precondition of the procedure `Collect` is never violated:

Lemma 7.3 *Whenever `Collect(`e`)` is called in line 16 of Algorithm 7.4, e is a detour.*

Proof. Let v be the leaf and w be the root incident to e. Since w is completed when e is scanned during the call `HC-visit(`v`)`, we have that w is neither the DFS father nor a DFS child of v, i.e. $dfs\_inedge[v] \neq e \neq dfs\_inedge[w]$.

An induction on the number of calls to `Collect` shows that $dfs\_inedge[v]$ is a tree edge and v is a DFS ancestor of w: Assume first that $e \in U_{\text{in}}$ (base case). Then e has already been scanned in the call `HC-visit(`w`)`. Let

us consider the state of the algorithm at that particular point in time: v must have been reached (otherwise v would have become a DFS child of w), but not completed (for v is still active now). Thus v has been active then, which implies that v is a DFS ancestor of w. Since the call `HC-visit(`w`)` has not reported a harmful cycle, $\langle e, v, \text{first edge on } v \xrightarrow{\text{DFS}} w \rangle$ is not admissible (see line 13 of Algorithm 7.4 and observe that the roles of v and w are interchanged). Thus $dfs\_inedge[v]$ is a tree edge.
Suppose now that e has been added by a call `Collect(`δ`)` (induction step). This implies that δ is incident to v and some root r. Let $\tilde{e} = origin[e]$, $\tilde{e}$ is incident to w and some leaf x. By the induction hypothesis, δ is a detour. Therefore, the situation is as shown in Figure 7.12: $dfs\_inedge[v]$ is a tree edge, and x is a node on $v \xrightarrow{\text{DFS}} r$. We observe that $\tilde{e}$ is present in the original graph U_{in} (because $dfs\_inedge[x]$ is a dominance edge). So $\tilde{e}$ has been scanned during the call `HC-visit(`w`)`. When this call terminated, x and its DFS ancestor v must have been active or completed. Since v is still active now, it must have been active when the call for w was made. Hence, v is a DFS ancestor of w.
The fact that $dfs\_inedge[v]$ is a tree edge implies that e is a dominance edge, because there is only one tree edge incident to v. Since v is the last node on the path of active nodes, when `Collect(`e`)` is called, all its proper DFS descendants are completed, in particular those on $v \xrightarrow{\text{DFS}} w$. □

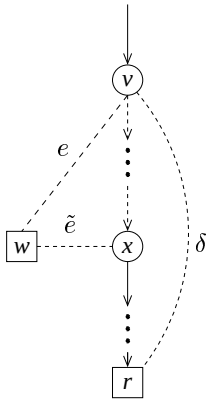


Figure 7.12: Situation in the induction step of the proof of Lemma 7.3.

In the sequel we will show a crucial property of the procedure `Collect`. Consider a call `Collect(`δ`)`. Let U_1 be the graph before the call and let U_2 denote the graph after the call. Then U_1 contains a harmful cycle if and only if U_2 does. We break this claim into two lemmas and prove them separately.

Lemma 7.4 *If there is a harmful cycle in U_2, then there is one in U_1.*

Proof. Suppose the statement is false and let C be a shortest harmful cycle in U_2. C must use one edge $e' = \{v, y\}$ added by `Collect(`δ`)`. Let $e = \mathit{origin}[e'] = \{x, y\}$. If C does not visit any node different from v on the path $v \xrightarrow{\text{DFS}} w$, we can replace e' by $\delta \circ (x \xrightarrow{\text{DFS}} w)^{\text{rev}} \circ e$ and obtain a harmful cycle in U_1.

Otherwise we assume that C starts in v with the edge e'. Since all nodes on $v \xrightarrow{\text{DFS}} w$ (except for v) are marked after the call and C ends with $\mathit{dfs\_inedge}[v]$, we can decompose $C = e' \circ P \circ \{s, t\} \circ Q$ such that s is a marked DFS descendant of v and Q avoids any marked descendants of v. The cycle $v \xrightarrow{\text{DFS}} s \circ \{s, t\} \circ Q$ is simple and uses only edges which are in both U_2 and U_1, because it avoids e' as well as any other edge added by `Collect(`δ`)`.
If it is not harmful, then the bend $\langle \mathit{dfs\_inedge}[s], s, \{s, t\}\rangle$ is not admissible, i.e. s is a leaf and both $\mathit{dfs\_inedge}[s]$ and $g = \{s, t\}$ are dominance edges. If s has been marked during the call `Collect(`δ`)`, a short-cut $g' = \{v, t\}$ with origin g has been added; so $g' \circ Q$ would be a shorter harmful cycle in U_2. Otherwise, a short-cut $g'' = \{v', t\}$ has been added to some DFS ancestor v' of s by an earlier call. As s is marked but v is not, v' cannot be a proper ancestor of v. Since s is a descendant of both v and v', v must be an ancestor of v'. The edge g'' still exists in U_2, otherwise t would be a marked descendant of v' and hence of v. So $v \xrightarrow{\text{DFS}} v' \circ g'' \circ Q$ is a harmful cycle that does not use any of the edges added by `Collect(`δ`)`. Hence, it is also contained in U_1. □

Lemma 7.5 *If there exists a harmful cycle in U_1, then there is one in U_2.*

Proof. Let C be a harmful cycle in U_1. If C avoids δ, there is nothing to show. Otherwise we assume that C starts in v with the edge δ. We decompose $C = \delta \circ P \circ \{s, t\} \circ Q$ such that s is a marked DFS descendant of v and Q avoids any marked descendants of v. (Note that w is a candidate for s.) The cycle $v \xrightarrow{\text{DFS}} s \circ \{s, t\} \circ Q$ is simple and avoids δ. Hence, it is contained in U_2. If it is not harmful, then s is a leaf and both $\mathit{dfs\_inedge}[s]$ and $g = \{s, t\}$ are dominance edges. Therefore a short-cut $g' = \{v', t\}$ has been added to some (not necessarily proper) DFS descendant v' of v. The edge g' still exists, otherwise t would be a marked descendant of v' and hence of v. So $v \xrightarrow{\text{DFS}} v' \circ g' \circ Q$ is a harmful cycle in U_1 that does not use δ. Hence, it is also contained in U_2. □

Combining the previous two lemmas, the next lemma follows by an easy induction on the number of invocations of `Collect`:

Lemma 7.6 *At any time the current graph U contains a harmful cycle iff the original graph U_{in} contains one.*

This implies that the algorithm is sound. When it reports a harmful cycle, then the current graph U contains one. This follows immediately from the fact that `Collect` never deletes an edge that is in the DFS forest (cf. Definition 7.3). By Lemma 7.6, the original graph U_{in} contains a harmful cycle.

It remains to prove that the algorithm is complete: When no harmful cycle is reported, then U_{in} contains none. Let $U^\star$ denote the current graph U when the algorithm terminates. By Lemma 7.6, it suffices to show that $U^\star$ does not contain a harmful cycle.

In the subsequent proofs we will argue about the completion times of the nodes. For a node v in $U^\star$ denote by $ct(v)$ the time when $status[v]$ is set to *completed*. We do need the exact times, we only compare time stamps of different nodes. If the algorithm terminates without reporting a cycle, then we can make some important observations about the relation of the completion times of a node v and its adjacent nodes:

Lemma 7.7 *Assume Algorithm 7.4 terminates without reporting a harmful cycle. Let v be a node in $U^\star$, and denote by u its DFS father (if it exists). Then $ct(u) > ct(v)$, and the following holds:*

1. *If v is a root:*
   For every node w adjacent to v with $w \neq u$, we have $ct(v) > ct(w)$.

2. *If v is a leaf and $dfs\_inedge[v]$ is a tree edge:*
   For every node w adjacent to v with $w \neq u$, we have $ct(v) > ct(w)$.

3. *If v is a leaf and $dfs\_inedge[v]$ is a dominance edge:*
   Let r be the root that is adjacent to v by the tree edge. For every node w adjacent to v with $w \neq r$, we have $ct(r) < ct(v) < ct(w)$.

The possible constellations are shown in Figure 7.13.

Proof. It is clear that $ct(u) > ct(v)$, because the call `HC-visit(`v`)` is invoked by `HC-visit(`u`)`.

Case 1: Assume that v is a root and there is an edge $e = \{v, w\}$ with $w \neq u$ and $ct(v) < ct(w)$. Suppose first that e already existed, before v was completed. Then e is scanned during the call `HC-visit(`v`)`. At that time w must have been active (any other status would imply $ct(v) > ct(w)$). So w is a DFS ancestor of v. Since no harmful cycle has been reported due to this scan of e, we conclude that $\langle e, w, \text{first edge on } w \xrightarrow{\text{DFS}} v \rangle$ is not admissible, i.e. $dfs\_inedge[w]$ is a tree edge. This implies that e is identified as a detour when scanned again later during the call `HC-visit(`w`)`. So e is deleted by

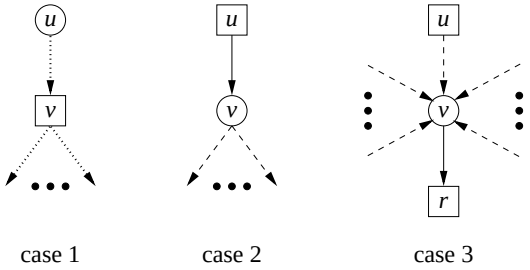


Figure 7.13: The constellations in $U^\star$ described in Lemma 7.7, the arrowhead of each edge points to the incident node with the smallest completion time. (Dotted edges may represent a tree edge or a dominance edge.)

`Collect`, a contradiction.
Assume now that e is added as a short-cut after v has been completed. In this case e is also recognized as a detour and deleted, again a contradiction.
Case 2: Suppose that v is a leaf, *dfs_inedge*$[v]$ is a tree edge, and there is an edge $e = \{v, w\}$ with $w \neq u$ and $ct(v) < ct(w)$. No matter if e is a short-cut or not, we know that e is scanned during the call `HC-visit(`v`)`. At that time w must have been active (otherwise $ct(v) > ct(w)$). But as w is a root, a harmful cycle would have been discovered then, a contradiction.
Case 3: Let v be a leaf such that *dfs_inedge*$[v]$ is a dominance edge, and let r denote the unique root adjacent to v by a tree edge. The inequality $ct(v) > ct(r)$ follows from the fact that v is the DFS father of r.
Assume now that there is an edge $e = \{v, w\}$ with $w \neq r$ and $ct(v) > ct(w)$. We know that e is no short-cut in this case. Thus e is scanned during the call `HC-visit(`w`)`. At that time v must have been active (any other status would imply $ct(w) > ct(v)$). Hence, v is a DFS ancestor of w, and the first edge f on $v \xrightarrow{\text{DFS}} w$ is the tree edge incident to v and r. Therefore, the bend $\langle e, v, f \rangle$ is admissible, which implies that a harmful cycle would have been reported, a contradiction. □

Now we are ready to finish the completeness proof.

Lemma 7.8 *If Algorithm 7.4 reports that U_{in} contains no harmful cycle, then U_{in} does not contain one.*

Proof. Suppose otherwise. Then the algorithm terminates with a graph $U^\star$, which contains a harmful cycle $C = [v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k = v_0]$ (cf. Lemma 7.6). We may assume that $ct(v_0)$ has the largest completion time of all nodes on C. Then we can show by induction that $ct(v_i) > ct(v_{i+1})$ for $i = 0, \ldots, k-1$. For $i = 0$ this is obvious. So suppose $i > 0$ and

$ct(v_{i-1}) > ct(v_i)$. If v_i is a root, or if v_i is a leaf s.th. *dfs_inedge*$[v_i]$ is a tree edge, then v_{i-1} is the only neighbour of v_i with a larger completion time (cf. cases 1 and 2 of Lemma 7.7). And the claim follows.
If v_i is a leaf and *dfs_inedge*$[v_i]$ is a dominance edge, then e_i must also be a dominance edge (see case 3 of Lemma 7.7). Since C is harmful, the bend $\langle e_i, v_i, e_{i+1}\rangle$ is admissible, i.e. e_{i+1} is the tree edge incident to v_i. By Lemma 7.7, we have $ct(v_i) > ct(v_{i+1})$.

Our inductive proof shows $ct(v_{k-1}) > ct(v_k) = ct(v_0)$, which contradicts the assumption that v_0 has the largest completion time. □

Implementation and runtime analysis

We discuss how to implement the harmful cycle test for an undirected dominance graph U_{in} such that it runs in time $O(n+m)$, where n is the number of nodes and m is the number of edges of U_{in}. In order to achieve this running time we have to refine the current implementation of the procedure `Collect` (see Algorithm 7.5). The problem is that `Collect` may visit a marked node several times. Recall that `Collect` walks up in the DFS forest from a node w to an ancestor v, which is not marked[5]. Since marked nodes are always skipped during this walk, we can save time if we are able to jump quickly from a node x to its closest unmarked ancestor in the DFS forest.

