Operations Research Techniques in Constraint Programming

ILLC Dissertation Series DS-2005-02

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-525 6051
fax: +31-20-525 5206
e-mail: illc@wins.uva.nl
homepage: http://www.illc.uva.nl/

Operations Research Techniques in Constraint Programming

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.mr. P.F. van der Heijden
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op dinsdag 19 april 2005, te 12.00 uur

door

Willem-Jan van Hoeve

geboren te Noordoostpolder

Promotiecommissie:

Promotor:
prof.dr. K.R. Apt

Overige leden:
prof.dr. P. van Emde Boas
prof.dr.ir. A.M.H. Gerards
prof.dr. M. de Rijke
prof.dr. A. Schrijver
dr. M. Laurent
dr. M. Milano
dr. L. Torenvliet

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Centrum voor Wiskunde en Informatica

The research reported in this thesis has been carried out at the Centrum voor Wiskunde en Informatica.

Printed and bound by PrintPartners Ipskamp, Enschede.

ISBN 90–6196–529–2

Preface

The past four and a half years I have been a PhD student at the CWI in Amsterdam, with great pleasure. I have carried out my research in the group PNA1, which contains researchers from operations research as well as constraint programming. This turned out to be a fertile basis for my investigations on the combination of the two fields. My supervisor, Krzysztof Apt, was working on a somewhat different research topic. To illustrate this, we have not written a single joint paper in all those years, although this is probably also due to his ethical views on science. As a result, I have performed my research rather independently, which I have appreciated very much. Nevertheless, Krzysztof has supported and advised me in many different ways. I want to thank him for his guidance and, perhaps as important, confidence.

Krzysztof realized that a background in constraint programming and operations research alone is not necessarily sufficient to perform interesting research on their combination. Hence, I was sent to Bologna, to collaborate with an expert in the field: Michela Milano. I have visited Bologna several times, and these visits turned out to be very fruitful. Apart from the papers we have written together, Michela has also acted as a mentor in the research process. I believe that her influence has been very important for my development. For all this I am very grateful to her.

Another pleasant aspect of my visits to Bologna has been the "working environment". I am very thankful to Andrea Lodi, Andrea Roli, Paolo Torroni, and all other members of the Drunk Brain Band for their hospitality, company, and (in some cases) interesting discussions about research issues.

During the past year I have also successfully collaborated with Gilles Pesant and Louis-Martin Rousseau. I want to thank them for this experience and for hosting me for a month in Montreal. Further, I am thankful to Eric Monfroy for interesting discussions and a very enjoyable visit to Nantes. Many of the above visits would not have been possible without the support of the CWI, which I have very much appreciated.

At the CWI, I have benefited enormously from the seminars and the knowledge of the members of PNA1. I want to thank all of them, in particular Bert

Gerards, Monique Laurent and Lex Schrijver, for their support. There are many other people that have made the CWI a stimulating and pleasant environment. Among them are Peter Zoeteweij, the members of PIMB (the CWI Christmas band), Pieter Jan 't Hoen, and last, but not least, Sebastian Brand, who has been an exemplary roommate.

Finally, I am very grateful to Aafke for her patience and support.

Amsterdam, March 2005 Willem-Jan van Hoeve

Table of Contents

Chapter 1

Introduction

1.1 Motivation

A *combinatorial problem* is the problem of finding an object with some desired property among a finite set of possible alternatives.

Many problems from industry exhibit a combinatorial nature. An example is the optimal routing of trucks to deliver goods from a depot to customers. There are many alternatives to distribute the goods among the trucks, and for each such distribution there are many alternative routes for each individual truck. Moreover, often we are restricted to deliver goods only within a certain time frame for each customer. This makes the search for an optimal solution even harder, because there may only be a few optimal solutions, respecting the time frames, among a huge set of possible alternatives. To solve combinatorial problems, we cannot simply consider all exponentially many possible alternatives.

Some combinatorial problems are solvable by an algorithm whose running time is bounded by a polynomial in the size of the representation of the problem. These problems are considered to be efficiently solvable, and are said to belong to the class P. For other problems such method is not known to exist and they are classified as follows. If we can determine in polynomial-time whether or not a particular alternative is a solution to a certain problem, the problem is said to be in the class NP. Note that all problems in P are also in NP. If a problem is in NP and moreover every other problem in NP can be transformed to this problem in polynomial time, the problem is said to be NP-complete. NP-complete problems are the hardest problems in NP. In this thesis we focus on solution methods for NP-complete combinatorial problems.

Several solution methods have been proposed to solve combinatorial problems faster. However, many real-life problems are still (yet) unsolvable, even when such techniques are applied, so further research is necessary. In this thesis we consider techniques from *operations research* and *constraint programming* to model and solve combinatorial problems.

Operations Research

The field of *operations research*, sometimes also called *management science*, became an independent discipline in the late 1930's, although its foundations were laid earlier. The emergence of this discipline was motivated by the need to efficiently plan military operations during world war II. Hence the name "operations" research. A proper definition for this field, including all of its still growing branches, is hard to provide. According to the Institute for Operations Research and the Management Sciences[1]:

> *"Operations Research and the Management Sciences are the professional disciplines that deal with the application of information technology for informed decision-making."*

However, this definition may not be sufficiently specific to distinguish it from related fields. In particular, constraint programming could also be viewed as a part of operations research, while we wish to distinguish the two in this thesis.

Instead of following a definition, we will use the term "operations research" to specify a particular set of methods and solution techniques for combinatorial problems. This set includes for example techniques from graph theory, (integer) linear programming and semidefinite programming. Many of these methods can be characterized by the fact that they provide *efficient solution techniques* for *specific problems*. Another common feature of these methods is that they are particularly useful for *optimization* problems.

Consider for example linear programming. In linear programming, a problem needs to be represented by linear equations (or inequalities) on continuous variables. Given such representation, linear programming can solve the problem in polynomial time. If the problem cannot be expressed by linear equations or inequalities on continuous variables, we cannot apply the efficient linear programming algorithms. This is often the case, for example because some of the problem variables should be integral.

Many operations research techniques have proven to be very useful in practice. For many real-life problems, operations research provides the method of choice, in particular those problems that match the above characterizations.

Constraint Programming

The field of *constraint programming* is relatively new; the first international workshop on "Principles and Practice of Constraint Programming" was held in 1993, while it became a conference in 1995. The basic concepts of constraint reasoning were developed in the field of *artificial intelligence* in the 1970s. Further development took place after the introduction of constraints in *logic programming* in the 1980s.

A *constraint* can be viewed as a restriction on the space of possible alternatives. For example, a constraint of the above truck routing problem is:

[1] See `http://www.informs.org/`.

"visit each customer within its time frame". Often one can formulate a combinatorial problem by means of a number of constraints. The idea of constraint programming is to systematically search through a set of possible alternatives, and to use each constraint individually to detect whether a certain set of alternatives may be omitted because it contains no solution. The latter process is called *constraint propagation*.

Because the constraints are verified individually, constraint programming can be characterized as being *flexible* and *generic*. For example, if a problem changes, we may simply add a new constraint without affecting the previous model. Another characterization is that constraint programming is particularly suitable for *feasibility* problems rather than optimization problems.

The generality of constraint programming may seem to limit its practical usage. However, it has been applied successfully to a number of real-life problem instances, for example scheduling problems.

A Combined Approach

When we want to solve a problem, we need to choose which method is most appropriate to apply. The differences between constraint programming and operations research should be taken into account when making this choice. For each problem, we should first identify its characteristics, and then apply the method that matches the problem's characteristics best.

Unfortunately, many practical problems are not characterized to exclusively match either constraint programming or operations research. Instead, they fit in the format of both fields. Consider for example again the problem of optimally routing trucks to deliver goods from a depot to customers, where we must visit each customer within its time frame. The optimization component of this problem typically asks for an operations research approach, while the scheduling component, arising from the time frames, is most suitable for constraint programming. This motivates the need for a *combined approach*.

Fortunately, the generality and flexibility of constraint programming allows the combination of the two fields. This leads to the topic of this thesis:

> **In this thesis we investigate the application of operations research techniques in constraint programming.**

We propose to apply operations research techniques both during the search phase and the propagation phase of constraint programming. During the propagation process we use techniques from graph theory. During search, we apply linear and semidefinite relaxations to guide the direction of search and to improve the optimization component.

The combination of operations research and constraint programming is a natural idea, because both methods can be used to solve the same problems, and they have complementary strengths and weaknesses. Many different combinations have been studied in the past. They range from specialized hybrid

algorithms for specific problems up to the complete integration of the two fields. The successful results have led to an annual international workshop on "the integration of artificial intelligence and operations research techniques in constraint programming for combinatorial optimisation problems" (CP-AI-OR), starting in 1999, which became a conference in 2004. We refer to Chandru and Hooker [1999], Hooker [2000] and Milano [2003] for a collection of successful combinations. More detailed references will be presented when appropriate in each following chapter.

1.2 Contributions and Outline

As described above, the solution method of constraint programming consists of *search* combined with *constraint propagation*. We propose to exploit operations research techniques in both components. The first part of the thesis is devoted to constraint propagation, and will mainly describe theoretical results. The second part considers search techniques. That part also contains a considerable amount of experimental results. Below we list the contributions of the thesis by chapter.

Part I: Propagation

A Systematic Overview of the Alldifferent Constraint. This chapter gives an overview of existing propagation algorithms for the alldifferent constraint. The contribution of this chapter is the presentation of previous results in a systematic way, based on the same combinatorial principles. Furthermore, several of the propagation algorithms are based on techniques from operations research. Hence, this chapter provides a basis for further application of such methods in constraint programming via propagation algorithms.

This chapter is an extended and revised version of the paper [van Hoeve, 2001].

Soft Global Constraints. In this chapter we consider *over-constrained* problems, for which no solution exists. In order to handle such problems some constraints are *softened*. Soft constraints induce a cost of violation and the aim is to minimize the total violation cost.
For many *soft global constraints* no efficient propagation algorithm was previously available. In this chapter we introduce a generic method for softening global constraints that can be represented by a flow in a graph. We represent the cost of violation by additional *violation arcs* and compute a minimum-cost flow in the extended graph. This can be done efficiently with an operations research method, i.e. flow theory. We apply our method

to a number of global constraints, and obtain efficient propagation algorithms.

Parts of this chapter are based on the papers [van Hoeve, 2004] and [van Hoeve, Pesant, and Rousseau, 2004]. It should be mentioned that the defendant made significant contributions to the latter (co-authored) paper.

Part II: Search

Postponing Branching Decisions. Constraint programming uses a *search tree* to find a solution to a problem. We propose to postpone a branching decision in case two or more branches are equally likely to be successful. We provide a theoretical and experimental analysis of this method. The framework will prove to be very useful in the next chapter, when we use information from a linear relaxation to guide the search.

This chapter is based on the papers [van Hoeve and Milano, 2003] and [van Hoeve and Milano, 2004]. It should be mentioned that the defendant made significant contributions to these papers and was responsible for the implementation of the method.

Reduced Costs as Branching Heuristic. In this chapter we propose to use *reduced costs* as branching heuristic in the constraint programming search tree. Reduced costs are obtained by solving a linear relaxation of the problem. By applying the "decision postponement" framework (see above) and *limited discrepancy search*, we also show how to improve the bound of this linear relaxation during search. Experimental results indicate the usefulness of our method.