Algorithm 7.6 Improved implementation of `Collect` with Union-Find

Procedure: Collect(δ)
1: let v be the leaf and w be the root incident to δ
2: $x \leftarrow$ `Find(`w`)`
3: **while** $x \neq v$ **do**
4: $f \leftarrow$ father of x in the DFS forest
5: *mark*$[x] \leftarrow$ *true*; `Union(`x, f`)` // *mark* is used for explanation only
6: **if** x is a leaf and *dfs_inedge*$[x]$ is a dominance edge **then**
7: **for all** dom. edges $e = \{x, y\}$ inc. to x s.th. $e \neq$ *dfs_inedge*$[x]$ **do**
8: add the dominance edge $e' = \{v, y\}$ to U
9: **end if**
10: $x \leftarrow$ `Find(`f`)`
11: **end while**

For this purpose, we use a `Union-Find` data structure in our improved implementation of `Collect` (see Algorithm 7.6). We maintain a partition of the nodes of U_{in}. Every set S in the partition has a unique representative

[5] v is not marked because v is active and marked nodes are always completed.

r_S. We have the invariant that all nodes in $S \setminus r_S$ are marked, and r_S is the closest unmarked DFS ancestor of each of them. Thus we can determine the closest unmarked ancestor of a node x by calling `Find(`x`)`. At the beginning of the cycle test, every node forms a singleton set. Whenever a node x gets marked, we unite the set of x with the set of its DFS father f (see line 5); the representative of the union is the representative of the former set of f. This establishes the invariant again.
Since the `Union` operations are not applied to arbitrary sets – we always unite the set of a node with the set of its DFS father – we can use the incremental tree disjoint set union algorithm by Gabow and Tarjan [GT85]. Thus the total time consumed by the `Union-Find` data structure is bounded by $O(n)$ plus the number of invocations of `Collect`.

Skipping marked nodes ensures that every dominance edge in U_{in} can be the origin of at most one short-cut. Since the origin of each short-cut is an edge in U_{in}, we conclude that the number of edges in the graph U is bounded by $2m$. It easy to see that `HC-visit` is invoked once for each node and that every edge is scanned at most twice. So we conclude that the total running time of the harmful cycle test is $O(n + m)$, which implies the following theorem:

Theorem 7.2 *Solvability of a directed dominance graph D can be decided in time $O(m)$, where m is the number of edges of D.*

Proof. Follows immediately from Theorem 7.1 and the observation that D has at most $m/2$ nodes. □

Chapter 8

Enumeration of solved forms

In this chapter we describe how the minimal solved forms of a dominance graph D can be enumerated efficiently. We discuss two algorithms. The first algorithm is obtained by plugging the efficient solvability check from the previous chapter into the brute-force enumeration algorithm from Section 7.1. This algorithm, called `Enum1`, runs in time $O(m + N \cdot n^2 m)$, where N is the number of minimal solved forms, n is the number of nodes and m is the number of edges of D. Then we give an example showing that this time bound is tight for `Enum1`. The example gives rise to an improved algorithm `Enum2`. Its running time is $O(m + N \cdot nm)$.

8.1 An efficient enumeration algorithm

The efficient solvability test from the previous section allows us to improve the brute force enumeration algorithm (Algorithm 7.1 on page 154) considerably. In order to obtain an efficient algorithm (cf. Algorithm 8.1), we replace the test for a directed cycle in D by the test for a harmful cycle in $\mathcal{U}(D)$ (see line 1).

We analyse the running time of Algorithm 8.1. Let us discuss how to compute the transitive reduction of D (in line 2) efficiently. It is well-known that this can be done in time $O(nm)$ (see [GK79, Sim88]), where n is the number of nodes and m is the number of edges. But only the top-level call needs to do the full-fledged reduction. The graphs processed in the recursive calls have been generated from a reduced graph D by adding a single irredundant edge (l, s) (cf. Transformation Rule 3 on page 153). We show how to exploit this to compute the transitive reduction of $D \cup (l, s)$ much faster.
Adding (l, s) makes an edge (v, w) redundant iff there is a path from v to l

Algorithm 8.1 Efficient enumeration of minimal solved forms

Procedure: Enum1(D)
1: **if** $\mathcal{U}(D)$ contains no harmful cycle **then**
2: eliminate all redundant dominance edges
3: **if** D has a root s with at least two incoming dominance edges **then**
4: apply the choice rule and generate two new instances D_1 and D_2
5: Enum1(D_1); Enum1(D_2)
6: **else** // D is in solved form
7: report D
8: **end if**
9: **end if**

and a path from s to w in D. So we mark all nodes in D that can reach l with red colour and all nodes that can be reached from s with green colour. Then we delete all edges where the source node is red and the target node is green. Finally, we add (l, s). This yields a reduced graph and can be done in time $O(m)$.

The running time of the top-level call `Enum1(`D`)` – without the time consumed by recursive calls – is $O(nm)$. The size of the dominance graphs can only decrease as the recursion depth increases. An application of the choice rule adds one edge, but it makes at least one edge redundant. Thus the number of edges can only decrease, which implies that the time spent by a recursive call is $O(m)$.

Consider a call `Enum1(`D'`)` such that D' is solvable but not a solved form itself. This call spawns two recursive calls `Enum1(`D'_1`)` and `Enum1(`D'_2`)`. If D'_2 is not solvable, then the latter call terminates immediately and D'_1 must be solvable. We charge the time for processing D'_2 to its solvable "sibling" D'_1. This shows that our analysis does not have to take into account recursive calls for unsolvable graphs.

If `Enum1` is applied to a solvable graph, it eventually reports at least one minimal solved form D_s. Let us charge the time spent by this application to D_s. As we have seen before, the recursion depth is bounded by n^2 (the maximum size of the reachability relation). So each reported solved form gets a total charge of $O(n^2 m)$. Denote by N the number of minimal solved forms of D. The total running time of `Enum1(`D`)` is $O(m + N \cdot n^2 m)$. (Note that the total running time is $O(m)$, if D is unsolvable.)

Observe that N may be exponential in the size of D, as shown by the example in Figure 8.1. On the left-hand side we see a dominance graph D that consists of $\Theta(k)$ nodes and edges and has $k + 2$ tree fragments.

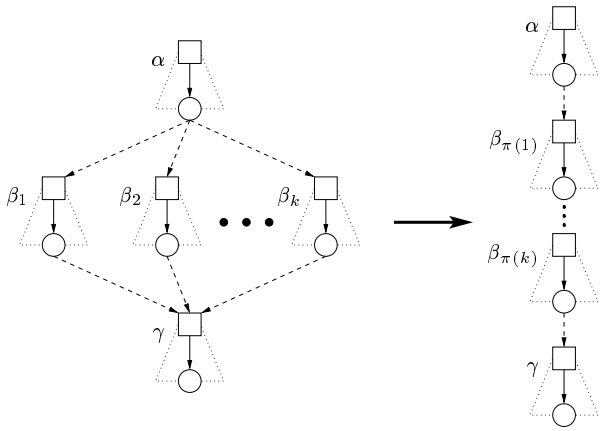


Figure 8.1: Example showing that N may be exponential in the size of D.

Each permutation π of $[1;k]$ corresponds to a different solved form of D (cf. right-hand side): α is the topmost and γ is the bottommost fragment, the fragments $\beta_1, \ldots, \beta_k$ are arranged according to π. So D has $k!$ solved forms. Hence, the running time of our enumeration algorithm may be exponential in the size of its input. But it is output sensitive, its running time is polynomial in the sum of the sizes of its input and output.

A worst case example for `Enum1`

We discuss an example which shows that the bounds given above are tight for Algorithm 8.1. We define a sequence $D^{(1)}, D^{(2)}, D^{(3)}, \ldots$ of dominance graphs. $D^{(k)}$ has $\Theta(k)$ nodes and $\Theta(k^2)$ edges, it has only one solved form, and `Enum1` may make $\Theta(k^2)$ applications of the choice rule to find it. We say "may", because the number of applications depends on the selection of the root and the two dominance edges to which the choice rule is applied.

The graph $D^{(k)}$ has $2k$ tree fragments (see Figure 8.2): α_1, ..., α_k, β_2, ..., β_k and γ. The fragment α_i has the root u_i and two leaves v_i and w_i, the fragment β_j consists of a root x_j and a leaf y_j, and the fragment γ has the root r and the leaf l. $D^{(k)}$ has the following dominance edges: (v_i, r) for $i = 1, \ldots, k$, (w_i, x_i) for $i = 2, \ldots, k$ and (v_i, x_j) for $1 \leq i < j \leq k$.

Clearly, $D^{(1)}$ is in solved form, so assume $k > 1$. We apply the choice rule exhaustively to the root r: For $i = 1, \ldots, k-1$ we apply the rule to the edges (v_k, r) and (v_i, r). This generates two amplifications G_i (by adding (v_k, u_i)) and H_i (by adding (v_i, u_k)). G_i is not solvable, because $\mathcal{U}(G_i)$ contains the harmful cycle $\{v_k, u_i\} \circ \{u_i, v_i\} \circ \{v_i, x_k\} \circ \{x_k, w_k\} \circ \{w_k, u_k\} \circ \{u_k, v_k\}$. So the recursive call for G_i terminates immediately, and the computation proceeds

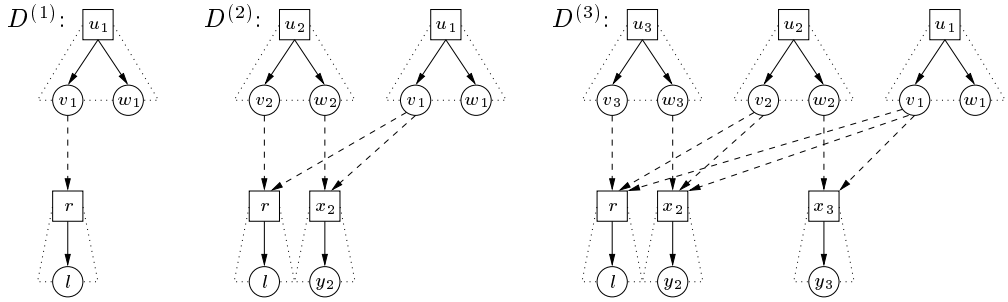


Figure 8.2: The first three graphs in our worst case example for `Enum1`.

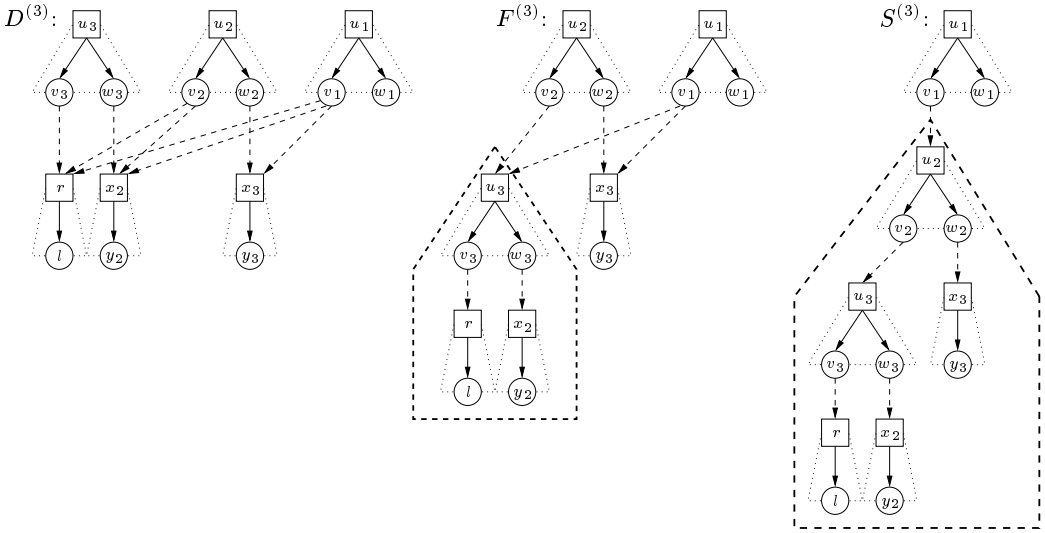


Figure 8.3: A possible computation of `Enum1` applied to $D^{(3)}$.

with H_i. In H_i, the edge (v_i, r) is redundant, the edge (v_k, r) remains. After $k-1$ applications of the choice rule we have transformed $D^{(k)}$ into a graph $F^{(k)}$, where the indegree of r is one. We can see this transformation in Figure 8.3 for $D^{(3)}$.

The subtree rooted at u_k in $F^{(k)}$ is in solved form (it is even a configuration). If we replace this subtree with a tree fragment consisting of two nodes, we obtain the graph $D^{(k-1)}$. (We invite the reader to compare $F^{(3)}$ in Figure 8.3 with $D^{(2)}$ in Figure 8.2.) An easy induction shows that $D^{(k)}$ has a unique solved form $S^{(k)}$ and that `Enum1` may have to apply the choice rule $(k-1)+(k-2)+\ldots+1 = \binom{k}{2}$ times to find it. $S^{(k)}$ is in some sense similar to $D^{(1)}$: If the subtree rooted at u_2 in $S^{(k)}$ is replaced by a fragment with

two nodes, one obtains $D^{(1)}$ (see Figure 8.3).

8.2 An improved enumeration algorithm

In this section we show how to improve the worst case running time of the enumeration algorithm. We will prove that a solved form of a dominance graph D can be constructed in time $O(nm)$ and all N minimal solved forms can be found in time $O(m+N \cdot nm)$, which shaves off a factor of n compared to our previous results.

We reconsider the example in Figure 8.2 and analyse why `Enum1` has spent so much time to solve it. In order to reduce the indegree of the root r in $D^{(k)}$ from k to one, the choice rule is applied $k-1$ times, and eventually only one solvable graph remains. But $k-1$ unsolvable dominance graphs are generated and to recognise unsolvability the harmful cycle test is called for each of them. Our goal is to make only the "necessary" choices, which means that we do not generate unsolvable instances anymore. To be more precise, we are given a root s in a solvable dominance graph D and we want to compute a sequence $D_1, \ldots, D_h$ of **solvable** amplifications of D with the following property: $\mathcal{S}(D) = \mathcal{S}(D_1) \dot\cup \ldots \dot\cup \mathcal{S}(D_h)$, and the indegree of s in D_i is one for $i = 1, \ldots, h$. We will do this in time $O(h \cdot m)$.

Suppose we apply the harmful cycle test to $\mathcal{U}(D)$. Since D is solvable, it will not report a harmful cycle. From the DFS forest that is computed we will be able to deduce the solvability or unsolvability of certain amplifications of D without generating them.

Let us look at a tree in the DFS forest that is generated by the top-level call `HC-visit(`s`)`, and assume that s has a DFS child l such that $dfs\_inedge[l]$ is a dominance edge (see left-hand side of Figure 8.4). Thus the only DFS child of l is the root r of the fragment of l, because the bend at l must be admissible. Assume further that the root r' is a DFS descendant of l. Then the edge $\{l, r'\}$ cannot belong to $\mathcal{U}(D)$, otherwise $\mathcal{U}(D)$ would contain a harmful cycle which contradicts the assumption that D is solvable. We conclude that $\tilde{D} = D \cup (l, r')$ is not solvable.

Suppose further that there is a leaf l' in the fragment of r' which is adjacent to s by a dominance edge (see right-hand side of the figure). If we apply the choice rule to the edges (l, s) and (l', s) it generates the graphs $\tilde{D}$ and $\hat{D} = D \cup (l', r)$. As we have just seen, $\tilde{D}$ is unsolvable. So every solved form of D is an amplification of $\hat{D}$. Hence, we can add (l', r) to D and remove the redundant edge (l', s) without changing the set of solved forms of D. Thus we reduce the indegree of s while preserving the solved forms.

The results from above are expressed in the following lemma:

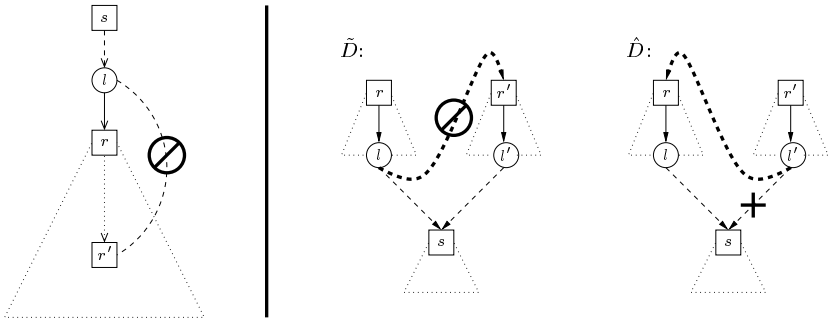


Figure 8.4: Left: DFS tree generated by `HC-visit(`s`)`; the marked edge $\{l, r\}$ cannot belong to $\mathcal{U}(D)$, if D is solvable. Right: Choice rule applied to (l, s) and (l', s).

Lemma 8.1 *Suppose that Algorithm 7.4 is applied to $\mathcal{U}(D)$ of a solvable dominance graph D. Assume that a top-level call `HC-visit`(s) is made. Let l be a leaf such that $dfs\_inedge[l] = \{l, s\}$ is a dominance edge. If l' is a proper DFS descendant of l and r' denotes the root of the fragment of l' in D, then $\tilde{D} = D \cup (l, r')$ is not solvable.*
Assume further that l' is adjacent to s. Then we have for every solved form D_s of D that $(l', r) \in \mathrm{Reach}(D_s)$, where r denotes the root of the fragment of l. Hence, we can add (l', r) to any amplification D' of D without changing the set of solved forms, i.e. $\mathcal{S}(D' \cup (l', r)) = \mathcal{S}(D')$.

Proof. Since no harmful cycle is reported by the algorithm, the tree edge $\{r', l'\}$ becomes a DFS forest edge. So r' is the DFS father of l' (if $dfs\_inedge[l']$ is the tree edge) or the DFS child of l' (if $dfs\_inedge[l']$ is a dominance edge). Hence, r' is a DFS descendant of l.

Let $U = \mathcal{U}(D)$ and $\tilde{U} = \mathcal{U}(\tilde{D})$. We compare computation of the calls `HC-Test(`U`)` and `HC-Test(`$\tilde{U}$`)`. We assume that the iterations in Algorithm 7.4 (see lines 4 and 9) process the iterated items according to some deterministic order, so that the computations of the two calls are parallel as long as possible. Thus both calls make the same computations until the edge $e = \{l, r'\}$ is scanned for the first time during the harmful cycle test for $\tilde{U}$. Observe that e cannot be the origin of a short-cut before l is completed. Since the bend $\langle dfs\_inedge[l], l, e\rangle$ is forbidden (see left-hand side of Figure 8.4), e is scanned only during the call `HC-visit(`r'`)`, although l becomes active before r'. As l is still active then, the algorithm reports a harmful cycle in $\tilde{U}$. Therefore $\tilde{D}$ is not solvable.

Suppose that l' is adjacent to s, i.e. D contains the dominance edge (l', s). Applying the choice rule to the edges (l, s) and (l', s) generates two graphs.

One of these graphs is $\tilde{D}$, the other is $\hat{D} = D \cup (l', r)$. We know that $\mathcal{S}(D) = \mathcal{S}(\tilde{D}) \dot{\cup} \mathcal{S}(\hat{D})$ (see page 152). As $\mathcal{S}(\tilde{D})$ is empty, every solved form D_s of D is an amplification of $\hat{D}$, which proves the rest of the claim. □

The lemma above allows us to reduce the indegree of s to one if s has only one DFS child (that is adjacent to it by a dominance edge). Now we consider two DFS children l_1 and l_2 of s and examine the solvability of $D_1 = D \cup (l_1, r_2)$ and $D_2 = D \cup (l_2, r_1)$ (see left-hand side of Figure 8.5). Observe that these graphs are the two instances generated by applying the choice rule to (l_1, s) and (l_2, s). We assume that l_1 is completed before l_2 and no other DFS child of s is completed in between.

Suppose we add the edge $d_1 = \{l_1, r_2\}$ to $U = \mathcal{U}(D)$ and obtain U_1 (as in the middle of the figure). If we invoke `HC-visit(`s`)` on U_1, the recursive call for l_1 ignores d_1, because $dfs\_inedge[l_1]$ is a dominance edge. So does the recursive call for r_2, because l_1 is already completed at that time. Hence, D_1 is solvable.

The situation is more complicated if we add $d_2 = \{l_2, r_1\}$ to U, which yields U_2 (see right-hand side of the figure). We know that D_2 is unsolvable iff U_2 contains a harmful cycle. It will turn out that this is the case iff during the call `HC-visit(`l_2`)` (including recursive calls spawned by this call) an edge $\{r', l_1\}$ is scanned.

Hence, the call `HC-visit(`s`)` provides us with enough information to avoid an explicit application of the choice rule to (l_1, s) and (l_2, s). The details are given below:

Lemma 8.2 *Suppose that Algorithm 7.4 is applied to $\mathcal{U}(D)$ of a solvable dominance graph D. Assume that a top-level call `HC-visit`(s) is made. Let l_1 and l_2 be two DFS children of s such that l_1 is completed before l_2 and no other DFS child of s is completed in between. Assume that the dfs_inedges of both leaves are dominance edges, and denote by r_1 and r_2 the roots of the fragments of l_1 and l_2, respectively. Then the following holds:*

1. *$D_1 = D \cup (l_1, r_2)$ is solvable. (In fact, applying Algorithm 7.4 to $\mathcal{U}(D_1)$ yields the same computation as for $\mathcal{U}(D)$.)*

2. *$D_2 = D \cup (l_2, r_1)$ is solvable iff during the call `HC-visit`(l_2) (including spawned recursive calls) no edge incident to l_1 is scanned.*

Proof. Let $U = \mathcal{U}(D)$ and $U_i = \mathcal{U}(D_i)$, $i = 1, 2$. As in the proof of the previous lemma, the basic idea is again to compare the computation of `HC-Test(`U`)` with those of `HC-Test(`U_1`)` and `HC-Test(`U_2`)`.

The first claim follows from the fact that the harmful cycle test for U_1 basically ignores the edge $d_1 = \{l_1, r_2\}$ (cf. middle of Figure 8.5). Since

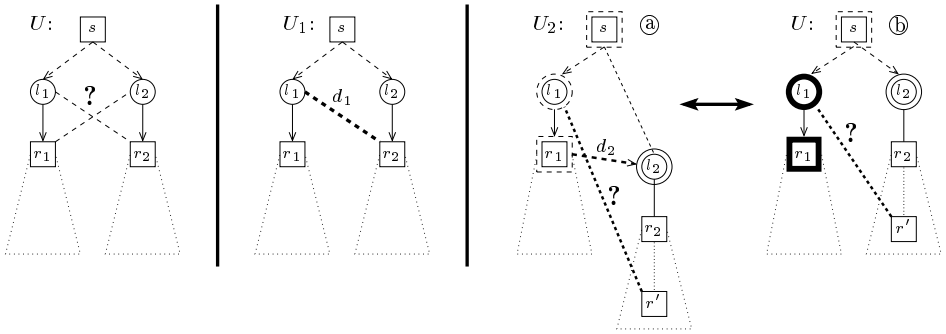


Figure 8.5: Examining two siblings l_1 and l_2 in the DFS forest generated by `HC-visit(`s`)`.

the bend $\langle \mathit{dfs\_inedge}[l_1], l_1, d_1 \rangle$ is not admissible, the edge is not scanned in the call `HC-visit(`l_1`)`. It is scanned for the first time during the call `HC-visit(`r_2`)`, and then no action is taken because l_1 is already completed at that time. Thus the computations of the harmful cycle test for U_1 and for U are identical. Since U contains no harmful cycle, the correctness of the cycle test implies that there is no harmful cycle in U_1. Hence, D_1 is solvable.

In order to prove the second claim, we examine the computations of the harmful cycle test for U and for U_2 in detail. Consider the computation for U_2, and assume w.l.o.g. that the edge $d_2 = \{l_2, r_1\}$ is the last edge that is scanned during the call `HC-visit(`r_1`)`. Since l_2 is unreached at that time, d_2 becomes the *dfs_inedge* of l_2 and a recursive call `HC-visit(`l_2`)` is spawned. We denote the state of the computation at that point in time by ⓐ, it is shown on the right-hand side of Figure 8.5.
We switch now to the harmful cycle test for U and consider the point in time when the call `HC-visit(`l_2`)` is made, the state of that computation is depicted as state ⓑ on the right-hand side of the figure.

The two states are similar, the only differences are the following: In state ⓐ, we have $\mathit{dfs\_inedge}[l_2] = d_2$ and l_1 and r_1 are still active, while in state ⓑ both l_1 and r_1 are completed and $\mathit{dfs\_inedge}[l_2] = \{s, l_2\}$. From the states ⓐ and ⓑ the computations for U_2 and U proceed in parallel ways until one the of the following occurs: l_2 gets completed or an edge incident to l_1 or r_2 is scanned.

Assume first that l_2 becomes completed. Then in the computation for U_2, r_1 and l_1 become completed, too, and the computation continues with the top-level call for s. In the computation for U, the recursive call for l_2 returns immediately to the top-level call. So after the completion of l_2, both computations are in exactly the same state again (except for $\mathit{dfs\_inedge}[l_2]$).

This means that each of them will terminate without reporting a harmful cycle.

Suppose now that an edge incident to l_1 or r_1 is scanned. At that time all nodes adjacent to r_1 except for l_1 and l_2 are already completed (because we assumed that d_2 is the last edge to be scanned in the call for r_1). This implies that an edge incident to l_1 is scanned. The harmful cycle test for U_2 will report a cycle, because l_1 is active. (In the computation for U no action will be taken, for l_1 is already completed there.)

So we have shown the following: The harmful cycle test for U_2 discovers a harmful cycle iff the test for U scans an edge incident to l_1 while l_2 is active. This proves the second claim of the lemma. □

The two lemmas allow us to design an improved enumeration algorithm. In order to be able to apply them, we make some changes to the edge scan procedure (see `E-visit` in Algorithm 8.2). When this procedure is called to scan the edges incident to a node v, the situation is as follows. We are building the DFS tree with the root node s at the top. One DFS child of s is active, we denote it by l; v is a (not necessarily proper) DFS descendant of l. By l_{prev} we denote the DFS child of s that has been completed last. (If l is the first child, then $l_{\text{prev}} = \mathit{none}$).

While constructing the DFS subtree rooted at l, we want to collect all DFS descendants l' of l that are adjacent to s (cf. Lemma 8.1), and we want to check whether an edge incident to l_{prev} is scanned (see Lemma 8.2). This explains why `E-visit` has the two additional arguments s and l_{prev}, and why a set L of leaves is returned. L contains all DFS descendants of v that are adjacent to s (cf. line 1). When an edge incident to l_{prev} is scanned, l_{prev} is also put into L (see line 5).
Moreover, the test that checks whether a scanned edge closes a harmful cycle has been omitted in `E-visit`, because we only apply it to graphs which do not have such a cycle.

We discuss now the function `EnumRoot(`D_{in}`,`s`)` (see Algorithm 8.3), which takes as input a solvable dominance graph D_{in} and a root s (with indegree at least 2) in D_{in}. It returns a set $\mathcal{A} = \{D_1, \ldots, D_h\}$ of dominance graphs with the following property: $\mathcal{S}(D_{\text{in}}) = \mathcal{S}(D_1) \dot\cup \ldots \dot\cup \mathcal{S}(D_h)$, each D_i is a solvable amplification of D_{in} and the indegree of s in D_i is one.