This chapter is a revised version of the papers [Milano and van Hoeve, 2002a] and [Milano and van Hoeve, 2002b]. It should be mentioned that the defendant made significant contributions to these papers and was responsible for the implementation of the method.

Semidefinite Relaxation as Branching Heuristic. In this chapter we investigate the usage of a *semidefinite relaxation* in constraint programming. We apply the solution to the relaxation as a branching heuristic in the constraint programming search tree. Additionally, the solution yields a bound on the objective value. Computational results show that constraint programming can indeed benefit from semidefinite relaxations.

This chapter is based on the papers [van Hoeve, 2003b], [van Hoeve, 2003a] and [van Hoeve, 2005].

Chapter 2

Foundation

This chapter presents the fundaments on which we base our work. From operations research we introduce concepts from graph theory, linear programming and semidefinite programming. Constraint programming will be presented in more depth, because it is the skeleton of our solution method.

2.1 Operations Research

2.1.1 Graph Theory

This section is for a large part based on the book [Schrijver, 2003].

Basic Notions

A *graph* or *undirected graph* is a pair $G = (V, E)$, where V is a finite set of *vertices* and E is a family[1] of *unordered* pairs from V, called *edges*. An edge between $u \in V$ and $v \in V$ is denoted as uv.

Given a graph $G = (V, E)$, the *complement graph* of G is $\overline{G} = (V, \overline{E})$ where $\overline{E} = \{uv \mid u, v \in V, uv \notin E, u \neq v\}$.

We say that sets $S_1, S_2, \ldots, S_k$ are *(pairwise) disjoint* if $S_i \cap S_j = \varnothing$ for all distinct $i, j \in \{1, \ldots, k\}$. A *partition* of a set S is a collection of disjoint subsets of S with union S. A graph $G = (V, E)$ is *bipartite* if there exists a partition S, T of V such that $E \subseteq \{st \mid s \in S, t \in T\}$.

A *walk* in a graph $G = (V, E)$ is a sequence $P = v_0, e_1, v_1, \ldots, e_k, v_k$ where $k \geq 0$, $v_0, v_1, \ldots, v_k \in V$, $e_1, e_2, \ldots, e_k \in E$ and $e_i = (v_{i-1}, v_i)$ for $i = 1, \ldots, k$. If there is no confusion, P may be denoted as $v_0, v_1, \ldots, v_k$ or $e_1, e_2, \ldots, e_k$. A walk is called a *path* if $v_0, \ldots, v_k$ are distinct. A closed walk, i.e. $v_0 = v_k$, is called a *circuit* if $v_1, \ldots, v_k$ are distinct.

A graph $G = (V, E)$ is *connected* if for any two vertices $u, v \in V$ there is a path connecting u and v. A graph is a *tree* if it is connected and contains no circuits.

A *directed graph*, or a *digraph*, is a pair $G = (V, A)$ where V is a finite set of vertices and A is a family of *ordered* pairs from V, called *arcs*. A pair occurring more than once in A is called a *multiple arc*. An arc from $u \in V$

[1] A family is a set in which elements may occur more than once. In the literature it is also referred to as *multiset*.

to $v \in V$ is denoted as (u, v). For $v \in V$, let $\delta^{\text{in}}(v)$ and $\delta^{\text{out}}(v)$ denote the family of arcs entering and leaving v, respectively. A vertex v with $\delta^{\text{in}}(v) = 0$ is called a *source*. A vertex v with $\delta^{\text{out}}(v) = 0$ is called a *sink*.

A digraph $G = (V, A)$ is called a *rooted tree* if it has exactly one source and the underlying undirected graph is a tree. The source is called the *root*.

A *directed walk* in a digraph $G = (V, A)$ is a sequence $P = v_0, a_1, v_1, \ldots, a_k, v_k$ where $k \geq 0$, $v_0, v_1, \ldots, v_k \in V$, $a_1, a_2, \ldots, a_k \in A$ and $a_i = (v_{i-1}, v_i)$ for $i = 1, \ldots, k$. Again, if there is no confusion, P may be denoted as $v_0, v_1, \ldots, v_k$ or $a_1, a_2, \ldots, a_k$. A directed walk is called a *directed path* if $v_0, \ldots, v_k$ are distinct. A closed directed walk, i.e. $v_0 = v_k$, is called a *directed circuit* if $v_1, \ldots, v_k$ are distinct.

A digraph $G' = (V', A')$ is a *subgraph* of a digraph $G = (V, A)$ if $V' \subseteq V$ and $A' \subseteq A$.

A digraph $G = (V, E)$ is *strongly connected* if for any two vertices $u, v \in V$ there is a directed path from u to v. A maximally strongly connected nonempty subgraph of a digraph G is called a *strongly connected component*, or *SCC*, of G. Here 'maximal' is taken with respect to subgraphs. We sometimes identify an SCC with its vertices.

Matching Theory

We present only those concepts of matching theory that are necessary for this thesis. For more information on matching theory we refer to Lovász and Plummer [1986], Gerards [1995] and Schrijver [2003, Chapter 16–38].

Given a graph $G = (V, E)$, a *matching* in G is a set $M \subseteq E$ of disjoint edges, i.e. no two edges in M share a vertex. A matching is said to *cover* a set $S \subseteq V$ if all vertices in S are an endpoint of an edge in M. A vertex $v \in V$ is called *M-free* if M does not cover v. The *size* of a matching M is $|M|$. The *maximum matching problem* is the problem of finding a matching of maximum size in a graph.

Given a matching M in G, an *M-augmenting path* is a sequence $v_0, v_1, \ldots, v_k$, with $v_i \in V$ distinct, k is odd, and the edges (v_i, v_{i+1}) are alternatingly not in M (for $i \in \{0, 2, 4, \ldots, k-1\}$) and in M (for $i \in \{1, 3, 5, \ldots, k-2\}$). For k odd or even, we call the path *M-alternating*. An *M-alternating circuit* is a sequence $v_0, v_1, \ldots, v_k$ with $v_i \in V$, $v_1, \ldots, v_k$ distinct, $v_0 = v_k$, k is even, and the edges (v_i, v_{i+1}) are alternatingly in M and not in M. On an M-augmenting path, we can exchange edges in M and not in M, to obtain a matching M' with $|M'| = |M| + 1$. The following result is due to Petersen [1891].

Theorem 2.1. *Let $G = (V, E)$ be a graph, and let M be a matching in G. Then either M is a maximum-size matching, or there exists an M-augmenting path.*

Proof. If M is a maximum-size matching, then there exists no M-augmenting path, because otherwise exchangement of edges on this path gives a larger matching.

If M' is a matching larger than M, consider the graph $G' = (V, M \cup M')$. In G', each vertex is connected to at most two edges. Hence, each component of G' is either a circuit or a path (possibly of length zero). As $|M'| > |M|$ there is at least one component containing more edges of M' than of M. This component forms an M-augmenting path. □

Hence, a maximum-size matching can be found by iteratively computing M-augmenting paths in G and extending M. If G is bipartite, say with partition V_1, V_2 of V, we can compute a maximum-size matching in $O(|E|\sqrt{V_1})$ time, where we assume $|V_1| \leq |V_2|$, following Hopcroft and Karp [1973]. In general graphs a maximum-size matching can be computed in $O(|V|\,|E|)$ time [Edmonds, 1965] or even $O(|E|\sqrt{|V|})$ time [Micali and Vazirani, 1980].

Flow Theory

We present only those concepts of flow theory that are necessary for this thesis. For more information on flow theory we refer to Ahuja, Magnanti, and Orlin [1993] and Schrijver [2003, Chapter 6–15].

Let $G = (V, A)$ be a directed graph and let $s, t \in V$. A function $f : A \to \mathbb{R}$ is called a *flow from s to t*, or an *$s - t$ flow*, if

$$\begin{array}{ll} (i) & f(a) \geq 0 \qquad\qquad\qquad\quad \text{for each } a \in A, \\ (ii) & f(\delta^{\text{out}}(v)) = f(\delta^{\text{in}}(v)) \text{ for each } v \in V \setminus \{s, t\}. \end{array} \tag{2.1}$$

Here $f(S) = \sum_{a \in S} f(a)$ for all $S \subseteq A$. Property (2.1)(ii) ensures *flow conservation*, i.e. for a vertex $v \neq s, t$, the amount of flow entering v is equal to the amount of flow leaving v.

The *value* of an $s - t$ flow f is defined as

$$\text{value}(f) = f(\delta^{\text{out}}(s)) - f(\delta^{\text{in}}(s)).$$

In other words, the value of a flow is the net amount of flow leaving s. This is equal to the net amount of flow entering t.

Let $d : A \to \mathbb{R}_+$ and $c : A \to \mathbb{R}_+$ be a "demand" function and a "capacity" function, respectively. We say that a flow f is *feasible* if

$$d(a) \leq f(a) \leq c(a) \quad \text{for each } a \in A.$$

Let $w : A \to \mathbb{R}$ be a "weight" function. We often also refer to such function as a "cost" function. For a directed path P in G we define $w(P) = \sum_{a \in P} w(a)$. Similarly for a directed circuit. The *weight* of any function $f : A \to \mathbb{R}$ is defined as

$$\text{weight}(f) = \sum_{a \in A} w(a) f(a).$$

A feasible flow is called a *minimum-weight flow* if it has minimum weight among all feasible flows with the same value. Given a graph $G = (V, A)$ with $s, t \in V$ and a number $\phi \in \mathbb{R}_+$, the *minimum-weight flow problem* is: find a minimum-weight $s - t$ flow with value ϕ.

Let f be an $s-t$ flow in G. The *residual graph* of f (with respect to d and c) is defined as $G_f = (V, A_f)$ where

$$A_f = \{a \mid a \in A, f(a) < c(a)\} \cup \{a^{-1} \mid a \in A, f(a) > d(a)\}.$$

Here $a^{-1} = (v, u)$ if $a = (u, v)$. We extend w to $A^{-1} = \{a^{-1} \mid a \in A\}$ by defining

$$w(a^{-1}) = -w(a)$$

for each $a \in A$.

In order to compute a minimum-weight $s - t$ flow of a given value we use the following notation. Any directed path P in G_f gives an undirected path in $G = (V, A)$. We define $\chi^P \in \mathbb{R}^A$ by

$$\chi^P(a) = \begin{cases} 1 \text{ if } P \text{ traverses } a, \\ -1 \text{ if } P \text{ traverses } a^{-1}, \\ 0 \text{ if } P \text{ traverses neither } a \text{ nor } a^{-1}, \end{cases}$$

for $a \in A$. We define $\chi^C \in \mathbb{R}^A$ similarly for a directed circuit C in G_f.

Using the above notation, a feasible $s - t$ flow in G with value ϕ and minimum weight can be found using Algorithm 1. It is sometimes referred to as the *successive shortest path algorithm*. The algorithm consists of two phases. The first phase tries to find a feasible flow of minimum weight that respects the demand for all arcs. This is done by adding the arc (t, s) to G, whereafter we successively update the flow for each arc whose demand is not yet respected. Then we remove the arc (t, s) and continue with the second phase, if necessary. In the second phase we extend the flow of phase one, until its value is ϕ. It can be proved that for integer demand and capacity functions Algorithm 1 finds an integer $s-t$ flow with minimum weight; see for example Schrijver [2003, p. 175–176].