The algorithm works on three graphs in parallel. The first one is the undirected dominance graph U, on which the harmful cycle test for $\mathcal{U}(D_{\text{in}})$ is simulated. The second graph is the directed dominance graph D, which is used to record the information that is gathered during the simulation; D is initialized with a copy of D_{in}. D is always a solvable amplification of D_{in}. The algorithm gradually adds dominance edges to D and removes dominance

Algorithm 8.2 Modified edge scan used by Algorithm 8.3

Function: E-visit(v; s, l_{prev})
1: **if** v is adjacent to s **then** $L \leftarrow \{v\}$ **else** $L \leftarrow \emptyset$
2: $\mathit{status}[v] \leftarrow \mathit{active}$
3: **for all** edges e incident to v s.th. $\langle \mathit{dfs\_inedge}[v], v, e\rangle$ is admissible **do**
4: let w be the node adjacent to v via e
5: **if** $w = l_{\text{prev}}$ **then** $L \leftarrow L \cup \{l_{\text{prev}}\}$
6: **case 1:** $\mathit{status}[w] = \mathit{unreached}$
7: $\mathit{dfs\_inedge}[w] \leftarrow e$; $L \leftarrow L \cup$ E-visit(w)
8: **case 2:** $\mathit{status}[w] = \mathit{completed}$ and w is a root
9: Collect(e)
10: **otherwise:** do nothing
11: **end for**
12: $\mathit{status}[v] \leftarrow \mathit{completed}$
13: return L

edges incident to s until the indegree of s becomes one. We want to point out that the edge additions may introduce parallel edges and turn D into a multigraph, so we remove parallel edges before D is returned (cf. line 27). The third graph $\tilde{D}$ is also a directed dominance graph and is initialized with a copy of D_{in}. $\tilde{D}$ is only needed for the correctness proof and will be discussed later.

The algorithm builds a DFS tree rooted at s. It uses a variable l_{prev} to store the DFS child of s which has been completed last; at the beginning l_{prev} is *none*. The simulation of the harmful cycle test starts with making s active and setting $\mathit{dfs\_inedge}[s]$ to *none*. Then the edges incident to s are scanned, we only consider dominance edges, because this enables us to apply the previous lemmas. (Note that this is not a problem, for we may assume that the original cycle test scans the dominance edges first.) Whenever a dominance edge $d = \{s, l\}$ is scanned such that l is unreached, we make l a DFS child of s by setting $\mathit{dfs\_inedge}[l] = d$ and we call `E-visit` for l. This constructs the DFS subtree rooted at l and returns a list L of leaves (see above).

Every leaf l' in $L \setminus \{l, l_{\text{prev}}\}$ is incident to s and a proper DFS descendant of l. We add the dominance edge (l', r) to D, where r denotes the root of the fragment of l (cf. Lemma 8.1). This makes the edge (l', s) redundant, so we remove it from D.

Assume now that $l_{\text{prev}} \neq \mathit{none}$. If l_{prev} is contained in L, we create a copy D' of D, add the dominance edge (l, r_{prev}) to D' and remove the redundant

Algorithm 8.3 Enumeration algorithm `EnumRoot`

Function: EnumRoot(D_{in}, s)
Require: D_{in} is a solvable dominance graph, s is a root in D_{in}
1: $\mathcal{A} \leftarrow \emptyset$ // $\mathcal{A}$ will store the result
2: $U \leftarrow \mathcal{U}(D_{\text{in}})$; $D \leftarrow D_{\text{in}}$; $\tilde{D} \leftarrow D_{\text{in}}$ // $\tilde{D}$ only needed for the proof
3: $l_{\text{prev}} \leftarrow \mathit{none}$ // no previous DFS child of s
4: initialize the status of all nodes to *unreached*
5: $\mathit{dfs\_inedge}[s] \leftarrow \mathit{none}$; $\mathit{status}[s] \leftarrow \mathit{active}$
6: **for all** dominance edges $d = \{s, l\}$ in U incident to s **do**
7: **if** $\mathit{status}[l] \neq \mathit{unreached}$ **then continue** (with for loop)
8: $\mathit{dfs\_inedge}[l] \leftarrow d$; $L \leftarrow$ E-visit(l; s, l_{prev})
9: $r \leftarrow$ root of fragment of l
10: **for all** $l' \in L \setminus \{l, l_{\text{prev}}\}$ **do**
11: // apply Lemma 8.1:
12: add dominance edge (l', r) to D; remove (l', s) from D
13: **end for**
14: **if** $l_{\text{prev}} \neq \mathit{none}$ **then**
15: **if** $l_{\text{prev}} \notin L$ **then**
16: $D' \leftarrow D$; $r_{\text{prev}} \leftarrow$ root of fragment of l_{prev} in D'
17: // apply Lemma 8.2 (part 2):
18: add dominance edge (l, r_{prev}) to D'; remove (l, s) from D'
19: remove parallel edges in D'
20: $\mathcal{A} \leftarrow \mathcal{A} \cup$ EnumRoot(D', s)
21: **end if**
22: // apply Lemma 8.2 (part 1):
23: add dominance edge (l_{prev}, r) to D and $\tilde{D}$; remove (l_{prev}, s) from D
24: **end if**
25: $l_{\text{prev}} \leftarrow l$
26: **end for**
27: remove parallel edges in D
28: $\mathcal{A} \leftarrow \mathcal{A} \cup \{D\}$
29: return $\mathcal{A}$

edge (l, s) from D'. (r_{prev} is the root of the fragment of l_{prev}.) We make a recursive call `EnumRoot(`D'`,`s`)` and add the reported graphs to the result set $\mathcal{A}$. This is motivated by the second statement of Lemma 8.2.
No matter if l_{prev} is in L or not, we add the dominance edge (l_{prev}, r) to D (cf. statement 1 of Lemma 8.2), we remove (l_{prev}, s) from D due to redundancy. (The edge (l_{prev}, r) is also added to $\tilde{D}$, but no edge is removed.) Finally, we set l_{prev} equal to l and continue with the scan.

When the scan terminates, l_{prev} is the only leaf in D that is adjacent to s via a dominance edge. In fact, whenever line 6 is executed, l_{prev} is the only completed leaf that is adjacent to s by a dominance edge. This can be seen by induction: When line 6 is executed for the first time $l_{\text{prev}} = \mathit{none}$ and no node is completed. Assume now that the invariant holds. The call of `E-visit` for l declares all DFS descendants of l completed. By the edge deletions in line 12 we ensure that all proper descendants get disconnected from s. In line 23 we disconnect l_{prev}, so that l is the only completed node adjacent to s by a dominance edge. After setting l_{prev} equal to l, the invariant holds again. Since all leaves connected to s by dominance edges are completed after the scan, we conclude that the indegree of s in D is one.

It is not obvious that D is solvable. Recall that we simulate a harmful cycle test for $\mathcal{U}(D_{\text{in}})$, so we cannot apply the lemmas to D, but we can apply them to $\tilde{D}$. This follows from the fact that the edges added to $\tilde{D}$ (see line 23) are ignored by the harmful cycle test (see statement 1 of Lemma 8.2), which implies that the computation of `EnumRoot` is also a simulation of the harmful cycle test for $\mathcal{U}(\tilde{D})$. Therefore, $\tilde{D}$ is solvable. The edges in D which are not in $\tilde{D}$ have been added in line 12. By Lemma 8.1, inserting these edges into $\tilde{D}$ does not change the set of solved forms. The same holds for the deletion of redundant edges. Thus $\mathcal{S}(D) = \mathcal{S}(\tilde{D})$.

An analogous argument shows that if a recursive call `EnumRoot(`D'`,`s`)` is made, then D' is a solvable dominance graph. Moreover, $\mathcal{S}(D)$ and $\mathcal{S}(D')$ are disjoint. This follows from the fact that the lines 18 and 23 can be seen as an application of the choice rule to the edges (l_{prev}, s) and (l, s). Thus an easy induction proves that the graphs in the returned set $\mathcal{A}$ have non-empty and pairwise disjoint sets of solved forms.

We analyse the running time of the algorithm. Let m denote the number of edges of D_{in}. The time needed for simulating the harmful cycle test for U is $O(m)$. The graph D always has m edges until the parallel edges are removed in line 27, which can be done in time $O(m)$. If we do not count the time spent in lines 16 – 20, the running time of `EnumRoot` is $O(m)$. If we make a recursive call, D' can be constructed in time $O(m)$. We charge this time to the recursive call. Since every call reports one dominance graph, the

total running time of `EnumRoot(`D_{in}`,`s`)` is $O(|\mathcal{A}| \cdot m)$, where $\mathcal{A}$ is the set of amplifications returned by this call.
We summarize our results in the following lemma:

Lemma 8.3 *If Algorithm 8.3 is applied to a solvable dominance graph D and a root s in D, it generates a set $\mathcal{A} = \{D_1, \ldots, D_h\}$ of amplifications of D. The running time is $O(h \cdot m)$, where m is the number of edges of D. Each graph $D_i \in \mathcal{A}$ is solvable, has at most m edges, and the indegree of s in D_i is one. Moreover, $\mathcal{S}(D) = \mathcal{S}(D_1) \dot\cup \ldots \dot\cup \mathcal{S}(D_h)$.*

Given the function `EnumRoot` it is straightforward to design the second enumeration algorithm `Enum2` (see Algorithm 8.4). Suppose we have a solvable dominance graph D. If all roots in D have indegree one, then D is a solved form; we report D and terminate. Otherwise we choose a root s in D with indegree greater than one. We call `EnumRoot(`D`,`s`)`, which generates a set $\mathcal{A}$ of amplifications of D. We apply `Enum2` recursively to every graph in $\mathcal{A}$.

Algorithm 8.4 Enumeration algorithm Enum2

Procedure: Enum2(D)
Require: D is a solvable dominance graph
1: **if** all roots in D have indegree at most one **then**
2: report D and terminate
3: **else**
4: pick root s s.th. indeg(s) > 1, every proper descendant has indegree 1
5: $\mathcal{A} \leftarrow$ EnumRoot(D,s)
6: **for all** $D' \in \mathcal{A}$ **do** Enum2(D')
7: **end if**

We have to be careful how we select s, because `EnumRoot` may add edges. We should make sure that s is not chosen again in a recursive call of `Enum2`. Since D is solvable, it is a directed acyclic graph, and hence it makes sense to talk about ancestors and descendants. We observe that all edges added by `EnumRoot(`D`,`s`)` are incident to proper ancestors of s in D. This suggests to pick a root s with indegree greater than one such that all its proper descendants have indegree one. It is easy to determine s in time $O(m)$: We perform a depth-first search on D, until the first root s with indegree greater than one is completed. (Note that all proper descendants of s are completed before s.)

Now we analyse the running time of `Enum2(`D`)`. Let N denote the number of minimal solved forms of D, and denote by n and m the number of nodes

and edges respectively. Clearly, all the graphs that are processed in recursive calls have at most $O(m)$ edges. Moreover, our selection rule for s guarantees that the recursion depth is bounded by n.
Consider a call `Enum2(`D'`)`. It invokes `EnumRoot(`D'`,`s`)` for some root s, which generates a set $\mathcal{A}'$ in time $O(|\mathcal{A}'| \cdot m)$. $\mathcal{A}'$ gives rise to $|\mathcal{A}'|$ recursive calls of `Enum2`. By charging $O(m)$ time to each of these recursive calls, we obtain an amortized time of $O(m)$ for the call `Enum2(`D'`)`.
By each call of `Enum2` at least one minimal solved form D is reported, and no solved form is reported twice. As the recursion depth is bounded by n, there are at most $N \cdot n$ calls of `Enum2`. Therefore the total running time of `Enum2(`D`)` is $O(N \cdot nm)$.
We summarize our results in the following theorem:

Theorem 8.1 *Let D be a dominance graph with n nodes and m edges, then the solvability of D can be decided in time $O(m)$. If D is solvable, a solved form of D can be constructed in time $O(nm)$, and all N solved forms can be enumerated in time $O(N \cdot nm)$.*

Proof. The only thing that remains to be discussed is how to construct a solved form of a solvable dominance graph in time $O(nm)$. We make a small modification to the function `EnumRoot`: We delete the lines 15 – 21, i.e. we do not make recursive calls anymore. Then the function runs in time $O(m)$ and constructs only one solvable dominance graph. If we plug this modified function into Algorithm 8.4, we obtain an algorithm that reports a single solved form in time $O(nm)$. (Note that the recursion depth is still bounded by n.) □

Chapter 9

Related work and discussion

This chapter is divided in two sections. The first section discusses some related work in the field of computational linguistics. It focuses on dominance constraints and two polynomial time solvable subclasses which can be applied to many problems in computational linguistics. The second section deals with related algorithms for deciding solvability and for enumerating solved forms.

9.1 Dominance constraints and subclasses

One might say that this section describes a struggle to find a logical language for talking about trees that is rich enough to model certain problems from computational linguistics and restricted enough to solve these problems efficiently. First we introduce the language of dominance constraints which is simple and powerful but unfortunately also very hard to process. Then we discuss two proper subclasses which are useful in practice and can be processed efficiently: normal and weakly normal dominance constraints.

Dominance constraints

The presentation in the following two sections is based on parts of [ADK+03], most of the results in this section are due to Koller and Niehren.

The language of dominance constraints is a logical language that talks about trees and the ancestor-descendant relation of their nodes. In this language trees are modelled as terms composed of function symbols. The ground term $f(g(a,a))$ corresponds to the tree shown in Figure 9.1. It uses three function symbols with different arities: f with arity one, g with arity two, and a

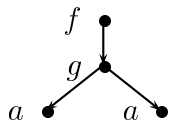


Figure 9.1: $f(g(a,a))$

with arity zero. A function symbol of arity zero is called a *constant*.

A dominance constraint ϕ is a conjunction of *dominance* and *labelling* literals of the following form

$$\phi \quad ::= \quad \phi \wedge \phi' \mid X \lhd^* Y \mid X{:}f(X_1, \ldots, X_n)$$

where X,Y and $X_1, \ldots, X_n$ are variables and f is a function symbol of arity n. The function symbols are drawn from a signature Σ, we assume that Σ contains at least one constant and one function symbol of arity at least two. We denote the arity of function symbol $f \in \Sigma$ by $\mathsf{ar}(f)$.

A solution of a constraint ϕ consists of a tree τ and a variable assignment α that maps the variables of ϕ to the nodes of τ such that all literals are satisfied.
We make this more precise. A *constructor tree* τ is a triple (V, E, L) such that the directed graph (V, E) is a rooted tree and $L : V \cup E \to \Sigma \cup \mathbb{N}$ with $L(V) \subseteq \Sigma$ (node labels) and $L(E) \subseteq \mathbb{N}$ (edge labels). The edge labels determine the left-to-right order of the outgoing edges of a node $u \in V$. For $k = 1, \ldots, \mathsf{ar}(L(u))$ there must be exactly one edge $e = (u, v) \in E$ with $L(e) = k$. The *variable assignment* is a mapping $\alpha : \mathsf{Vars}(\phi) \to V$, where $\mathsf{Vars}(\phi)$ is the set of all variables occurring in ϕ.
A tuple (τ, α) satisfies a dominance literal $X \lhd^* Y$ of ϕ iff there is a path in τ from $\alpha(X)$ to $\alpha(Y)$, i.e. $\alpha(X)$ is a (not necessarily proper) ancestor of $\alpha(Y)$. We say that (τ, α) satisfies the labelling literal $X{:}f(X_1, \ldots, X_n)$ iff $L(\alpha(X)) = f$, and $e_k = (\alpha(X), \alpha(X_k))$ is an edge in E with $L(e_k) = k$ for $k = 1, \ldots, n$.

A solution of the constraint $\phi = U{:}f(V) \wedge V{:}g(X, Y) \wedge X{:}a \wedge Y{:}a$ is the tree τ in Figure 9.1 together with the obvious variable assignment α. Since the variable assignment does not have to be surjective, any tree that contains τ as a subtree could also appear in a solution. Moreover, the variable assignment does not have to be injective, the same tree together with an extended variable assignment α' would also be a solution for the constraint $\phi \wedge W{:}f(Z)$. The variables U and W (as well as V and Z) are mapped to the same node of τ. We say that U and W *overlap* in the solution (τ, α').

The overlapping feature (in conjunction with the dominance literals) makes this language very expressive, and it is very hard to decide whether a given dominance constraint has a solution. In fact, Koller et al. [KNT01] showed that this problem is NP-complete. We do not give the proof here but we show an example that illustrates the problem. Consider the constraint $X{:}f(X_1, X_2) \wedge Y{:}f(Y_1, Y_2) \wedge Y \lhd^* X \wedge X \lhd^* Y_1$, which is depicted in Figure 9.2. Every so-

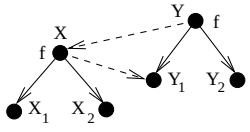


Figure 9.2: Overlap

lution to this constraint must map X to same node as either Y or Y_1, i.e. X must overlap with Y or Y_1 in any solution.

In our drawings we indicate a parent-child relation implied by a labelling literal $X : f(\ldots, Y, \ldots)$ by a solid dart from X to Y. A dominance literal $X \lhd^* Y$ is visualized by a dashed dart from X to Y. Hence, every dominance constraint ϕ corresponds in a canonical way to a directed labelled graph G_ϕ. The nodes of G_ϕ are the variables of ϕ and the edges are $\{(X,Y) \mid X{:}f(\ldots,Y,\ldots) \in \phi \text{ or } X \lhd^* Y \in \phi\}$. The edges induced by labelling literals are called *tree edges*, and the edges that stem from dominance literals are *dominance edges*. The node and edge labels of G_ϕ are given by the labelling literals (cf. the definition of a constructor tree). The connected components of the subgraph induced by tree edges are called the *fragments* of G_ϕ.

Normal dominance constraints

The NP-completeness result shed doubt on the practical usefulness of dominance constraints until Althaus et al. [ADK+01] were able to give a positive result. They extended the language of dominance constraints by allowing *inequality literals* of the form $X \neq Y$.[1] Then they identified the subclass of *normal* dominance constraints, for which solvability can be decided in polynomial time. The main property of these constraints is that the overlapping is restricted: Only a root of a tree fragment may overlap with an unlabelled leaf of another tree fragment. (Recall that this corresponds to the idea of plugging roots into holes (i.e. unlabelled leaves) from Section 6.1.) A formal definition follows:

Definition 9.1 (normal dominance constraint)
A dominance constraint ϕ with inequality *is a conjunction of dominance, labelling and inequality literals of the following form*

$$\phi \quad ::= \quad \phi \wedge \phi' \mid X \lhd^* Y \mid X{:}f(X_1, \ldots, X_n) \mid X \neq Y$$

where X, Y and $X_1, \ldots, X_n$ are variables and f is a function symbol of arity n. In a labelling literal $X{:}f(X_1, \ldots, X_n)$, the variable X occurs in parent position *and the variables $X_1, \ldots, X_n$ occur in* child position. *A variable that only occurs in parent position in ϕ is called a* root, *and a variable that only occurs in child position in ϕ is called a* hole.
A dominance constraint ϕ with inequality is called normal *if it satisfies the following conditions:*

[1] As the reader has probably guessed, (τ, α) satisfies $X \neq Y$ iff $\alpha(X) \neq \alpha(Y)$.

1. *If X and Y are variables that occur in parent position of two distinct labelling literals, then ϕ contains the literal $X \neq Y$.*

2. *Every variable appears in at least one labelling literal of ϕ.*

3. *For each variable there is at most one occurrence in parent position and at most one occurrence in child position of a labelling literal. No variable occurs twice in a single labelling literal.*

4. *If ϕ contains the literal $X \lhd^* Y$, then X is a hole and Y is a root.*

When we talk about dominance constraints in the sequel, we will always allow inequality literals. We want to make some remarks about this definition. Condition 1 (together with Condition 2) implies that the only possible overlapping is between a hole and a root. Observe that Condition 3 does not exclude fragments of solid edges in G_ϕ which contain a cycle, but if a fragment is acyclic, then it is tree shaped. Condition 4 requires that dashed edges in G_ϕ are directed from holes to roots. We want to point out that not every leaf of a solid tree fragment in G_ϕ has to be a hole. In the constraint $X{:}f(Y,Z) \wedge Z : a$ the variable Y is a hole, but Z is not, although in G_ϕ both Y and Z are leaves of the fragment with root X.

Assume that we relax the definition of dominance graphs (see page 145) a little bit and allow tree fragments of arbitrary height (instead of height one). Clearly, all the results from the previous chapters continue to hold. A dominance graph D with fragments of height different from one can be transformed in linear time into an "equivalent" graph D' with fragments of height one: If we have a fragment of height greater than one, we remove all nodes but the root and the leaves and connect the root to all the leaves by tree edges. If there is a fragment of height zero[2], i.e. just a root r, we add a new leaf l and the tree edge (r,l) to the graph. This relaxation enables us to view the graph G_ϕ of a normal dominance constraint ϕ as a dominance graph, if all fragments of G_ϕ are acyclic.
As it was the case for dominance graphs, we also define the notion solved form for a dominance constraint:

Definition 9.2 (solved form) *A dominance constraint ϕ is in* solved form *if G_ϕ is a forest. ϕ is called an* amplification of *a (normal) dominance constraint ϕ' if ϕ and ϕ' contain the same labelling and inequality literals and* $\mathrm{Reach}(G_\phi) \supseteq \mathrm{Reach}(G_{\phi'})$ *(i.e. ϕ entails all dominance literals of ϕ'). A* solved form of ϕ' *is an amplification of ϕ' which is in solved form.*

[2]E.g., the constraint $X{:}a$ corresponds to a fragment of height zero.

A solved form ϕ of ϕ' is called minimal *if there is no solved form ϕ'' of ϕ' with* $\mathrm{Reach}(G_{\phi''}) \subset \mathrm{Reach}(G_\phi)$.

We will show now that normal dominance constraints and dominance graphs are equivalent with respect to solved forms. The precise correspondence is given by the following theorem:

Lemma 9.1 *Let ϕ be a normal dominance constraint such that G_ϕ contains no cyclic fragment. Then there is a one-to-one correspondence between the minimal solved forms of ϕ and the minimal solved forms of G_ϕ (viewed as a dominance graph), which is given by the mapping $\phi' \mapsto G_{\phi'}$.*

Proof. By definition, every solved form ϕ' of ϕ is mapped to a solved form of G_ϕ. The converse is in general not true, because of Condition 4 of Definition 9.1: There may be a solved form of G_ϕ with a dominance edge emanating from a leaf which is not a hole in ϕ.

We will show that this is not the case for a minimal solved form D' of G_ϕ: If we apply one of our enumeration algorithms to G_ϕ (like Algorithm 8.1 on page 178), it will report D'. Every edge in D' which is not contained in G_ϕ has been inserted by an application of the choice rule. Since the choice rule only adds edges to leaves which already have an outgoing dominance edge (see Figure 7.2 on page 153), we conclude that every dominance edge in D' emanates from a hole in ϕ. Hence, D' corresponds to a solved form ϕ' of ϕ with $G_{\phi'} = D'$.

So far we have proven that the mapping $\phi' \mapsto G_{\phi'}$ and its inverse map a minimal solved form to a solved form. An easy argument shows that a minimal solved form of ϕ is mapped to a minimal solved form of G_ϕ and vice versa. □

This theorem allows us to apply the algorithms from the previous chapters to normal dominance constraints. But we have not proven yet that the notion *solved form* deserves its name for dominance constraints. We show below that every dominance constraint in solved form has a solution. This is not a trivial result. The construction that transforms a dominance graph in solved form into a configuration (see again Figure 6.5 on page 148) does not work for dominance constraints, because this construction may add a dominance edge to a leaf which is labelled by a constant, i.e. it is not a hole.

Lemma 9.2 *A dominance constraint ϕ in solved form has a solution.*

Proof. We only sketch the proof, the details can be found in the proof of Lemma 3.6 in [ADK+03]. By definition, G_ϕ (including its dominance edges)

is a forest. If it contains more than one tree, we add a new unlabelled root r and connect r to the root of every tree by dominance edges. So from now on we may assume that G_ϕ is a single tree.

In general, G_ϕ is not a constructor tree (see page 192), because some nodes and some edges – the holes and the dominance edges – are not labelled. Since the signature Σ contains a function symbol f of arity at least two and a constant a, we transform G_ϕ into a constructor tree τ. We repeatedly apply the transformation shown in Figure 9.3. As every variable in ϕ is a node in τ, we can choose the variable mapping α to be the identity mapping. Thus α satisfies every inequality literal. Hence, (τ, α) is a solution of ϕ. (This solution does not involve any plugging at all, this issue is addressed later.) □

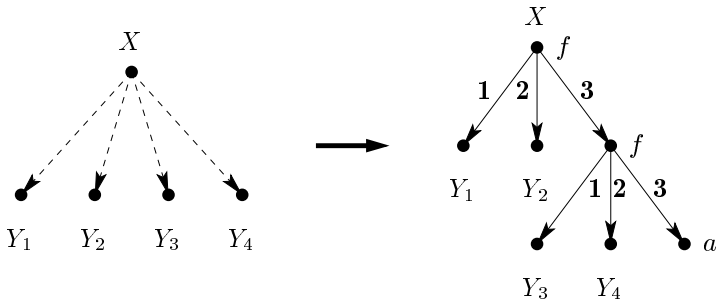


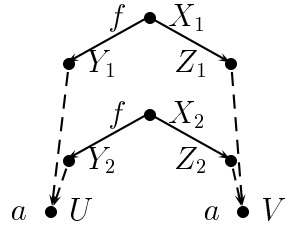
Figure 9.3: Transforming a subgraph induced by a hole and its outgoing dominance edges into a labelled subgraph. (In the example $\mathsf{ar}(f) = 3$.)

For a normal dominance constraint one can show a converse statement: The existence of a solution implies the existence of a solved form. The following lemma makes a slightly stronger statement:

Lemma 9.3 *Every solution of a normal dominance constraint ϕ satisfies some solved form of ϕ.*

Proof. See the proof of Lemma 3.7 in [ADK+03]. □

We give an example to illustrate that this lemma does not hold for arbitrary dominance constraints: $\bigwedge_{i=1}^{2}(X_i{:}f(Y_i, Z_i) \wedge Y_i \lhd^* U \wedge Z_i \lhd^* V) \wedge U{:}a \wedge V{:}a$ This constraint has a solution (the tree $f(a, a)$ with the obvious variable mapping), but it has no solved form. It is not normal because it violates Condition 1 of Definition 9.1. In order to make the constraint normal, one would



have to add inequality literals, in particular $X_1 \neq X_2$. Hence, the normal constraint would not have a solution.

Combining the previous lemmas, we see that a normal dominance constraint ϕ has a solution iff it has a solved form. By Lemma 9.1, we can apply the harmful cycle test (Algorithm 7.4) to G_ϕ in order to decide whether ϕ has a solution.