The time complexity of Algorithm 1 is $O(\phi \cdot \text{SP})$, where SP is the time to compute a shortest directed path in G. Although faster algorithms exist for general minimum-weight flow problems, this algorithm suffices when applied to our problems. This is because in our case the value of all flows is bounded by the number of variables of our application.

Given a minimum-weight $s - t$ flow, we want to compute the additional weight when an unused arc is forced to be used.

Theorem 2.2. *Let f be a minimum-weight $s-t$ flow of value ϕ in $G = (V, A)$ with $f(a) = 0$ for some $a \in A$. Let C be a directed circuit in G_f with $a \in C$,*

Algorithm 1 Minimum-weight $s-t$ flow of value ϕ in $G=(V,A)$

```
set f = 0
if d(a) > 0 for some a ∈ A then
  add the arc (t,s) with d(t,s) = 0, c(t,s) = φ, w(t,s) = 0 and f(t,s) = 0 to G
  while there exists an arc (u,v) with f(u,v) < d(u,v) do
    compute a directed v − u path P in G_f minimizing w(P)
    if P does not exist then
      stop
    else
      define the directed circuit C = P, u, v
    end if
    reset f = f + εχ^C, where ε is maximal subject to f + εχ^P ≤ c and
    f(u,v) + ε ≤ d(u,v)
  end while
  remove the arc (t,s) from G
end if
if value(f) ≥ φ then
  stop
else
  while value(f) < φ do
    compute a directed s − t path P in G_f minimizing w(P)
    if P does not exist then
      stop
    end if
    reset f = f + εχ^P, where ε is maximal subject to d ≤ f + εχ^P ≤ c and
    value(f) + ε ≤ φ
  end while
end if
```

minimizing $w(C)$. Then $f' = f + \varepsilon\chi^C$, where ε is subject to $d \le f + \varepsilon\chi^C \le c$, has minimum weight among all $s-t$ flows g in G with $\text{value}(g) = \phi$ *and* $g(a) = \varepsilon$. *If C does not exist, f' does not exist. Otherwise,* $\text{weight}(f') = \text{weight}(f) + \varepsilon \cdot w(C)$.

The proof of Theorem 2.2 relies on the fact that for a minimum-weight flow f in G, the residual graph G_f does not contain cycles with negative weight.

Other Graph Problems

Given a digraph $G=(V,A)$, a *directed Hamiltonian path* is a directed path P such that $v \in P$ for all $v \in V$. The *directed Hamiltonian path problem* is: given a digraph $G=(V,A)$ and $s,t \in V$, is there a directed Hamiltonian $s-t$ path in G? This problem is NP-complete; see problem GT39 in [Garey and Johnson, 1979].

Given a digraph $G=(V,A)$, a *directed Hamiltonian circuit* is a directed circuit C such that $v \in C$ for all $v \in V$. The *Asymmetric Travelling Salesman*

Problem, or *ATSP*, is: given a digraph $G = (V, A)$ and a "cost" function $w : A \to \mathbb{R}_+$, find a directed Hamiltonian circuit of minimum cost.

If $c(u, v) = c(v, u)$ for all distinct $u, v \in V$, the ATSP is called the *Symmetric Travelling Salesman Problem*, or *TSP*. Both the TSP and the ATSP are NP-hard problems; see problem ND22 in [Garey and Johnson, 1979].

Given a graph $G = (V, E)$, a *stable set* is a set $S \subseteq V$ such that no two vertices in S are joined by an edge in E. The *(weighted) stable set problem* is: given a graph $G = (V, E)$ and a "weight" function $w : E \to \mathbb{R}$, find a stable set with maximum total weight. Here "total weight" means the sum of the weights of the vertices in the stable set. The value of a stable set with maximum weight in a graph G is called the *weighted stable set number* and denoted by $\alpha(G)$[2]. In the unweighted case (when all weights are equal to 1), this problem amounts to the *maximum cardinality stable set problem*. Determining the weighted stable set number of a graph is an NP-complete problem; see problem GT20 in [Garey and Johnson, 1979].

Given a graph $G = (V, E)$, a *clique* is a set $S \subseteq V$ such that every two vertices in S are joined by an edge in E. The *maximum clique problem* is: given a graph $G = (V, E)$ and a "weight" function $w : E \to \mathbb{R}$, find a clique with maximum total weight. Again, "total weight" means the sum of the weights of the vertices in the clique. The value of a maximum weight clique in a graph G is called the *weighted clique number* and is denoted by $\omega(G)$[3]. Determining the weighted clique number of a graph is an NP-complete problem; see problem GT19 in [Garey and Johnson, 1979].

The weighted stable set number of a graph G is equal to the weighted clique number in the complement graph of G, i.e. $\alpha(G) = \omega(\overline{G})$. Hence, a maximum clique problem can be translated into a weighted stable set problem in the complement graph.

2.1.2 Linear Programming

In linear programming, a model is called a "program". It consists of continuous variables and linear constraints (inequalities or equalities). The aim is to optimize a linear cost function. There exist many textbooks on linear programming. A good introduction is given by Chvátal [1983]. A thorough theoretical overview is presented by Schrijver [1986]. In this section we mainly follow Nemhauser and Wolsey [1988].

One of the standard forms of a linear program is

[2] In the literature $\alpha(G)$ usually denotes the unweighted stable set number. The weighted stable set number is then denoted as $\alpha_w(G)$. In this thesis, it is not necessary to make this distinction.

[3] Similar to $\alpha(G)$ and $\alpha_w(G)$, we do not distinguish the usual notation for the weighted clique number $\omega_w(G)$ and $\omega(G)$.

$$\begin{aligned}
\min \quad & c_1x_1 + \quad c_2x_2 + \ldots + \quad c_nx_n \\
\text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
& a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
& \vdots \qquad\qquad\qquad\qquad \vdots \\
& a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \\
& x_1, \ldots, x_n \geq 0
\end{aligned}$$

or, using matrix notation,

$$\min \{c^\mathsf{T}x \mid Ax = b, x \geq 0\} \tag{2.2}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$ and $x \in \mathbb{R}^n$. Here c represents the "cost" vector and x is the vector of variables. Every linear program can be represented by a linear program in the form of (2.2); see for example Schrijver [1986, Section 7.4].

Recall that the *rank* of a matrix is the number of linearly independent rows or columns of the matrix. For simplicity, we assume in the following that the rank of A is m, i.e. there are no redundant equations in (2.2).

Let $A = (a_1, a_2, \ldots, a_n)$ where a_j is the j-th column of A. For some "index set" $I \subseteq \{1, \ldots, n\}$ we denote by A_I the submatrix of A consisting of the columns a_i with $i \in I$.

Because the rank of A is m, there exists an index set $B = \{B_1, \ldots, B_m\}$ such that the $m \times m$ submatrix $A_B = (a_{B_1}, \ldots, a_{B_m})$ is nonsingular. We call A_B a *basis* of A. Note that A_B is invertible because it is nonsingular. Let $N = \{1, \ldots, n\} \setminus B$. If we permute the columns of A such that $A = (A_B, A_N)$, we can write $Ax = b$ as

$$A_Bx_B + A_Nx_N = b,$$

where $x = (x_B, x_N)$. Then a solution to $Ax = b$ is given by $x_B = A_B^{-1}b$ and $x_N = 0$. This solution is called a *basic solution*. A basic solution is feasible if $A_B^{-1}b \geq \mathbf{0}$. The vector x_B contains the *basic variables* and the vector x_N contains the *nonbasic variables*.

Let A_B be a basis of A. Then we define the *reduced cost vector* $\overline{c} \in \mathbb{R}^n$ as

$$\overline{c}^\mathsf{T} = c^\mathsf{T} - c_B^\mathsf{T}A_B^{-1}A.$$

We have the following result (cf. Nemhauser and Wolsey [1988, page 31]):

Theorem 2.3. *(x_B, x_N) is an optimal solution if and only if $\overline{c} \geq \mathbf{0}$.*

To solve linear programs one often uses the *simplex method*, invented by Dantzig [1951], which applies Theorem 2.3. Roughly, the simplex method moves from one basis to another by replacing a column in A_B by a column in A_N, until it finds a basic feasible solution for which all reduced costs are nonnegative. The method is very fast in practice, although it has an exponential worst-case time complexity.

Despite the exponential worst-case time complexity of the simplex method, linear programs have been shown to be solvable in polynomial time. This was first proved theoretically by Khachiyan [1979] using the so-called ellipsoid method. A more practical polynomial-time algorithm for solving linear programs was presented by Karmarkar [1984a,b]. The latter is characterized as an "interior-point" method.

2.1.3 Semidefinite Programming

In this section we briefly introduce semidefinite programming. A general overview of semidefinite programming, covering many aspects, is given by Wolkowicz, Saigal, and Vandenberghe [2000]. An overview of semidefinite programming applied to combinatorial optimization is given by Goemans and Rendl [2000] and Laurent and Rendl [2004]. Further references can also be found on the web pages maintained by C. Helmberg[4] and F. Alizadeh[5].

A matrix $X \in \mathbb{R}^{n \times n}$ where $n > 0$ is said to be *positive semidefinite* (denoted by $X \succeq 0$) when $y^{\mathsf{T}} X y \geq 0$ for all vectors $y \in \mathbb{R}^n$. Semidefinite programming makes use of positive semidefinite matrices of variables. Semidefinite programs have the form

$$\begin{aligned} &\max \operatorname{tr}(WX) \\ &\text{s.t. } \operatorname{tr}(A_j X) \leq b_j \ (j = 1, \ldots, m) \\ &\quad X \succeq 0. \end{aligned} \tag{2.3}$$

Here $\operatorname{tr}(X)$ denotes the *trace* of X, which is the sum of its diagonal elements, i.e. $\operatorname{tr}(X) = \sum_{i=1}^{n} X_{ii}$. The matrix X, the cost matrix $W \in \mathbb{R}^{n \times n}$ and the constraint matrices $A_j \in \mathbb{R}^{n \times n}$ are supposed to be symmetric. The m reals b_j and the m matrices A_j define m constraints.

We can view semidefinite programming as an extension of linear programming. Namely, when the matrices W and A_j $(j = 1, \ldots, m)$ are all supposed to be diagonal matrices[6], the resulting semidefinite program is equal to a linear program, where the matrix X is replaced by a non-negative vector of variables $x \in \mathbb{R}^n$. In particular, then a semidefinite programming constraint $\operatorname{tr}(A_j X) \leq b_j$ corresponds to a linear programming constraint $a_j^{\mathsf{T}} x \leq b_j$, where a_j represents the diagonal of A_j.

Theoretically, semidefinite programs have been proved to be polynomially solvable to any fixed precision using the ellipsoid method; see Grötschel, Lovász, and Schrijver [1988]. In practice, nowadays fast interior point methods, which also run in polynomial time, are being used for this purpose; see Alizadeh [1995] for an overview.

[4] See `http://www-user.tu-chemnitz.de/~helmberg/semidef.html`.

[5] See `http://new-rutcor.rutgers.edu/~alizadeh/sdp.html`.

[6] A diagonal matrix is a matrix with nonnegative values on its diagonal entries only.

2.2 Constraint Programming

This section is for a large part based on the book [Apt, 2003].

2.2.1 Basic Notions

Let x be a variable. The *domain* of x is a set of values that can be assigned to x. In this thesis we only consider variables with *finite* domains.

Consider a finite sequence of variables $\mathcal{Y} = y_1, y_2, \ldots, y_k$ where $k > 0$, with respective domains $\mathcal{D} = D_1, D_2, \ldots, D_k$ such that $y_i \in D_i$ for all i. A *constraint* C on $\mathcal{Y}$ is defined as a subset of the Cartesian product of the domains of the variables in $\mathcal{Y}$, i.e. $C \subseteq D_1 \times D_2 \times \cdots \times D_k$. A constraint C is called a *binary constraint* if it is defined on two variables. If C is defined on more than two variables, we call C a *global constraint*.

A *constraint satisfaction problem*, or a *CSP*, is defined by a finite sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_n$ with respective domains $\mathcal{D} = D_1, D_2, \ldots, D_n$, together with a finite set of constraints $\mathcal{C}$, each on a subsequence of $\mathcal{X}$. To simplify notation, we often omit the braces "{ }" when presenting a specific set of constraints. A CSP P is also denoted as $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Consider a CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ with $\mathcal{X} = x_1, x_2, \ldots, x_n$ and $\mathcal{D} = D_1, D_2, \ldots, D_n$. A tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ *satisfies* a constraint $C \in \mathcal{C}$ on the variables $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ if $(d_{i_1}, d_{i_2}, \ldots, d_{i_m}) \in C$. A tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ is a *solution* to a CSP if it satisfies every constraint $C \in \mathcal{C}$.

A *consistent* CSP is a CSP for which a solution exists. An *inconsistent* CSP is a CSP for which no solution exists. A *failed* CSP is a CSP with an empty domain or with only singleton domains that together are not a solution to the CSP. A *solved* CSP is a CSP with only singleton domains that together are a solution to the CSP. Note that a failed CSP is also inconsistent, but not all inconsistent CSPs are failed.

Consider the CSPs $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and $P' = (\mathcal{X}, \mathcal{D}', \mathcal{C}')$. P and P' are called *equivalent* if they have the same solution set. P is said to be *smaller* than P' if they are equivalent and $D_i \subseteq D'_i$ for all i. This relation is written as $P \preceq P'$. P is *strictly smaller* than P', if $P \preceq P'$ and $D_i \subset D'_i$ for at least one i. This is written as $P \prec P'$. When both $P \preceq P'$ and $P' \preceq P$ we write $P \equiv P'$.

We fix the following notation. For a sequence of variables $K = x_1, x_2, \ldots, x_n$ we denote $D_K = \bigcup_{x_i \in K} D_i$. When the domain D of a variable x is a singleton, say $D = \{d\}$, we also write $x = d$.

Example 2.1. Let x_1, x_2, x_3 be variables with respective domains $D_1 = \{2, 3\}$, $D_2 = \{1, 2, 3, 4\}$, $D_3 = \{1, 2\}$. On these variables we impose the binary constraint $x_1 + x_2 \leq 4$ and the global constraint `alldifferent`(x_1, x_2, x_3). The latter states that the variables x_1, x_2 and x_3 should be pairwise different[7].

[7] A formal definition of the `alldifferent` constraint is given in Chapter 3.

We denote the resulting CSP as

$$x_1 \in \{2,3\}, x_2 \in \{1,2,3,4\}, x_3 \in \{1,2\},$$
$$x_1 + x_2 \leq 4,$$
$$\texttt{alldifferent}(x_1, x_2, x_3).$$

A solution to this CSP is $x_1 = 3$, $x_2 = 1$ and $x_3 = 2$. □

Often we want to find a solution to a CSP that is optimal with respect to certain criteria. A *constraint optimization problem*, or a *COP*, is a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{D} = D_1, \ldots, D_n$, together with an *objective function*

$$f : D_1 \times \cdots \times D_n \to \mathbb{Q}$$

to be optimized. An *optimal solution* to a COP is a solution to P that is optimal with respect to f. The objective function value is often represented by a variable z, together with the "constraint" `maximize` z or `minimize` z for a maximization or a minimization problem, respectively.

In constraint programming, the goal is to find a solution (or all solutions) to a given CSP, or an optimal solution (or all optimal solutions) to a given COP. The solution process interleaves *constraint propagation*, or *propagation* in short, and *search*.

2.2.2 Propagation

Constraint propagation removes (some) inconsistent values from the domains, based on considerations of the individual constraints. Doing so, the search space can be significantly reduced. Hence, constraint propagation is essential to make constraint programming solvers efficient.

Let C be a constraint on the variables $x_1, \ldots, x_m$ with respective domains $D_1, \ldots, D_m$. A *propagation algorithm* for C removes values from $D_1, \ldots, D_m$ that do not participate in a solution to C. A propagation algorithm does not need to remove *all* such values, as this may lead to an exponential running time due to the nature of some constraints.

Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP. We transform P into a smaller CSP P' by repeatedly applying the propagation algorithms for all constraints in $\mathcal{C}$ until there is no more domain reduction. This process is called *constraint propagation*. When the process terminates, we say that each constraint, and the CSP, is *locally consistent* and that we have *achieved* a notion of *local consistency* on the constraints and the CSP. The term "local consistency" reflects that we do not obtain a globally consistent CSP, but a CSP in which all constraints are "locally", i.e. individually, consistent. A thorough description of the process of constraint propagation is given by Apt [1999, 2003].

The strongest local consistency notion for a constraint is to demand that every domain value of every variable in the constraint belongs to a solution to

the constraint. For historic reasons, it is referred to as *hyper-arc consistency*. It does not guarantee a solution to the whole CSP though, because of other constraints that need to be satisfied.

Example 2.2. Consider again the CSP of Example 2.1, i.e. $x_1 \in D_1 = \{2, 3\}$, $x_2 \in D_2 = \{1, 2, 3, 4\}$, $x_3 \in D_3 = \{1, 2\}$, and

$$x_1 + x_2 \leq 4, \tag{2.4}$$

$$\texttt{alldifferent}(x_1, x_2, x_3). \tag{2.5}$$

We apply constraint propagation until both constraints are hyper-arc consistent:

$$\begin{array}{r} x_1 \in \{2,3\} \\ x_2 \in \{1,2,3,4\} \\ x_3 \in \{1,2\} \end{array} \overset{(2.4)}{\longrightarrow} \begin{array}{l} x_1 \in \{2,3\} \\ x_2 \in \{1,2\} \\ x_3 \in \{1,2\} \end{array} \overset{(2.5)}{\longrightarrow} \begin{array}{r} x_1 \in \{3\} \\ x_2 \in \{1,2\} \\ x_3 \in \{1,2\} \end{array} \overset{(2.4)}{\longrightarrow} \begin{array}{l} x_1 \in \{3\} \\ x_2 \in \{1\} \\ x_3 \in \{1,2\} \end{array} \overset{(2.5)}{\longrightarrow} \begin{array}{l} x_1 \in \{3\} \\ x_2 \in \{1\} \\ x_3 \in \{2\} \end{array}$$

The two constraints are examined sequentially, as indicated above the arcs. We first examine constraint (2.4), and deduce that values 3 and 4 in D_2 do not appear in a solution to it. Then we examine constraint (2.5), and remove value 2 from D_1. This is because x_2 and x_3 saturate values 1 and 2. Next we need to re-examine constraint (2.4) and remove value 2 from D_2. Then we consider constraint (2.5) again and remove value 1 from D_3.

The resulting CSP is hyper-arc consistent. In fact, we found a solution to the CSP. □

Constraint propagation is usually applied each time a domain has been changed, which happens very often during the solution process. Consequently, the method that we apply to make a CSP locally consistent should be as efficient as possible. The efficiency of constraint propagation is influenced by the order in which the propagation algorithms are applied, and by the efficiency of the propagation algorithms themselves. In this thesis we investigate the latter: efficient propagation algorithms to achieve a given notion of local consistency.

2.2.3 Search

Search Tree

The solution process of constraint programming uses a *search tree*, which is a particular rooted tree. The vertices of search trees are often referred to as *nodes*. The arcs of search trees are often referred to as *branches*. Further, if (u, v) is an arc of a search tree, we say that v is a *direct descendant* of u and u is the *parent* of v.

Definition 2.4 (Search tree). *Let P be a CSP. A* search tree *for P is a rooted tree such that*

- *its nodes are CSPs,*
- *its root is P,*
- *if $P_1, \ldots, P_m$ where $m > 0$ are all direct descendants of P_0, then the union of the solution sets of $P_1, \ldots, P_m$ is equal to the solution set of P_0 for every node P_0.*

We say that a node P of a search tree is *at depth d* if the length of the path from the root to P is d.

Definition 2.4 is a very general notion. In constraint programming, a search tree is dynamically built by splitting a CSP into smaller CSPs, until we reach a failed or a solved CSP. A CSP is split into smaller CSPs either by splitting a constraint (for example a disjunction) or by splitting the domain of a variable. In this thesis we only apply the latter.

At each node in the search tree we apply constraint propagation to the corresponding CSP. As a result, we may detect that the CSP is inconsistent, or we may reduce some domains of the CSP. In both cases less nodes need to be generated and traversed, so propagation can speed up the solution process. However, in order to be effective, constraint propagation must be efficient, i.e. the time spent on propagation should be less than the time that is gained by it.

Variable and Value Ordering Heuristics

To split the domain of a variable, we first select a variable and then decide how to split its domain. This process is guided by *variable* and *value ordering heuristics*. They impose an ordering on the variables and values, respectively. The order in which variables and values are selected has a great impact on the search process.

In order to introduce variable and value ordering heuristics, we need the following definitions. A relation $\preceq$ on a set S is called a *partial order* if it is reflexive ($s \preceq s$ for all $s \in S$), transitive ($s \preceq t$ and $t \preceq u$ implies $s \preceq u$), and antisymmetric ($s \preceq t$ and $t \preceq s$ implies $s = t$). A partial order $\preceq$ is a *total order* if $s \preceq t$ or $t \preceq s$ for all $s, t \in S$. Given a partial order $\preceq$ on a set S, an element $s \in S$ is called a *least* element if $s \preceq t$ for all $t \in S$. Two elements $s, t \in S$ are *incomparable* with respect to $\preceq$ if $s \not\preceq t$ and $t \not\preceq s$.

A *variable ordering heuristic* imposes a partial order on the variables with non-singleton domains. An example is the *most constrained first* variable ordering heuristic. It orders the variables with respect to their appearance in the constraints. A variable that appears the most often, is ordered least. It is likely that changing the domains of such variables will cause more values to be removed by constraint propagation. Another variable ordering heuristic is the *smallest domain first* heuristic. It is also referred to as the *first fail* heuristic. This heuristic orders the variables with respect to the size of their domains. A variable that has the smallest domain is ordered least. Advantages of this heuristic are that less nodes need to be generated in the search tree, and

that inconsistent CSPs are detected earlier. In case two or more variables are incomparable, we can for example apply the lexicographic ordering to these variables and obtain a total order.

A *value ordering heuristic* induces a partial order on the domain of a variable. It orders the values in the domain according to a certain criterion, such that values that are ordered least are selected first. An example is the *lexicographic* value ordering heuristic, which orders the values with respect to the lexicographic ordering. Another example is the *random* value ordering heuristic, which orders the variables randomly. In case a value ordering heuristic imposes a partial order on a domain, we can apply the lexicographic or random value ordering heuristic to incomparable values, to create a total order. A value ordering heuristic is also referred to as a *branching heuristic* because it decides the order of the branches in the search tree.