Let us examine how the size of G_ϕ depends on the size of ϕ. The number n of nodes of G_ϕ is equal to the number of variables of ϕ. G_ϕ does not depend on the inequality literals of ϕ at all. Let l denote the size of a "reasonable" encoding[3] of ϕ. Then G_ϕ has $O(l)$ edges and nodes. The next theorem follows immediately from Theorem 8.1:

Theorem 9.1 *Let ϕ be a normal dominance constraint with n variables. Denote by l the encoding length of ϕ. Deciding whether ϕ has a solution can be done in time $O(l)$. If ϕ is solvable, a solved form (and a solution) can be constructed in time $O(nl)$, and all N solved forms of ϕ can be enumerated in time $O(N \cdot nl)$.*

Constructive solutions of normal dominance constraints

Consider a dominance constraint ϕ which has a solution (τ, α). Then any constructor tree τ' which contains τ as a subgraph gives rise to a solution (τ', α). Hence, ϕ has infinitely many solutions. For many problems in computational linguistics the acceptable solutions may only contain material that is mentioned in the labelling constraints of ϕ. Koller et al. [KNT03] gave a formal definition of this property:

Definition 9.3 (constructive solution) *A solution (τ, α) of a dominance constraint ϕ is called* constructive *if the following holds: For every node n in τ there is a variable X in ϕ such that $\alpha(X) = n$ and X is not a hole.*

A constructive solution requires that every hole of ϕ overlaps with a root. Thus this notion formalizes the idea of plugging roots into holes from Section 6.1. Unfortunately, deciding whether a (normal) dominance constraint has a constructive solution is NP-complete. (This follows immediately from Theorem 10.1 in [ADK+03].) Observe that the solution which is built in the proof of Lemma 9.2 is not constructive.

[3] Each variable is encoded by an index in $[1..n]$, which fits into one machine word. The encoding size of a dominance literal is constant, and the size of a labelling literal involving a function symbol f is linear in the arity of f. Inequality literals implied by normality are not encoded at all.

But Koller et al. [KNT03] were able to identify a subclass of constraints called *leaf-labelled, chain-connected normal dominance constraints* for which the existence of a constructive solution can be decided in linear time (with our harmful cycle test). A normal dominance constraint ϕ is *leaf-labelled* if every variable occurs on the left-hand side of a labelling or a dominance literal. This means that the leaf of every fragment of G_ϕ is either labelled by a constant or the source of a dominance edge.
ϕ is called *chain-connected* if the fragments $F_1, \ldots, F_k$ of G_ϕ can be partitioned into two disjoint sets $\mathcal{O}$ and $\mathcal{U}$ such that the following holds (cf. Figure 9.4):

1. $\mathcal{O}$ is not empty.
2. For all i let r_i be the root of F_i. For $i = 1, \ldots, k-1$ either
   - $F_i \in \mathcal{O}$ and $F_{i+1} \in \mathcal{U}$, and there is a hole $X_{i,r}$ in F_i such that $(X_{i,r}, r_{i+1}) \in \mathrm{Reach}(G_\phi)$; or
   - $F_i \in \mathcal{U}$ and $F_{i+1} \in \mathcal{O}$, and there is a hole $X_{i+1,l}$ in F_{i+1} such that $(X_{i+1,l}, r_i) \in \mathrm{Reach}(G_\phi)$.
3. For all $i \in [2..k-1]$ with $F_i \in \mathcal{O}$ the holes $X_{i,l}$ and $X_{i,r}$ are different.

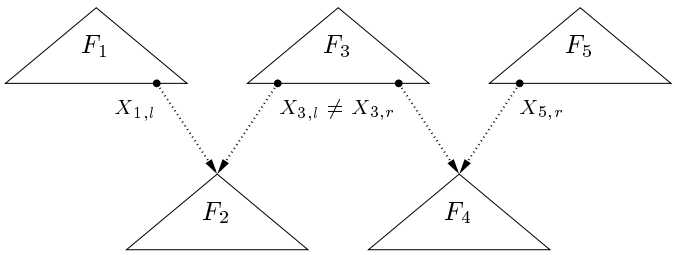


Figure 9.4: A schematic picture of a chain. A dotted dart represents a path in G_ϕ (possibly a single dominance edge).

Koller et al. show the following theorem (see Theorem 13 in [KNT03]) for the above defined subclass of normal dominance constraints:

Theorem 9.2 *Every solved form ϕ' of a leaf-labelled, chain-connected normal dominance constraint ϕ has a constructive solution.*

This implies that the existence of a constructive solution of ϕ can be checked in linear time. Koller et al. also show that $G_{\phi'}$ is always a configuration of G_ϕ. Hence, a constructive solution of ϕ can be built from ϕ' in linear

time. So if ϕ is solvable, all N constructive solutions can be enumerated in time $O(N \cdot nl)$ where n is the number of variables and l is the encoding length of ϕ.

Koller et al. conjecture that all linguistically useful constraints fall in the subclass of leaf-labelled, chain-connected normal dominance constraints. In order to justify their conjecture, they define a nontrivial grammar for a fragment of English and show that it only generates dominance constraints that belong to the class above.
Moreover, they consider an underspecification formalism called *Hole Semantics* ([Bos96] and [Bos02]). They describe a back-and-forth translation between this formalism and normal dominance constraints. Due to this translation, the algorithms from the Chapters 7 and 8 can be used to speed up the processing of Hole Semantics for practically useful instances.

Weakly normal dominance constraints

Bodirsky et al. [BDMN04] introduced a subclass of dominance constraints which is called *weakly normal dominance constraints*. This class is a proper superclass of the class of normal dominance constraints. The main difference is that Condition 4 of Definition 9.1 is relaxed for a weakly normal dominance constraint ϕ: If ϕ contains a dominance literal $X \lhd^* Y$, then X is a hole **or a root** and Y is a root.

Weakly normal dominance constraints correspond to *weakly normal dominance graphs*, which are dominance graphs that allow root-to-root dominance edges. Bodirsky et al. show how to enumerate all N minimal solved forms of a solvable weakly normal dominance graph D in time $O(N \cdot nm)$, where n is the number of nodes and m is the number of edges of D. This matches the asymptotic running time of the best enumeration algorithm presented in this thesis. However, with respect to deciding solvability their best result is $O(nm)$. This is a factor of n slower than our solvability test for normal dominance constraints (see Algorithm 7.4).

This gap gives rise to the question whether the approach from Chapter 7 can be generalized to weakly normal dominance graphs. The answer is probably negative as the example D on the left-hand side of Figure 9.5 demonstrates. It is easy to see that D is unsolvable. $\mathcal{U}(D)$ contains three simple cycles: $C_1 = [a, b, f, d, g, c, a]$, $C_2 = [a, b, f, d, a]$ and $C_3 = [a, c, g, d, a]$. We will show that none of these cycles alone proves unsolvability, i.e. none of them corresponds to an unsolvable subgraph of D. The graph $D_1 = D \setminus (a, d)$ is solvable (cf. middle of the figure) and $\mathcal{U}(D_1)$ contains C_1. An analogous observation can be made for $D_2 = D \setminus (c, g)$ and C_2 (see right-hand side of

the figure), as well as for $D_3 = D \setminus (b, f)$ and C_3 (symmetrical to D_2 and C_2).

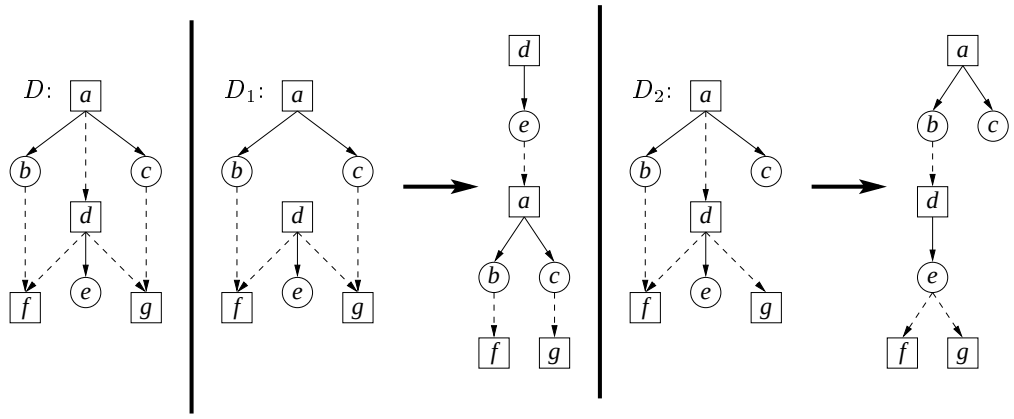


Figure 9.5: An unsolvable weakly normal dominance graph D without an unsolvable simple cycle.

Niehren and Thater [NT03] show that enlarging the subclass of efficiently solvable dominance constraints from normal to weakly normal dominance constraints is relevant for computational linguistics. They describe a sublanguage of Minimal Recursion Semantics (see [CFS97]), which is called MRS-nets. By defining a back-and-forth translation between MRS-nets and normal dominance nets (a subclass of weakly normal dominance constraints), they show that the algorithms in [BDMN04] can be applied to MRS-nets.

9.2 Related algorithms

In this section we give an overview about the algorithms that have been developed for solving dominance constraints. To the best of our knowledge the first algorithms for solving dominance constraints are based on constraint programming (see [DG99] and [DN99]). The implementations are based on set constraints [MM97]. The value of a set variable S is a set (usually of integers), hence the domain of S is a set of sets. The encoding of a dominance constraint as a constraint program with set constraints is similar in all approaches. We follow the presentation of Koller and Niehren [KN02].

Consider a dominance constraint ϕ with n variables $X_1, \ldots, X_n$. The following descriptions become easier if we assume that ϕ has a solution (τ, α), which we do not know yet of course. For any pair $\{X_i, X_j\}$ of variables we can distinguish the relative position of the nodes $n_i = \alpha(X_i)$ and $n_j = \alpha(X_j)$

in τ: $n_i = n_j$, or n_i is a proper ancestor of n_j (denoted as $n_i \lhd^+ n_j$), or n_i is a proper descendant of n_j, or neither of the three previous cases holds, which we denote by $n_i \perp n_j$. With respect to a variable X_i we can partition the variable indices in four disjoint sets:

- $\mathsf{Eq}_i := \{j \mid \alpha(X_i) = \alpha(X_j)\}$
- $\mathsf{Up}_i := \{j \mid \alpha(X_i) \lhd^+ \alpha(X_j)\}$
- $\mathsf{Down}_i := \{j \mid \alpha(X_j) \lhd^+ \alpha(X_i)\}$
- $\mathsf{Disj}_i := \{j \mid \alpha(X_i) \perp \alpha(X_j)\}$

As we do not know these sets, we introduce finite set variables eq_i, up_i, $down_i$ and $disj_i$, which allow us to reason about these sets even if they are not determined. For convenience we add two auxiliary set variables $equp_i$ and $eqdown_i$, which will be constrained to be the union of eq_i with up_i and $down_i$, respectively. Moreover, we introduce for each variable X_i a finite domain integer variable $label_i$ which represents the – currently unknown – label of $\alpha(X_i)$ in τ. In addition, we have a tuple variable $children_i$ for the children of $\alpha(X_i)$. The variable X_i itself is modelled as a tuple variable x_i. For each variable X_i of ϕ we post the following constraints to the solver:

$$\begin{array}{ll} & x_i = [eq_i, up_i, down_i, disj_i, children_i, label_i] \\ \wedge & i \in eq_i \ \wedge \ eq_i \,\dot\cup\, up_i \,\dot\cup\, down_i \,\dot\cup\, disj_i = [1..n] \\ \wedge & equp_i = eq_i \,\dot\cup\, up_i \ \wedge \ eqdown_i = eq_i \,\dot\cup\, down_i \end{array}$$

Now we show how the literals of ϕ are modelled in the constraint program:

$$\begin{array}{lcl} [\![\, X_i \lhd^* X_j \,]\!] & \equiv & equp_i \subseteq equp_j \ \wedge \ eqdown_i \supseteq eqdown_j \\ & \wedge & disj_i \subseteq disj_j \\ \\ [\![\, X_i{:}f(X_{j_1}, \ldots, X_{j_n}) \,]\!] & \equiv & label_i = \mathsf{id}(f) \ \wedge \ children_i = [x_{j_1}, \ldots, x_{j_n}] \\ & \wedge & down_i = eqdown_{j_1} \,\dot\cup\, \ldots \,\dot\cup\, eqdown_{j_n} \\ & \wedge & up_{j_1} = equp_i \ \wedge \ \ldots \ \wedge \ up_{j_n} = equp_i \\ \\ [\![\, X_i \neq X_j \,]\!] & \equiv & eq_i \cap eq_j = \emptyset \end{array}$$

These constraints alone do not guarantee that every solution of the constraint program encodes a forest. Consider for example the dominance constraint $X_1 \neq X_2$; there is a solution where $eq_1 = up_2 = \{1\}$, $eq_2 = up_1 = \{2\}$. So for any pair of variables $\{X_i, X_j\}$ of ϕ we introduce an integer variable R_{ij} and impose the following constraint:

$$\begin{array}{l} R_{ij} \in [1..4] \ \wedge \ \mathsf{xor}(R_{ij} = 1 \wedge x_i = x_j, \ R_{ij} = 2 \wedge [\![\, X_i \lhd^+ X_j \,]\!], \\ \qquad\qquad R_{ij} = 3 \wedge [\![\, X_j \lhd^+ X_i \,]\!], \ R_{ij} = 4 \wedge [\![\, X_i \perp X_j \,]\!]) \end{array}$$

where

$$\begin{array}{rcl} [\![X_i \lhd^+ X_j]\!] & \equiv & [\![X_i \lhd^* X_j]\!] \wedge [\![X_i \neq X_j]\!] \\ [\![X_i \perp X_j]\!] & \equiv & eqdown_i \subseteq disj_j \wedge eqdown_j \subseteq disj_i \end{array}$$

These constraints enforce that any solution of the constraint program encodes a forest. The variable R_{ij} is useful for enumerating all solutions of the constraint program during the search process.