Domain Splitting Procedures

When we have selected a variable and a value ordering heuristic imposing a total order on its domain, we apply a domain splitting procedure. Given a domain, a *domain splitting procedure* generates a partition of the the domain. Examples of domain splitting procedures are *labelling* and *bisection*. Consider a domain $D = \{d_1, d_2, \ldots, d_m\}$ and a total order $\preceq$ such that $d_1 \preceq d_2 \preceq \cdots \preceq d_m$. Then labelling splits D into

$$\{d_1\}, \{d_2\}, \ldots, \{d_m\}.$$

In practice the labelling procedure is often implemented to split a domain D into

$$\{d_1\}, \{d_2, \ldots, d_m\}.$$

In the literature this procedure is also called *enumeration*.

Let $k = \lfloor m/2 \rfloor$. Then bisection splits the above domain D into

$$\{d_1, \ldots, d_k\}, \{d_{k+1}, \ldots, d_m\}.$$

Consider a CSP $P_0 = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and variable $x \in \mathcal{X}$ whose domain has been split into the partition $D_1, \ldots, D_k$. Then we define the directs descendants of P_0 as $P_i = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{x \in D_i\})$ for $i = 1, \ldots, k$. In practice, we adjust the domain a variable instead of adding a constraint to define a descendant. If the partition "respects" the value ordering heuristic that was applied to the domain, i.e. $d_i \preceq d_j$ for all $d_i \in D_i$, $d_j \in D_j$, $i < j$ and $i = 1, \ldots k-1$, the corresponding descendants inherit the ordering of the value ordering heuristic, i.e. $P_1 \preceq \cdots \preceq P_k$.

Search Strategies

A *search strategy* defines the *traversal* of the search tree. In this thesis we apply the search strategies *depth-first search* and *limited discrepancy search*.

We assume that all direct descendants of a node in a search tree are totally ordered, for example based on the value ordering heuristic. Then the least element corresponds to the first descendant.

First we describe *depth-first search*, or *DFS*:

> Start at the root node and proceed by descending to its first descendant. This process continues until we reach a leaf. Then we backtrack to the parent of the leaf and descend to its next descendant, if it exists. This process continues until we are back at the root node and all its descendants have been visited.

Next we describe *limited discrepancy search*, or *LDS*, introduced by Harvey and Ginsberg [1995]. LDS is motivated by the following idea. Suppose we have "good" heuristics to build the search tree, i.e. the first leaf that we visit is likely to be a solution. If this leaf is not a solution, it is likely that we only made a small number of mistakes along the path from the root to this leaf. Hence, we visit next the leaf nodes whose paths from the root differ only in one choice from the initial path. We continue this process by gradually visiting leaves with a higher discrepancy from the initial path. Formally, let P_0 be a node with ordered descendants $P_1, P_2, \ldots, P_m$. The *discrepancy* of P_i is the discrepancy of $P_0 + i - 1$ for $i = 1, 2, \ldots, m$. The discrepancy of the root node is 0. LDS can now be described as:

> Set the level of discrepancy $k = 0$. Start at the root node and proceed by descending to its first descendant provided that its discrepancy is not higher than k. This process continues until we reach a leaf. Then we backtrack to the parent of the leaf and descend to its next descendant, provided that it exists and its discrepancy is not higher than k. This process continues until we are back at the root node and all its descendants whose discrepancy is not higher than k have been visited. Set $k = k + 1$ and repeat this process until we are back at the root node and all its descendants have been visited.

Note that for a certain discrepancy $k > 0$, LDS also visits leaves with discrepancy less than k. So leaves are visited several times. In practice however, this is avoided by keeping track of the remaining depth to be searched. Let the discrepancy of a node P be d, and let the length of a path from P to a leaf be l, then we only consider descendants whose discrepancy is between $k - l$ and k. This search strategy is called *improved limited discrepancy search* [Korf, 1996]. An illustration of LDS is given in Figure 2.1.

It should be noted that both DFS and ILDS are *complete* (sometimes referred to as exact) search strategies, that are not redundant. In other words, they explore all paths from the root to a leaf exactly once. It has been shown by Harvey and Ginsberg [1995] that LDS almost always outperforms DFS, even when the value ordering heuristic is less informative.

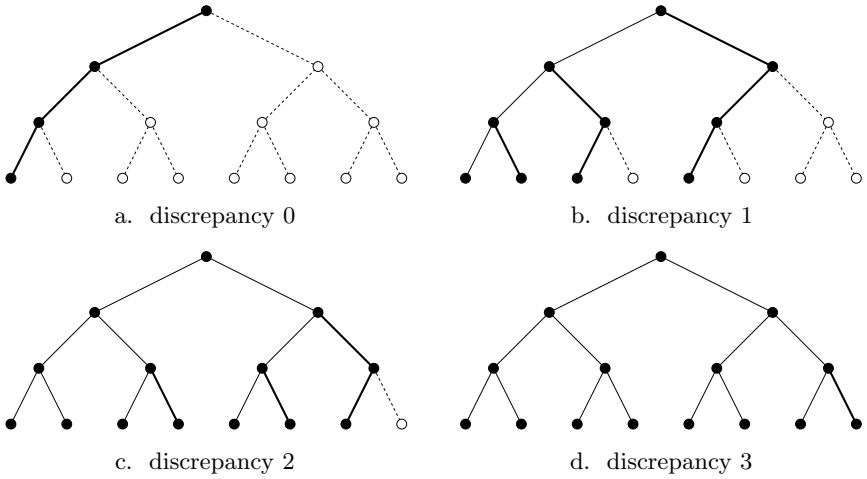


Figure 2.1. Limited discrepancy search. For each discrepancy the top node of the tree indicates the root, visited nodes are filled, and bold arcs indicate the parts of the tree that are traversed newly.

The two search strategies DFS and LDS have in common that backtracking to a previous node only takes place after we have visited a leaf. This leads us to a more general notion which we call *depth-first based search strategies*, described as:

> Start at the root node and proceed by descending to its first descendant. This process continues until we reach a leaf. Then we backtrack to some previously visited node and descend to its next descendant, if it exists and if it is allowed. This process continues until all leafs have been visited.

As stated above, DFS and LDS are examples of depth-first based search strategies. Other examples are *depth-bounded discrepancy search* or DDS, and *discrepancy-bounded depth-first search*, or *DBDFS* [Beck and Perron, 2000].

DDS, introduced by Walsh [1997], is based on the assumption that a heuristic is more likely to make a mistake near the root node. A motivation for this assumption is the removal of domain values due to propagation along a path from the root to a leaf. Hence, DDS allows discrepancies only above a certain depth-bound, which is gradually increased. At iteration i, the depth-bound is $i-1$. Below this depth, only descendants are allowed that do not increase the discrepancy. Experiments in [Walsh, 1997] show that DDS is competitive to, or even more efficient than, ILDS.

DBDFS is a variant of ILDS that performs depth-first search on all nodes on paths from the root that have up to some bounded number of discrepancies. Given a width k, the i-th iteration visits nodes with discrepancies between

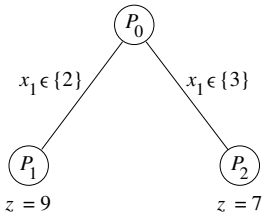


Figure 2.2. The search tree of Example 2.3.

$(i-1)k$ and $ik-1$. Experiments presented by Beck and Perron [2000] indicate that DBDFS has a competitive performance compared to LDS and DDS.

Optimization

The search for an *optimal* solution (or all optimal solutions) to a COP is performed similar to the search for a solution to a CSP. Recall that a COP consists of a CSP together with an objective function f to be optimized. Assume (without loss of generality) that we want to minimize f. The objective function value is represented by a variable z. When we find a solution to the CSP, the corresponding value of z, say $z = \beta$, serves as an upper bound for the optimal value of f. We then add the constraint $z < \beta$ to all CSPs in the search tree and continue. In practice, we replace the maximum value of the domain of z by β.

Example 2.3. We present the solution process of constraint programming, using the following CSP P_0:

$$
\begin{aligned}
&x_1 \in \{1,2,3\}, x_2 \in \{2,3\}, x_3 \in \{1,2,3\}, z \in \{6,\ldots,12\},\\
&\texttt{minimize}\ z,\\
&z = x_1 + 2x_2 + x_3,\\
&x_1 + x_2 \geq 5,\\
&\texttt{alldifferent}(x_1, x_2, x_3).
\end{aligned}
$$

To build a search tree, we apply the lexicographic variable and value ordering heuristics and use labelling as domain splitting procedure. As search strategy we use DFS. At each node we apply hyper-arc consistency constraint propagation. The CSP P_0 is the root. The search tree is depicted in Figure 2.2.

We first apply constraint propagation to P_0. It follows that

$$x_1 \in \{2,3\}, x_2 \in \{2,3\}, x_3 \in \{1\}, z \in \{7,\ldots,10\}.$$

We select the lexicographically least variable, x_1, split its domain into $\{2\}$ and $\{3\}$, and generate the descendants P_1 and P_2 where $P_1 = P_0$ with $x_1 \in \{2\}$ and $P_2 = P_0$ with $x_1 \in \{3\}$.

We descend to node P_1 and apply constraint propagation. It follows that

$$x_1 \in \{2\}, x_2 \in \{3\}, x_3 \in \{1\}, z \in \{9\}.$$

We have found a solution with $z = 9$. Hence we add to all CSPs the constraint $z < 9$.

Next we backtrack to P_0, descend to P_2, and apply constraint propagation. It follows that

$$x_1 \in \{3\}, x_2 \in \{2\}, x_3 \in \{1\}, z \in \{7\}.$$

We have found a solution with $z = 7$. Hence we add to all CSPs the constraint $z < 7$. Next we backtrack to P_0 and stop because all its descendants have been visited.

We return the optimal solution we found in leaf P_2. □

Part I

Propagation

Chapter 3

A Systematic Overview of the Alldifferent Constraint

This chapter surveys the most important developments over the years regarding the `alldifferent` *constraint. First we introduce the underlying concepts from combinatorial theory. Then we give an overview and a comparison of different propagation algorithms achieving arc, range, bounds and hyper-arc consistency, respectively. In addition, two variants of the* `alldifferent` *constraint are discussed: the symmetric* `alldifferent` *constraint and the minimum weight* `alldifferent` *constraint. Finally, the convex hull representation of the* `alldifferent` *constraint in integer linear programming is considered.*

3.1 Introduction

One of the constraints that is present in practically all constraint programming systems, is the famous `alldifferent` constraint, which states that all variables in this constraint must be pairwise different.

Already in the early days of constraint programming the importance of such "disequality constraints" was recognized. For example, in the 1970s Lauriere [1978] introduced ALICE, "A language and a program for stating and solving combinatorial problems". In this system the keyword "`DIS`", applied to a set of variables, is used to state that the variables must take different values. It defines a global structure (i.e. the set of variables form a "clique of disjunctions"), that is exploited during the search for a solution.

After the introduction of constraints in logic programming, for example in the system CHIP [Dincbas, Van Hentenryck, Simonis, Aggoun, Graf, and Berthier, 1988], it was also possible to express the constraint of difference as the well-known `alldifferent` constraint. In the system ECL^{i}PS^e [Wallace, Novello, and Schimpf, 1997] this constraint was introduced as `alldistinct`. However, in the early constraint (logic) programming systems the `alldifferent` constraint was treated internally as a sequence of disequalities; see for example Van Hentenryck [1989]. Unfortunately, the global information is lost in that way. The global view was retrieved with the propagation algorithm introduced by Régin [1994], that considers all disequalities simultaneously.