The nice property of the solvers based on set constraints is that they can handle arbitrary dominance constraints. Moreover, the practical applications in computational linguistics usually involve not only dominance constraints. Here the flexibility of the constraint programming approach becomes apparent: Other constraints can be integrated seamlessly into the existing program. But the approach that uses set constraints has some major drawbacks. Since even deciding solvability for dominance constraints is NP-complete, there are no non-trivial runtime guarantees. And applied to instances from computational linguistics these solvers are too slow to be really practical; observe that the program above uses $\Theta(n^2)$ disjunctive propagators (xor) for a constraint with n variables.

In fact, with respect to the class of normal dominance constraints the solvers based on set constraints were outperformed by the first polynomial time solver by Althaus et al. [ADK+01]. We want to point that these algorithms have been integrated in the constraint solver of Oz [Smo95]. This makes sure that the advantages of the constraint programming approach are not lost.

The enumeration algorithm for minimal solved forms by Althaus et al. is similar to the procedure `Enum1` in this thesis (see Algorithm 8.1). The core of their algorithm is also a harmful cycle[4] test. Their test is different from Algorithm 7.4 and it is less efficient.

We sketch the approach in [ADK+01] and [ADK+03]. The idea is to transform the problem of finding a harmful cycle in an undirected dominance graph U to a matching problem in an auxiliary graph A. For every edge $e = \{u, v\}$ in U there are two nodes n_{eu} and n_{ev} in A. The edge set of A is partitioned in two sets M and K. M contains an edge $\{n_{eu}, n_{ev}\}$ for every edge $e = \{u, v\}$ of U. For every admissible bend $\langle e, v, f\rangle$ in U (cf. Definition 7.2) we have an edge $\{n_{ev}, n_{fv}\}$ in K. It is easy to see that M is a perfect matching in A. Althaus et al. show that A contains a harmful cycle iff A contains a perfect matching M' that differs from M.
The construction of the auxiliary graph is illustrated by an example in Figure 9.6. Each edge of U gives rise to two nodes in A (indicated as black

[4] In [ADK+01] these cycles are called *hypernormal cycles*.

bullets) and to an edge in M (indicated by a thick solid line). Every admissible bend in U corresponds to an edge in K (indicated by a thick dashed line). Only one bend in U is forbidden, it is marked by "!" in the middle of the figure. The harmful cycle $C = [a, b, d, c]$ in U corresponds to an alternating cycle C' in A. Hence, it gives rise to the perfect matching $M' = M \oplus C'$ in A, which is depicted on the right-hand side of the figure. Observe that the "harmless" cycle $[d, c, f, g, h, e]$ in U does not correspond to an alternating cycle in A.

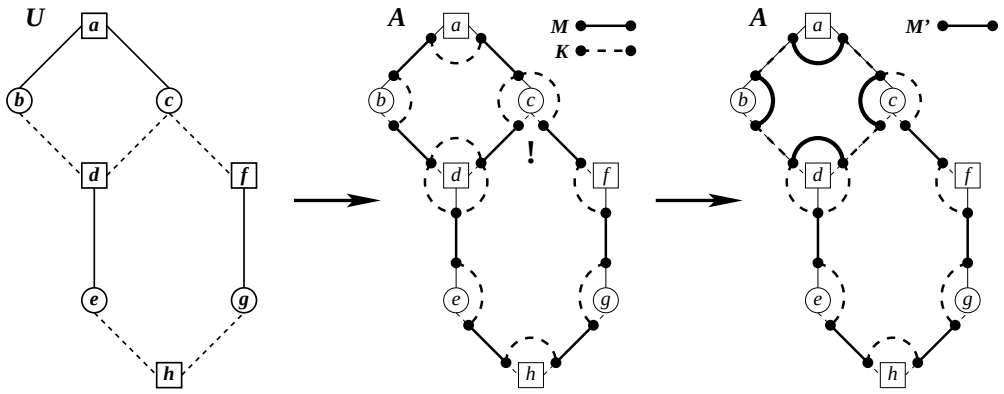


Figure 9.6: An example illustrating the construction of the auxiliary graph.

In order to test whether A contains a perfect matching $M' \neq M$, one can use an algorithm by Gabow et al. [GKT01] which has a linear running time in the size of A. Unfortunately, there are examples where A is much larger than U. Let n be the number of nodes and m be the number of edges of U. Then A has $n' = m$ nodes and it may have $m' = \Theta(nm)$ edges. Thus the worst case running time of the solvability test is $\Theta(nm)$. Using this test, Althaus et al. obtain an enumeration algorithm that can enumerate all minimal solved forms of a dominance graph in time $O(n^3 m)$ per solved form.

The matching uniqueness test by Gabow et al. is based on Edmonds' blossom-shrinking idea [Edm65]. A reader who is familiar with this technique may have observed some similarities with the procedure `Collect` (see Algorithm 7.5) that is used by the harmful cycle test in this thesis. Algorithm 7.4 operates directly on the dominance graph and its worst case running time is $O(m)$, which is a factor of n better.

Finally, we briefly discuss the algorithms by Bodirsky et al. [BDMN04], which can be applied to weakly normal dominance graphs. Their work is based on the notion of *freeness*. A node u in a dominance graph D is called *free* if there is a solved form of D where u has indegree zero. So if u is free,

one can construct a solved form D' of D where u is the root of a tree in the forest D'. Clearly, if D is solvable, it must contain at least one free node. Bodirsky et al. show that a free node u satisfies the following two properties:

1. The indegree of u in D is zero.
2. In $\mathcal{U}(D)$ there are no two distinct tree edges e and f which are incident to u and belong to same biconnected component[5] of $\mathcal{U}(D)$.

Let D be a dominance graph which consists of a single connected component. Suppose we want to find a solved form of D. The algorithm of Bodirsky et al. first chooses some candidate u for a free node, i.e. a node that satisfies the two properties described above. (If no such candidate exists, the algorithm fails.) Then u is removed from D, and for each connected component of $D \setminus u$ a solved form is computed recursively. Finally, these solved forms and u are assembled to a solved form of D.
It can be shown that if the algorithm fails (for some choice of candidates), then D is not solvable. Moreover, the algorithm can be easily modified to enumerate all minimal solved forms of D: If there are several candidates for a free node, one has to consider all of them instead of choosing only one of them.

The running time of the solvability test by Bodirsky et al. is $O(nm)$, which is an order of magnitude slower than Algorithm 7.4. Concerning the enumeration of solved forms, they match the asymptotic running time of the procedure `Enum2` (see Algorithm 8.4), which is $O(N \cdot nm)$ to enumerate all N minimal solved forms. Of course, one has to take into account that their algorithms can be applied to a larger class of problems.

To summarize the second part of this thesis, one can say that we have developed efficient algorithms which can be applied to practical problems in computational linguistics. They contributed to proving that dominance constraints are not only a theoretical tool for modelling problems, but also give rise to practical implementations to solve these problems. For the class of normal dominance graphs, our algorithms can compete with the best known enumeration algorithms and they outperform all other known solvability tests by at least a factor of n.

[5] A graph G is *biconnected* if it cannot be disconnected by the removal of a single node. A *biconnected component* of $\mathcal{U}(D)$ is a maximal biconnected subgraph.

Summary

This thesis is divided in two parts: The first part discusses propagation algorithms for some constraints. The second part deals with dominance graphs, these graphs can be used to represent and solve some tree processing problems arising in computational linguistics.

A constraint satisfaction problem (CSP) consists of a finite set of variables $X_1, \ldots, X_n$ with associated domains $D_1, \ldots, D_n$ and a finite set of constraints on these variables. The task is to find a solution, which means a variable assignment that maps every variable X_i to a value in D_i and satisfies all constraints. The set of all possible variable assignments (including assignments that are not solutions) is called the search space.

A very successful approach for solving CSPs interleaves constraint propagation, which prunes parts of the search space that do not contain a solution, and search, which explores the remaining parts. Thus the overall performance of this approach depends heavily on the complexity of the propagation algorithms and the amount of pruning that they achieve.
We present propagation algorithms for the following constraints:

- *Sortedness* and *Alldiff*:
  The constraint $\mathit{Sortedness}(X_1, \ldots, X_n; Y_1, \ldots, Y_n)$ holds iff sorting the sequence $[X_1, \ldots, X_n]$ (in non-descending order) yields the sequence $[Y_1, \ldots, Y_n]$. The constraint $\mathit{Alldiff}(X_1, \ldots, X_n)$ holds iff $X_1, \ldots, X_n$ are pairwise different.
  We assume that all variable domains are intervals and for each of the two constraints we develop a bound-consistency algorithm that runs in time $O(n)$ plus the time needed to sort the interval endpoints.

- *WeightedPartialAlldiff* (abbreviated as *WPA*):
  The constraint $\mathit{WPA}(X_1, \ldots, X_n; \mathit{undef}; T; W)$ is a generalization of *Alldiff*. Not all assignment variables $X_1, \ldots, X_n$ have to take different values; the special value *undef* may be assigned to several variables. Only those assignment variables which are not equal to *undef* have to take pairwise distinct values. Moreover, with every value different from

undef that occurs in one of the domains $Dom(X_1)$, ..., $Dom(X_n)$ we associate a weight that is determined by the value-weight table T. The constraint states that $\sum_{i=1}^{n} weight(X_i) = W$, where W is the weight variable.
We show that the problem of deciding whether this constraint has a solution is NP-complete. But we identify some application scenarios where we can achieve arc-consistency for the assignment variables in time $O(nm)$. Here m is the sum of the cardinalities of the domains of the assignment variables.

- *NonOverlapping*:
  This constraint states that two objects in the two-dimensional plane $\mathbb{R}^2$ should not overlap. The shape of each object is determined by a convex polygon *Shp* and the position of each object is specified by two variables X and Y. The actual object is obtained by applying the translation vector (X, Y) to *Shp*.
  We suppose that the variable domains are (closed) intervals of real numbers, and we give a bound-consistency algorithm. Under the assumption that comparisons and basic arithmetic operations can be performed in constant time, the running time of our algorithm is linear in the size of the input polygons.

The second part deals with a tree processing problem from computational linguistics. The problem is given to us in the form of a dominance graph. Informally, such a graph contains a collection of tree fragments which have to be assembled into a tree T such that some given constraints are satisfied. These constraints have the form "node u should *dominate* node v", which means that u should be an ancestor of v in T.

We describe a criterion which allows us to decide efficiently whether a given dominance graph D is solvable: We show that solvability is equivalent to the absence of certain cycles in D (so-called harmful cycles). Based on this criterion we develop an algorithm for deciding solvability of D that runs in time $O(n + m)$, where n is the number of nodes and m is the number of edges of D. Finally, we present an algorithm that can enumerate all N (minimal) solved forms of D in time $O(m + N \cdot nm)$.

Zusammenfassung

Diese Arbeit besteht aus zwei Teilen. Im ersten Teil behandeln wir Propagierungsalgorithmen für einige Constraints. Der zweite Teil beschäftigt sich mit Dominanzgraphen; diese Graphen dienen der Beschreibung von Baumverarbeitungsproblemen aus dem Bereich Computer-Linguistik.

Ein Constraint-Problem ist gegeben durch eine endliche Menge von Variablen $X_1, \ldots, X_n$, die zugehörigen Wertebereiche $D_1, \ldots, D_n$ und eine endliche Menge von Constraints (Bedingungen, Anforderungen) für diese Variablen. Eine Lösung des Problems ist eine Variablenbelegung, die jeder Variablen X_i einen Wert in ihrem Wertebereich D_i zuweist, so daß alle Constraints erfüllt sind. Der Suchraum eines Constraint-Problems ist die Menge aller Variablenbelegungen (auch solche, die keine Lösung sind, gehören dazu).

Ein erfolgreicher Ansatz zur Lösung von solchen Problemen besteht darin, abwechselnd Constraint-Propagierung und Suche einzusetzen. Dabei dient die Propagierung dazu, Teile des Suchraumes zu eliminieren, die keine Lösung enthalten, so daß die Suche nur noch einen kleinen Teil des ursprünglichen Raumes explorieren muß. Somit hängt der Erfolg dieses Ansatzes stark von der Komplexität der Propagierungsalgorithmen ab und davon, wieviel des Suchraumes sie eliminieren können.
Wir beschreiben Propagierungsalgorithmen für die folgenden Constraints:

- *Sortedness* und *Alldiff*:
  Der Constraint *Sortedness*$(X_1, \ldots, X_n; Y_1, \ldots, Y_n)$ ist genau dann erfüllt, wenn man die Sequenz $[Y_1, \ldots, Y_n]$ durch (aufsteigendes) Sortieren der Sequenz $[X_1, \ldots, X_n]$ erhält. Der Constraint *Alldiff*$(X_1, \ldots, X_n)$ ist genau dann erfüllt, wenn den Variablen $X_1, \ldots, X_n$ paarweise verschiedene Werte zugewiesen werden.
  Die Wertebereiche, auf denen unsere Propagierer arbeiten sind Intervalle. Unsere Algorithmen erreichen "bound-consistency", d.h. die Endpunkte der Ausgabeintervalle sind konsistent. Die Laufzeit ist $O(n)$ plus die Zeit, die zum Sortieren der Endpunkte der Eingabeintervalle benötigt wird.