Throughout the history of constraint programming, the `alldifferent` constraint has played a special role. Various papers and books make use of this special constraint to show the benefits of constraint programming, either

to show its modelling power, or to show that problems can be solved faster using this constraint. From a modelling point of view, the `alldifferent` constraint arises naturally in problems that are based upon a permutation or when a directed graph has to be covered with disjoint circuits. Numerous applications exist in which the `alldifferent` constraint is of vital importance, for example quasi-group completion problems [Gomes and Shmoys, 2002], air traffic management [Barnier and Brisset, 2002; Grönkvist, 2004] and rostering problems [Tsang, Ford, Mills, Bradwell, Williams, and Scott, 2004]. Finally, many other global constraints can be viewed as an extension of the `alldifferent` constraint, for example the `sort` constraint [Older, Swinkels, and van Emden, 1995; Zhou, 1997], the `cycle` constraint [Beldiceanu and Contejean, 1994], the `diffn` constraint [Beldiceanu and Contejean, 1994] and the global cardinality constraint [Régin, 1996].

Over the years, the `alldifferent` constraint has been well-studied in constraint programming. As we will see, for the `alldifferent` constraint at least six different propagation algorithms exist, each achieving a different kind of local consistency, or achieving it faster. Many of these algorithms rely on techniques from operations research, i.e. flow theory and matching theory. Such propagation algorithms are of particular interest concerning the scope of this thesis.

The propagation algorithms often make use of the same underlying principles. To make them more understandable, accessible and coherent, this chapter presents a systematic overview of the `alldifferent` constraint, which may probably be regarded as the most well-known, most influential and most studied constraint in the field of constraint programming.

This chapter is organized as follows. In Section 3.2 we present results from combinatorial theory with respect to the `alldifferent` constraint. Many of the considered propagation algorithms rely on these results. In Section 3.3 we formally define different notions of local consistency that are applied to the `alldifferent` constraint.

Each of the Sections 3.4.1 up to 3.4.4 treats a different notion of local consistency. The sections are ordered in increasing strength of the considered local consistency. The treatment consists of a description of the particular notion of local consistency with respect to the `alldifferent` constraint, together with a description of an algorithm that achieves that local consistency. Section 3.4 ends with a time complexity survey and a discussion.

Section 3.5 gathers two variants of the `alldifferent` constraint. First a symmetric version of the `alldifferent` constraint is considered. Then the weighted `alldifferent` constraint is presented, where a linear objective function in conjunction with the `alldifferent` constraint is exploited.

In Section 3.6 considers the `alldifferent` polytope, which is a particular description of the solution set of the `alldifferent` constraint, using linear

constraints. This description can be applied in integer linear programming models.

Finally, in Section 3.7 we end this chapter with some concluding remarks.

3.2 Combinatorial Background

3.2.1 Alldifferent and Bipartite Matching

This section shows the equivalence of a solution to the `alldifferent` constraint and a matching in a bipartite graph.

Definition 3.1 (Alldifferent constraint). *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. Then*

$$\texttt{alldifferent}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \in D_i, d_i \neq d_j \text{ for } i \neq j\}.$$

Definition 3.2 (Value graph). *Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, D_2, \ldots, D_n$. The bipartite graph $G = (X \cup D_X, E)$ with $E = \{x_i d \mid d \in D_i\}$ is called the* value graph *of X.*

Theorem 3.3. *Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, D_2, \ldots, D_n$. Let G be the value graph of X. Then $(d_1, \ldots, d_n) \in \texttt{alldifferent}(x_1, \ldots, x_n)$ if and only if $M = \{x_1 d_1, \ldots, x_n d_n\}$ is a matching in G.*

Proof. An edge $x_i d_i$ (for some $i \in \{1, \ldots, n\}$) in M corresponds to the assignment $x_i = d_i$. As no edges in M share a vertex, $x_i \neq x_j$ for all $i \neq j$. □

Note that the matching M in Theorem 3.3 covers X, and is therefore a maximum-size matching.

Example 3.1. We want to assign four tasks (1, 2, 3 and 4) to five machines (A, B, C, D and E). To each machine at most one task can be assigned. However, not every task can be assigned to every machine. Table 3.1 below presents the possible combinations. For example, task 2 can be assigned to machines B and C.

Task	Machines
1	B, C, D, E
2	B, C
3	A, B, C, D
4	B, C

Table 3.1. Possible task - machine combinations.

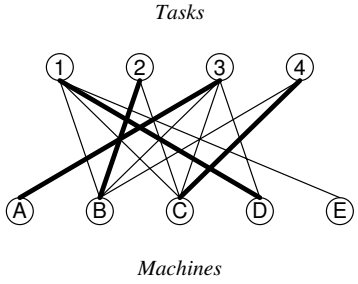


Figure 3.1. The value graph for the task assignment problem of Example 3.1. Bold edges form a matching covering all tasks.

This problem is modelled as follows. We introduce a variable x_i for task $i = 1, \ldots, 4$, whose value represents the machine to which task i is assigned. The initial domains of the variables are defined by the possible combinations in Table 3.1. Since the tasks have to be assigned to different machines, we introduce an `alldifferent` constraint. The problem is thus modelled as the CSP

$$x_1 \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}, x_2 \in \{\mathsf{B}, \mathsf{C}\}, x_3 \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}, x_4 \in \{\mathsf{B}, \mathsf{C}\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

The value graph of $X = x_1, \ldots, x_n$ is presented in Figure 3.1. The bold edges in the value graph denote a matching covering X. It corresponds to a solution to the CSP, i.e. $x_1 = \mathsf{D}, x_2 = \mathsf{B}, x_3 = \mathsf{A}$ and $x_4 = \mathsf{C}$. □

3.2.2 Hall's Marriage Theorem

A useful theorem to derive constraint propagation algorithms for the `alldifferent` constraint is Hall's *Marriage Theorem*[1] [Hall, 1935]:

> If a group of men and women marry only if they have been introduced to each other previously, then a complete set of marriages is possible if and only if every subset of men has collectively been introduced to at least as many women, and vice versa[2].

The following formulation is stated in terms of the `alldifferent` constraint.

Theorem 3.4. *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. The constraint* `alldifferent`$(x_1, \ldots, x_n)$ *has a solution if and only if*

$$|K| \leq |D_K| \tag{3.1}$$

for all $K \subseteq \{x_1, \ldots, x_n\}$.

[1] As noted by Schrijver [2003, p. 392], the name 'marriage theorem' was introduced by H. Weyl in 1949.

[2] Here we assume that marriage is restricted to two persons of different sex.

Proof. The direct proof presented here is adapted from Schrijver [2003, p. 379], and originally due to Easterfield [1946]. Call a set K tight if equality holds in (3.1). Necessity of the condition being obvious, we prove sufficiency. We use induction, with the hypothesis that Theorem 3.4 holds for k variables with $k < n$.

If there is a $d \in D_n$ such that

$$\begin{array}{l} x_1 \in D_1 \setminus \{d\}, \ldots, x_{n-1} \in D_{n-1} \setminus \{d\}, \\ \texttt{alldifferent}(x_1, \ldots, x_{n-1}) \end{array} \tag{3.2}$$

has a solution, then we are done. Hence, we may assume the opposite, i.e. in (3.2), for each $d \in D_n$, there exists a subset $K \subseteq \{1, \ldots, n-1\}$ with $|K| > |D_K \setminus \{d\}|$. Then, by induction, $\texttt{alldifferent}(x_1, \ldots, x_{n-1})$, with $x_1 \in D_1, \ldots, x_{n-1} \in D_{n-1}$, has a solution if and only if for each $d \in D_n$ there is a tight subset $K \subseteq \{1, \ldots, n-1\}$ with $d \in D_K$. Choose any such tight subset K. Without loss of generality, $K = \{1, \ldots, k\}$. By induction, $\texttt{alldifferent}(x_1, \ldots, x_k)$ has a solution, using all values in D_K. Moreover,

$$\begin{array}{l} x_{k+1} \in D_{k+1} \setminus D_K, \ldots, x_n \in D_n \setminus D_K, \\ \texttt{alldifferent}(x_{k+1}, \ldots, x_n) \end{array}$$

has a solution. This follows inductively, since for each $L \subseteq \{k+1, \ldots, n\}$,

$$\left| \bigcup_{i \in L} (D_i \setminus D_K) \right| = \left| \bigcup_{i \in K \cup L} D_i \right| - |D_K| \geq |K \cup L| - |D_K| = |K| + |L| - |D_K| = |L|\,,$$

where the "$\geq$" relation is obtained by applying condition (3.1). Then $\texttt{alldifferent}(x_1, \ldots, x_n)$ has a solution, using all values in $D_K \cup D_L$. □

The following example shows an application of Theorem 3.4.

Example 3.2. Consider the following CSP

$$\begin{array}{l} x_1 \in \{2,3\}, x_2 \in \{2,3\}, x_3 \in \{1,2,3\}, x_4 \in \{1,2,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3, x_4). \end{array}$$

For any $K \subseteq \{x_1, x_2, x_3, x_4\}$ with $|K| \leq 3$, we have $|K| \leq |D_K|$. For $K = \{x_1, x_2, x_3, x_4\}$ however, $|K| > |D_K|$, and by Theorem 3.4 this CSP has no solution. □

3.3 Local Consistency Notions

We now introduce four notions of local consistency of a constraint. In Section 3.4 we present propagation algorithms for each of them, when applied to the `alldifferent` constraint.

Definition 3.5 (Arc consistency). *A binary constraint C on the variables x_1 and x_2 with respective domains D_1 and D_2 is called* arc consistent *if for all values $d_1 \in D_1$ there exists a value $d_2 \in D_2$ such that $(d_1, d_2) \in C$, and for all values $d_2 \in D_2$ there exists a value $d_1 \in D_1$ such that $(d_1, d_2) \in C$.*

Definition 3.6 (Hyper-arc consistency). *A constraint C on the variables $x_1, \ldots, x_m$ with respective domains $D_1, \ldots, D_m$ is called* hyper-arc consistent *if for each variable x_i and each value $d_i \in D_i$, there exist a value $d_j \in D_j$ for all $j \neq i$ such that $(d_1, \ldots, d_m) \in C$.*

Note that arc consistency is equal to hyper-arc consistency applied to binary constraints. Both arc consistency and hyper-arc consistency ensure that all values in every domain belong to a tuple that satisfies the constraint, with respect to the current variable domains.

For a finite, linearly ordered domain D_i, we define $\min D_i$ and $\max D_i$ to be its minimum value and its maximum value, respectively. We use braces "{ }" and brackets "[ ]" to indicate a set and an interval of domain values, respectively. Thus, the set $\{1, 3\}$ contains the values 1 and 3 whereas the interval $[1, 3] \subset \mathbb{N}$ contains 1, 2 and 3.

Definition 3.7 (Bounds consistency). *A constraint C on the variables $x_1, \ldots, x_m$ with respective domains $D_1, \ldots, D_m$ is called* bounds consistent *if for each variable x_i and each value $d_i \in \{\min D_i, \max D_i\}$, there exist a value $d_j \in [\min D_j, \max D_j]$ for all $j \neq i$ such that $(d_1, \ldots, d_m) \in C$.*

Definition 3.8 (Range consistency). *A constraint C on the variables $x_1, \ldots, x_m$ with respective domains $D_1, \ldots, D_m$ is called* range consistent *if for each variable x_i and each value $d_i \in D_i$, there exist a value $d_j \in [\min D_j, \max D_j]$ for all $j \neq i$ such that $(d_1, \ldots, d_m) \in C$.*

Range consistency does not ensure the feasibility of the constraint with respect to the domains, but with respect to intervals that include the domains. It can be regarded as a relaxation of hyper-arc consistency. Bounds consistency can be in turn regarded as a relaxation of range consistency. It does not even ensure feasibility of the constraint for all values in the domains, but only for the minimum and the maximum value, while still verifying the constraint with respect to intervals that include the domains. This is formalized in Theorem 3.10.

Definition 3.9 (Locally consistent CSP). *A CSP is arc consistent, respectively range consistent, bounds consistent or hyper-arc consistent if all its constraints are.*

If we apply to a CSP P a propagation algorithm that achieves range consistency on P, we denote the result by $\Phi_{\mathrm{R}}(P)$. Analogously, $\Phi_{\mathrm{B}}(P)$, $\Phi_{\mathrm{A}}(P)$ and $\Phi_{\mathrm{HA}}(P)$ denote the application of bounds consistency, arc consistency and hyper-arc consistency on P, respectively.

Theorem 3.10. *Let P be CSP. Then $\Phi_{\mathrm{HA}}(P) \preceq \Phi_{\mathrm{R}}(P) \preceq \Phi_{\mathrm{B}}(P)$ and all relations may be strict.*

Proof. Both hyper-arc consistency and range consistency verify all values of all domains. But hyper-arc consistency verifies the constraints with respect to the exact domains D_i, while range consistency verifies the constraints with respect to intervals that include the domains: $[\min D_i, \max D_i]$. A constraint that holds on a domain D_i also holds on the interval $[\min D_i, \max D_i]$ since $D_i \subseteq [\min D_i, \max D_i]$. The converse is not true, see Example 3.3. Hence $\Phi_{\mathrm{HA}}(P) \preceq \Phi_{\mathrm{R}}(P)$, possibly strict.

Both range consistency and bounds consistency verify the constraints with respect to intervals that include the domains. But bounds consistency only considers $\min D_i$ and $\max D_i$ for a domain D_i, while range consistency considers all values in D_i. Since $\{\min D_i, \max D_i\} \subseteq D_i$, $\Phi_{\mathrm{R}}(P) \preceq \Phi_{\mathrm{B}}(P)$. Example 3.3 shows that $\Phi_{\mathrm{R}}(P) \prec \Phi_{\mathrm{B}}(P)$ cannot be discarded. □

Example 3.3. Consider the following CSP

$$P = \begin{cases} x_1 \in \{1,3\}, x_2 \in \{2\}, x_3 \in \{1,2,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3). \end{cases}$$

Then $\Phi_{\mathrm{B}}(P) \equiv P$, while

$$\Phi_{\mathrm{R}}(P) = \begin{cases} x_1 \in \{1,3\}, x_2 \in \{2\}, x_3 \in \{1,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3) \end{cases}$$

and $\Phi_{\mathrm{HA}}(P) \equiv \Phi_{\mathrm{R}}(P)$. Next, consider the CSP

$$P' = \begin{cases} x_1 \in \{1,3\}, x_2 \in \{1,3\}, x_3 \in \{1,3\}, \\ \texttt{alldifferent}(x_1, x_2, x_3). \end{cases}$$

This CSP is obviously inconsistent, since there are only two values available, namely 1 and 3, for three variables that must be pairwise different. Indeed, $\Phi_{\mathrm{HA}}(P')$ is a failed CSP, while $\Phi_{\mathrm{R}}(P') \equiv P'$. □

3.4 Propagation for Local Consistency Notions

This section analyzes four notions of local consistency when applied to the `alldifferent` constraint: arc consistency of a decomposed CSP, bounds consistency, range consistency and hyper-arc consistency. For each local consistency notion a corresponding constraint propagation algorithm is presented.

3.4.1 Local Consistency of a Decomposed CSP

In order to apply arc consistency, we decompose the `alldifferent` constraint into a set of binary constraints that preserves the solution set.

Definition 3.11 (Binary decomposition). *Let C be a constraint on the variables $x_1, \ldots, x_n$. A* binary decomposition *of C is a minimal set of binary constraints $C_{\text{dec}} = \{C_1, \ldots, C_k\}$ (for integer $k > 0$) on the variables $x_1, \ldots, x_n$ such that the solution set of C equals the solution set of $\bigwedge_{i=1}^{k} C_i$.*

Note that we can extend the definition of binary decomposition by defining the constraints in C_{dec} on arbitrary variables, such that the solution set of $\bigwedge_{i=1}^{k} C_i$ is mapped to the solution set of C and vice versa, as proposed by Rossi, Petrie, and Dhar [1990]. In this thesis this extension is not necessary, however.

The binary decomposition of $\texttt{alldifferent}(x_1, x_2, \ldots, x_n)$ is

$$\bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}. \tag{3.3}$$

A constraint propagation algorithm that achieves arc consistency on this binary decomposition (3.3) has the following behaviour:

> Whenever the domain of a variable contains only one value, remove this value from the domains of the other variables that occur in the `alldifferent` constraint. Repeat this procedure as long as no more changes occur or a domain becomes empty.

One of the drawbacks of this method is that one needs $\frac{1}{2}(n^2 - n)$ disequalities to express an n-ary `alldifferent` constraint. Moreover, the worst-case time complexity of this method is $O(n^2)$, as shown by the following example.

Example 3.4. For some integer $n > 1$, consider the CSP

$$P = \begin{cases} x_1 \in \{1\}, x_i \in \{1, 2, \ldots, i\} \;\text{ for } i = 2, 3, \ldots, n, \\ \bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\} \end{cases}$$

If we make P arc consistent, we start by removing $\{1\}$ from all domains other than D_1. Next we need to remove value $\{i\}$ from all domains other than D_i, for $i = 2, \ldots, n$. This procedure takes in total $n(n-1)$ steps. □

Another, even more important, drawback of the above method is the loss of information. When the set of binary constraints is being made arc consistent, only two variables are compared at a time. However, when the `alldifferent` constraint is being made hyper-arc consistent, all variables are considered at the same time, which allows a much stronger local consistency. This is shown in Theorem 3.12; see also Stergiou and Walsh [1999].

Theorem 3.12. *Let P be a CSP and P_{dec} the same CSP in which all* `alldifferent` *constraints have been replaced by their binary decomposition. Then $\Phi_{\text{HA}}(P) \preceq \Phi_{\text{A}}(P_{\text{dec}})$.*

Proof. To show that $\Phi_{\text{HA}}(P) \preceq \Phi_{\text{A}}(P_{\text{dec}})$, consider a value $d \in D_i$ for some i that is removed after making P_{dec} arc consistent. This removal must be due

to the fact that $x_j = d$ for some $j \neq i$. But then $d \in D_i$ is also removed when making P hyper-arc consistent. The converse is not true, as illustrated in Example 3.5. □

Example 3.5. For some integer $n \geq 3$ consider the CSPs

$$P = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \text{ for } i = 1, 2, \ldots, n-1, \\ x_n \in \{1, 2, \ldots, n\}, \\ \texttt{alldifferent}(x_1, x_2, \ldots, x_n), \end{cases}$$

$$P_{\text{dec}} = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \text{ for } i = 1, 2, \ldots, n-1, \\ x_n \in \{1, 2, \ldots, n\}, \\ \bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}. \end{cases}$$

Then $\Phi_{\text{A}}(P_{\text{dec}}) \equiv P_{\text{dec}}$, while

$$\Phi_{\text{HA}}(P) = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \text{ for } i = 1, 2, \ldots, n-1, \\ x_n \in \{n\}, \\ \texttt{alldifferent}(x_1, x_2, \ldots, x_n). \end{cases}$$

□

Our next goal is to find a local consistency notion for the set of disequalities that is equivalent to the hyper-arc consistency notion for the `alldifferent` constraint. Relational consistency [Dechter and van Beek, 1997] is used for this.

Definition 3.13 (Relational $(1, m)$-consistency). *A set of constraints $S = \{C_1, C_2, \ldots, C_m\}$ is relationally $(1, m)$-consistent if all domain values $d \in D_i$ of variables appearing in S, appear in a solution to the m constraints, evaluated simultaneously. A CSP $P = (\mathcal{X}, D, C)$ is relationally $(1, m)$-consistent if every set of m constraints $S \subseteq \mathcal{C}$ is relationally $(1, m)$-consistent.*

Note that arc consistency is equivalent to relational $(1, 1)$-consistency.

Let $\Phi_{\text{RC}(1,m)}(P)$ denote the CSP after achieving relational $(1, m)$-consistency on a CSP P.

Theorem 3.14. *Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D = D_1, D_2, \ldots, D_n$. Let $P = (X, D, C)$ be the CSP with $C = \{\texttt{alldifferent}(x_1, \ldots, x_n)\}$ and let $P_{\text{dec}} = (X, D, C_{\text{dec}})$ be the CSP with $C = \bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}$. Then*

$$\Phi_{\text{HA}}(P) \equiv \Phi_{\text{RC}(1, \frac{1}{2}(n^2 - n))}(P_{\text{dec}}).$$

Proof. By construction, the `alldifferent` constraint is equivalent to the simultaneous consideration of the sequence of corresponding disequalities. The number of disequalities is precisely $\frac{1}{2}(n^2-n)$. If we consider only $\frac{1}{2}(n^2-n)-i$ disequalities simultaneously ($1 \leq i \leq \frac{1}{2}(n^2-n)-1$), there are i unconstrained relations between variables, and the corresponding variables could take the same value when a certain instantiation is considered. Therefore, we really need to take all $\frac{1}{2}(n^2-n)$ constraints into consideration, which corresponds to relational $(1, \frac{1}{2}(n^2-n))$-consistency. □

As suggested before, the pruning performance of $\Phi_{\mathrm{A}}(P_{\mathrm{dec}})$ is rather poor. Moreover, the time complexity is relatively high, namely $O(n^2)$, as shown in Example 3.4. Nevertheless, this propagation algorithm applies quite well to several problems, such as the n-queens problem ($n < 200$), as indicated by Leconte [1996] and Puget [1998].

Further comparison of non-binary constraints and their corresponding decompositions are given by Stergiou and Walsh [1999] and Gent, Stergiou, and Walsh [2000]. In particular the `alldifferent` constraint and its binary decomposition are extensively studied.

3.4.2 Bounds Consistency

A bounds consistency propagation algorithm for the `alldifferent` constraint was first introduced by Puget [1998]. We summarize his method in this section. The idea is to use Hall's Marriage Theorem to construct an algorithm that achieves bounds consistency.

Definition 3.15 (Hall interval). *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. Given an interval I, define $K_I = \{x_i \mid D_i \subseteq I\}$. I is a* Hall interval *if $|I| = |K_I|$.*

Theorem 3.16. *The constraint* `alldifferent`$(x_1, \ldots, x_n)$ *is bounds consistent if and only if $|D_i| \geq 1$ $(i = 1, \ldots, n)$ and*

i) for each interval I: $|K_I| \leq |I|$,
ii) for each Hall interval I: $\{\min D_i, \max D_i\} \cap I = \varnothing$ for all $x_i \notin K_I$.