- *WeightedPartialAlldiff* (abgekürzt *WPA*):
  Der Constraint $WPA(X_1, \ldots, X_n; undef; T; W)$ ist eine Verallgemeinerung von *Alldiff*. Die Werte der Variablen $X_1, \ldots, X_n$ müssen nicht alle unterschiedlich sein; der spezielle Wert *undef* kann mehreren Variablen zugewiesen werden. Nur die von *undef* verschiedenen Variablen müssen paarweise verschiedene Werte annehmen. Jedem Wert v (außer *undef*) ist durch die Gewichtstabelle T ein Gewicht $weight(v)$ zugeordnet. Der Constraint sagt aus, daß $\sum_{i=1}^{n} weight(X_i) = W$ sein muß.
  Wir zeigen, daß das Erfüllbarkeitsproblem für *WPA*-constraints NP-vollständig ist. Für einige Anwendungsszenarien entwickeln wir Algorithmen, die "arc-consistency" erreichen und die Laufzeit $O(nm)$ haben, wobei m die Summe der Kardinalitäten der Wertebereiche von $X_1, \ldots, X_n$ bezeichnet. ("Arc-consistency" bedeutet, daß jeder Wert in einem Ausgabe-Wertebereich Teil einer Lösung des Constraints ist.)

- *NonOverlapping*:
  Dieser Constraint legt fest, daß sich 2 zwei-dimensionale Objekte nicht überlappen. Das Aussehen eines Objektes wird jeweils durch eine konvexes Polygon *Shp* festgelegt, und seine Position wird durch zwei Variablen X und Y beschrieben. Das eigentliche Objekt erhält man, indem man *Shp* um den Translationsvektor (X, Y) verschiebt.
  Wir gehen davon aus, daß die Wertebereiche der Variablen (abgeschlossene) Intervalle von reellen Zahlen sind. Unser Algorithmus erreicht "bound-consistency" (vgl. oben). Unter Annahme das Vergleiche und die grundlegenden arithmetischen Operationen auf reellen Zahlen in konstanter Zeit ausgeführt werden können ist die Laufzeit linear in der Größe der Polygone in der Eingabe.

Im zweiten Teil beschäftigen wir uns mit einem Baumverarbeitungsproblem aus der Computerlinguistik. Das Problem wird durch einen Dominanzgraphen beschrieben. Solch ein Graph enthält eine Menge von Baumfragmenten, die zu einem Baum T zusammengesetzt werden sollen, so daß die Vorfahrrelation von T gewisse Bedingungen erfüllt. Diese haben die Form "der Knoten u soll den Knoten v *dominieren*", d.h. u soll ein Vorfahr von v in T sein.

Wir beschreiben ein Kriterium, das die Grundlage für einen effizienten Lösbarkeitstest bildet: Ein Dominanzgraph D ist genau dann, wenn er keinen "bösen Zyklus" (*harmful cycle*) enthält. Unser Lösbarkeitstest hat eine Laufzeit von $O(n + m)$, wobei n die Anzahl der Knoten und m die Anzahl der Kanten von D ist. Darüber hinaus entwickeln wir einen Algorithmus, der alle N (minimalen) gelösten Formen von D in Zeit $O(m+N \cdot nm)$ aufzählen kann.

Bibliography

[AC92] H. Alshawi and R. Crouch. Monotonic semantic interpretation. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 32–39, 1992.

[ADK+01] E. Althaus, D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. An Efficient Algorithm for the Configuration Problem of Dominance Graphs. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA 01)*, pages 815–824, 2001.

[ADK+03] E. Althaus, D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. An Efficient Algorithm for the Configuration Problem of Dominance Graphs. *Journal of Algorithms*, 48:194–219, 2003.

[AGU72] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AMO93] R.K. Ahuja, T.L. Magnati, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.

[Apt03] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[BC94] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.

[BCT02] N. Beldiceanu, M. Carlsson, and S. Thiel. Cost-Filtering Algorithms for the two Sides of the Sum of Weights of Distinct

Values Constraint. SICS Technical Report T2002:14, SICS, 2002.

[BDMN04] M. Bodirsky, D. Duchier, S. Miele, and J. Niehren. An Efficient Algorithm for Weakly Normal Dominance Constraints. Accepted for the ACM-SIAM Symposium on Discrete Algorithms (SODA 04), 2004.

[Bel00] N. Beldiceanu. Global Constraints as Graph Properties on Structured Network of Elementary Constraints of the Same Type. SICS Technical Report T2000/01, SICS, 2000.

[BGC00] N. Bleuzen-Guernalec and A. Colmerauer. Optimal narrowing of a block of sortings in optimal time. *Constraints*, 5(1–2):85–118, 2000.

[BGT01] N. Beldiceanu, Q. Guo, and S. Thiel. Non-overlapping Constraints between Convex Polytopes. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *LNCS*, pages 392–407, 2001.

[Bos96] J. Bos. Predicate Logic Unplugged. In *Proceedings of the 10th Amsterdam Colloquium*, pages 133–143, 1996.

[Bos02] J. Bos. *Underspecification and resolution in discourse semantics.* PhD thesis, Universität des Saarlandes, Saarbrücken, 2002.

[BRVS95] R. Backofen, J. Rogers, and K. Vijay-Shanker. A First-order Axiomatization of the Theory of Finite Trees. *Journal of Logic, Language, and Information*, 4:5–39, 1995.

[CF94] A. Chamard and A. Fischler. MADE - A Workshop Scheduler System written in CHIP. In *Proceedings of the 2nd International Conference on the Practical Applications of Prolog (PAP-94)*, pages 123–136, April 1994.

[CFS97] A. Copestake, D. Flickinger, and I. Sag. Minimal Recursion Semantics. An Introduction. Manuscript, 1997.

[Cha92] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM Journal on Computing*, 21(4):671–696, 1992.

[CL97] Y. Caseau and F. Laburthe. Solving Various Weighted Matching Problems with Constraints. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330 of *LNCS*, pages 17–31, 1997.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for Dense Graphs and Networks on the Random Access Computer. *Algorithmica*, 15(5):521–549, 1996.

[dBvKOS00] M. de Berg, M. van Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Theory and Applications*. Springer, 2000.

[DG99] D. Duchier and C. Gardent. A constraint-based treatment of descriptions. In *Proceedings of IWCS-3*, Tilburg, 1999.

[DN99] D. Duchier and J. Niehren. Solving dominance constraints with finite set constraint programming. Technical report, Universität des Saarlandes, Saarbrücken, 1999.

[DT99] D. Duchier and S. Thater. Parsing with Tree Descriptions: a constraint-based approach. In *Proceedings of the 6th International Workshop on Natural Language Understanding and Logic Programming (NLULP'99)*, pages 17–32, 1999.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[EKN01] M. Egg, A. Koller, and J. Niehren. The Constraint Language for Lambda Structures. *Journal of Logic, Language, and Information*, 10:457–485, 2001.

[EL03] F. Eisenbrand and S. Laue. A faster algorithm for two variable integer programming. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC 03)*, LNCS. Springer, 2003. To appear.

[ENRX98] M. Egg, J. Niehren, P. Ruhrberg, and F. Xu. Constraints over Lambda-Structures in Semantic Underspecification. In *Proceedings of the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics (COLING/ACL'98)*, pages 353–359, 1998.

[FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with O(1) Worst Case Access Time. *Journal of the ACM*, 31(3):538–544, 1984.

[FvDFH90] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer graphics : principles and practice.* Systems programming series. Addison-Wesley, 2nd edition, 1990.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Mathematical sciences series. Freeman, 1979.

[GK79] A. Goralcikova and V. Koubek. A reduct-and-closure algorithm for graphs. In *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 301–307, 1979.

[GKT01] H.N. Gabow, H. Kaplan, and R.E. Tarjan. Unique Maximum Matching Algorithms. *Journal of Algorithms*, 40:159–183, 2001.

[Glo67] F. Glover. Maximum Matching in a Convex Bipartite Graph. *Naval Research Logistics Quarterly*, 14:313–316, 1967.

[GO97] J.E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press, 1st edition, 1997.

[GS87] L.J. Guibas and R. Seidel. Computing Convolutions by Reciprocal Search. *Discrete Computational Geometry*, 2:175–193, 1987.

[GSC87] L.J. Guibas, J. Stolfi, and K.L. Clarkson. Solving related two- and three-dimensional linear programming problems in logarithmic time. *Theoretical Computer Science*, 49:81–84, 1987.

[GT85] H.N. Gabow and R.E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[GW98] C. Gardent and B. Webber. Describing discourse semantics. In *Proceedings of the 4th TAG+ Workshop*, Philadelphia, 1998.

[Hal35] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[Har99] W. Harvey. Computing two-dimensional integer hulls. *SIAM Journal on Computing*, 28:2285–2299, 1999.

[HMMN84] S. Hertel, M. Mäntylä, K. Mehlhorn, and J. Nievergelt. Space Sweep Solves Intersection of Convex Polyhedra. *Acta Informatica*, 21:501–519, 1984.

[Hob83] J. Hobbs. An Improper Treatment of Quantification in Ordinary English. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 57–63, 1983.

[Hol95] C. Holzbaur. OFAI clp(q,r) Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[KN02] A. Koller and J. Niehren. Constraint Programming in Computational Linguistics. In *Words, Proofs, and Dialog*, volume 141 of *CSLI Lecture Notes*, pages 95–122. CSLI Press, 2002.

[KNT01] A. Koller, J. Niehren, and R. Treinen. Dominance Constraints: Algorithms and Complexity. In M. Moortgat, editor, *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics (1998)*, volume 2014 of *LNAI*, 2001.

[KNT03] A. Koller, J. Niehren, and S. Thater. Bridging the Gap Between Underspecification Formalisms: Hole Semantics as Dominance Constraints. In *Proceedings of the 11th EACL*, Budapest, 2003.

[KT03] I. Katriel and S. Thiel. Fast Bound-Consistency for the Global Cardinality Constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 437–451, 2003.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, 1976.

[Lec96] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *Constraint-96, Second Workshop on Constraint-based Reasoning*, pages 19–28, Key West, Florida, 1996.

[LG84] A. Lahrichi and M. Gondran. Théorie des parties obligatoires et découpes à deux dimensions. Research report HI/4762-02, Électricité de France (EDF), 1984.

[LOQTvB03] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint. In *Proceedings of IJCAI-03*, 2003.

[Meh84] K. Mehlhorn. *Data structures and algorithms. Volume 1 : Sorting and searching*, volume 1 of *EATCS monographs on theoretical computer science*. Springer, 1984.

[MHF83] M.P. Marcus, D. Hindle, and M.M. Fleck. D-theory: Talking about Talking about Trees. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 129–136, 1983.

[MM97] T. Müller and M. Müller. Finite set constraints in Oz. In *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 1997.

[MN99] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[MT00] K. Mehlhorn and S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 306–319, 2000.

[Mus95] R.A. Muskens. Order-independence and underspecification. In J. Groenendijk, editor, *Ellipsis, Underspecification, Events and More in Dynamic Semantics*. DYANA Deliverable R.2.2.C, 1995.

[NT03] J. Niehren and S. Thater. Bridging the Gap Between Underspecification Formalisms: Minimal Recursion Semantics as Dominance Constraints. In *41st Meeting of the Association of Computational Linguistics*, pages 367–374, 2003.

[OSvE95] W.J. Older, G.M. Swinkels, and M.H. van Emden. Getting to the real problem: experience with BNR Prolog in OR. In *Proceedings of the 3rd International Conference on the Practical Applications of Prolog (PAP-95)*, pages 465–478, April 1995.

[Per00] G. Perrier. From Intuitionistic Proof Nets to Interaction Grammars. In *Proceedings of the 5th TAG+ Workshop*, 2000.

[PM79] F.P. Preparata and D.E. Muller. Finding the intersection of n half-spaces in time $o(n \log n)$. *Theoretical Computer Science*, 8:45–55, 1979.

[PRB01] T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *LNCS*, pages 451–463, 2001.

[Pug98] J.-F. Puget. A Fast Algorithm for the Bound Consistency of Alldiff Constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 359–366, 1998.

[QvBLO+03] C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S.B. Sadjad. An Efficient Bound Consistency Algorithm for the Global Cardinality Constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 600–614, 2003.

[Rég94] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.

[Rég96] J.-C. Régin. Generalized Arc-Consistency for Global Cardinality Constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 209–215, 1996.

[Rég99] J.-C. Régin. Arc-Consistency for Global Cardinality Constraints with Costs. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP 1999)*, volume 1713 of *LNCS*, pages 390–404, 1999.

[Rey93] U. Reyle. Dealing with ambiguities by underspecification: construction, representation, and deduction. *Journal of Semantics*, 10:123–179, 1993.

[RVSW95] O. Rambow, K. Vijay-Shanker, and D. Weir. D-Tree Grammars. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 151–158, 1995.

[Sel02] M. Sellmann. An Arc-Consistency Algorithm for the Minimum Weight All Different Constraint. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *LNCS*, pages 744–749, 2002.

[SH74] I.E. Sutherland and G.W. Hodgman. Reentrant Polygon Clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[Sim88] K. Simon. An Improved algorithm for Transitive Closure on Acyclic Digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

[Smo95] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, pages 324–343. Springer, 1995.

[SS03] C. Schulte and G. Smolka. *Finite Domanin Constraint Programming in Oz. A Tutorial*, 1.2.5 edition, 2003. Available at `www.mozart-oz.org`.

[TW67] J.W. Thatcher and J.B. Wright. Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic. *Mathematical Systems Theory*, 2(1):57–81, 1967.

[vH01a] W.J. van Hoeve. The alldifferent constraint: A Survey. Extended version of [vH01b]. Preliminary version at `http://www.cwi.nl/~wjvh`, 2001.

[vH01b] W.J. van Hoeve. The alldifferent constraint: A Survey. In *Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

[VS92] K. Vijay-Shanker. Using Descriptions of Trees in a Tree Adjoining Grammar. *Computational Linguistics*, 18:481–518, 1992.

[Zho97] J. Zhou. A Permutation-Based Approach for Solving the Job-Shop Problem. *Constraints*, 2(2):185–213, 1997.