Proof. Let I be a Hall interval and $x_i \notin K_I$. If `alldifferent`$(x_1, \ldots, x_n)$ is bounds consistent, it has a solution when $x_i = \min D_i$, by Definition 3.7. From Theorem 3.4 immediately follows that $\min D_i \notin I$. Similarly for $\max D_i$.

Conversely, suppose `alldifferent`$(x_1, \ldots, x_n)$ is not bounds consistent. Thus, there exist a variable x_i and a value $d_i \in \{\min D_i, \max D_i\}$ for some $i \in \{1, \ldots, n\}$, such that `alldifferent`$(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ has no solution, where $x_j \in D'_j = [\min(D_j \setminus \{d_i\}), \max(D_j \setminus \{d_i\})]$ for all $j \neq i$. By Theorem 3.4, there exists some $K \subseteq \{x_1, \ldots, x_n\} \setminus \{x_i\}$ such that $|K| > |D'_K|$. Choose $I = D'_K$ and consider K_I with respect to `alldifferent`$(x_1, \ldots, x_n)$. Then either I is a Hall interval and $d_i \in K_I$, or $|K| > |K_I|$. □

Example 3.6. Consider the following CSP

$$x_1 \in \{1,2\}, x_2 \in \{1,2\}, x_3 \in \{2,3\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3).$$

Observe that the variables x_1 and x_2 both have domain $\{1,2\}$, Hence, values 1 and 2 cannot be assigned to any other variable and therefore, value 2 should be removed from D_3.

The algorithm detects this when the interval I is set to $I = [1,2]$. Then the number of variables for which $D_i \subseteq I$ is 2. Since $|I| = 2$, I is a Hall interval. The domain of x_3 is not in this interval, and $\{\min D_3, \max D_3\} \cap I = \{\min D_3\}$. In order to obtain the empty set in the right hand side of the last equation, we need to remove $\min D_i$. The resulting CSP is bounds consistent. □

Theorem 3.16 is used to construct an algorithm that achieves bounds consistency on the `alldifferent` constraint. Consider all intervals $I = [l,u]$ where l ranges over all minimum domain values and u over all maximum domain values. There are maximally n^2 such intervals, as there are n variables. Count the number of variables that are contained in I. If a Hall interval is detected, update the bounds of the appropriate variables. This can be done in $O(n)$ steps. Hence the time complexity of this algorithm is $O(n^3)$.

However, it is more efficient to first sort the variables. To update the minimum domain values, we sort the variables in increasing order of their maximum domain value. Then we maintain an interval by the minimum and maximum bounds of the variables, which are inserted in the above order. Whenever we detect a maximum-length Hall interval, we update the bounds of the appropriate variables, reset the interval and continue with the next variable. To update the maximum domain values, we can apply the same method after interchanging $[\min D_i, \max D_i]$ with $[-\max D_i, -\min D_i]$ for all i.

The sorting of the variables can be done in $O(n \log n)$ time. As shown by Puget [1998], we can use a balanced binary tree to keep track of the number of variables within an interval, which allows updates in $O(\log n)$ per variable. Hence the total time complexity reduces to $O(n \log n)$.

An improvement on top of this was suggested and implemented by Lopez-Ortiz, Quimper, Tromp, and van Beek [2003]. They noticed that while keeping track of the number of variables within an interval, some of the used counters are irrelevant. They give two implementations, one with a linear running time plus the time needed to sort the variables, and one with an $O(n \log n)$ running time. Their experiments show that the latter algorithm is faster in practice.

A different algorithm for achieving bounds consistency of the `alldifferent` constraint was presented by Mehlhorn and Thiel [2000]. Instead of Hall intervals, they exploit the correspondence with finding a matching in a bipartite graph, similar to the algorithm presented in Section 3.4.4. Their algorithm runs in $O(n)$ time plus the time needed to sort the variables according to the bounds of the domains.

Although the worst-case time complexity of the above algorithms is always $O(n \log n)$, under certain conditions the sorting can be performed in linear time, which makes the algorithms by Lopez-Ortiz et al. [2003] and Mehlhorn and Thiel [2000] run in linear time. This is the case in many practical instances, for example when the variables encode a permutation.

3.4.3 Range Consistency

Leconte [1996] introduced an algorithm that achieves range consistency for the `alldifferent` constraint. To explain this algorithm we follow the same procedure as in the previous subsection. Again we use Hall's Marriage Theorem to construct the algorithm.

Definition 3.17 (Hall set). *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. Given $K \subseteq \{x_1, \ldots, x_n\}$, define the interval $I_K = [\min D_K, \max D_K]$. K is a* Hall set *if $|K| = |I_K|$.*

Note that in the above definition I_K does not necessarily need to be a Hall interval, because $K_{I_K} = \{x_i \mid D_i \subseteq I_K\} \supseteq K$ (see Definition 3.15).

Theorem 3.18. *The constraint* `alldifferent`$(x_1, \ldots, x_n)$ *is range consistent if and only if $|D_i| \geq 1$ $(i = 1, \ldots, n)$ and $D_i \cap I_K = \varnothing$ for each Hall set $K \subseteq \{x_1, \ldots x_n\}$ and each $x_i \notin K$.*

Proof. Let K be a Hall set and $x_i \notin K$. If `alldifferent`$(x_1, \ldots, x_n)$ is range consistent, it has a solution when $x_i = d$ for all $d \in D_i$, by Definition 3.8. From Theorem 3.4 immediately follows that $D_i \cap I_K = \varnothing$.

Conversely, suppose `alldifferent`$(x_1, \ldots, x_n)$ is not range consistent. Thus, there exist a variable x_i and a value $d_i \in D_i$ for some $i \in \{1, \ldots, n\}$, such that `alldifferent`$(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ has no solution, where $x_j \in D'_j = [\min D_j \setminus \{d_i\}, \max D_j \setminus \{d_i\}]$ for all $j \neq i$. By Theorem 3.4, there is some $K \subseteq \{x_1, \ldots, x_n\} \setminus \{x_i\}$ with $|K| > |D'_K|$. Note that $D'_K = I_K$. Consider I_K with respect to `alldifferent`$(x_1, \ldots, x_n)$. If K is a Hall set, then $d_i \in I_K$. Otherwise, $|K| > |I_K|$. Then either some domain is empty, or K contains a Hall set K', and $D_j \cap I_{K'} \neq \varnothing$ for some $x_j \in K \setminus K'$. □

We can deduce a first propagation algorithm from Theorem 3.18 in a similar way as we did for bounds consistency. Namely, consider all intervals $I = [l, u]$ where l ranges over all minimum domain values and u over all maximum domain values. Then we count again the number of variables that are contained in I. If a Hall set is detected, we update the domains of the appropriate variables. This is one in $O(n)$ steps. Hence the time complexity of this algorithm is $O(n^3)$, as there are again maximally n^2 intervals to consider.

A faster algorithm is presented by Leconte [1996]. We first sort (and store) the variables twice, according to their minimum and maximum domain value, respectively. The main loop considers the variables ordered by their maximum

domain value. For each such variable, we maintain the interval I_K and start adding variables to K. For this we consider all variables (inner loop), now sorted by their minimum domain value. When we detect a Hall set, updating the domains can be done in $O(1)$ time, either within or after the inner loop, and we proceed with the next variable. As sorting the variables can be done in $O(n \log n)$ time, the total time complexity of this algorithm is $O(n^2)$. This time complexity is optimal, as is illustrated in the following example, taken from Leconte [1996].

Example 3.7. Consider the following CSP

$$\begin{array}{ll} x_i \in \{2i+1\} & \text{for } i = 0, 1, \ldots, n, \\ x_i \in \{0, 1, \ldots, 2n+2\} & \text{for } i = n+1, n+2, \ldots, 2n, \\ \texttt{alldifferent}(x_0, x_1, \ldots, x_{2n}). & \end{array}$$

In order to make this CSP range consistent, we have to remove the $n+1$ first odd integers from the domains of the n variables whose domain is not yet a singleton. This takes $O(n^2)$ time. □

Observe that this algorithm has an opposite viewpoint from the algorithm for bounds consistency, although it looks similar. Where the algorithm for bounds consistency takes the domains (or intervals) as a starting point, the algorithm for range consistency takes the variables instead. But they both attempt to reach a situation in which the cardinality of a set of variables is equal to the cardinality of the union of the corresponding domains.

3.4.4 Hyper-arc Consistency

A hyper-arc consistency propagation algorithm for the `alldifferent` constraint was proposed by Régin [1994]. The algorithm is based on matching theory (see Section 2.1.1 and Section 3.2.1). We first give a characterization in terms of Hall's Marriage Theorem.

Definition 3.19 (Tight set). *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$. $K \subseteq \{x_1, \ldots, x_n\}$ is a* tight set *if $|K| = |D_K|$.*

Theorem 3.20. *The constraint $\texttt{alldifferent}(x_1, \ldots, x_n)$ is hyper-arc consistent if and only if $|D_i| \geq 1$ $(i = 1, \ldots, n)$ and $D_i \cap D_K = \varnothing$ for each tight set $K \subseteq \{x_1, \ldots x_n\}$ and each $x_i \notin K$.*

Proof. Let K be a tight set and $x_i \notin K$. If $\texttt{alldifferent}(x_1, \ldots, x_n)$ is hyper-arc consistent, it has a solution when $x_i = d$ for all $d \in D_i$, by Definition 3.6. From Theorem 3.4 immediately follows that $D_i \cap D_K = \varnothing$.

Conversely, suppose $\texttt{alldifferent}(x_1, \ldots, x_n)$ is not hyper-arc consistent. Thus, there exist a variable x_i and a value $d_i \in D_i$ for some $i \in \{1, \ldots, n\}$, such that $\texttt{alldifferent}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ has no solution, where $x_j \in D'_j = D_j \setminus \{d_i\}$ for all $j \neq i$. By Theorem 3.4, $|K| > |D'_K|$ for some

$K \subseteq \{x_1, \ldots, x_n\} \setminus \{x_i\}$. If K is a tight set with respect to `alldifferent`$(x_1, \ldots, x_n)$, then $d_i \in D_K$. Otherwise, $|K| > |D_K|$. Then either some domain is empty, or K contains a tight set K', and $D_j \cap D_{K'} \neq \varnothing$ for some $x_j \in K \setminus K'$. □

Following Theorem 3.20, the `alldifferent` constraint can be made hyper-arc consistent by generating all tight sets K and updating $D_i = D_i \setminus D_K$ for all $x_i \notin K$. This approach is similar to the algorithms for achieving bounds consistency and range consistency. For bounds consistency and range consistency we could generate the respective Hall intervals and Hall sets rather easily because we were dealing with intervals containing the domains. In order to generate tight sets similarly we should consider all possible subsets $K \subseteq \{x_1, \ldots, x_n\}$. As the number of subsets is exponential in n, this approach is not practical. A different, more constructive, approach makes use of matching theory to update the domains, and was introduced by Régin [1994].

Theorem 3.21. *Let G be the value graph of a sequence of variables $X = x_1, x_2, \ldots, x_n$ with respective finite domains $D_1, D_2, \ldots, D_n$. The constraint* `alldifferent`$(x_1, \ldots, x_n)$ *is hyper-arc consistent if and only if every edge in G belongs to a matching in G covering X.*

Proof. Immediate from Definition 3.6 and Theorem 3.3. □

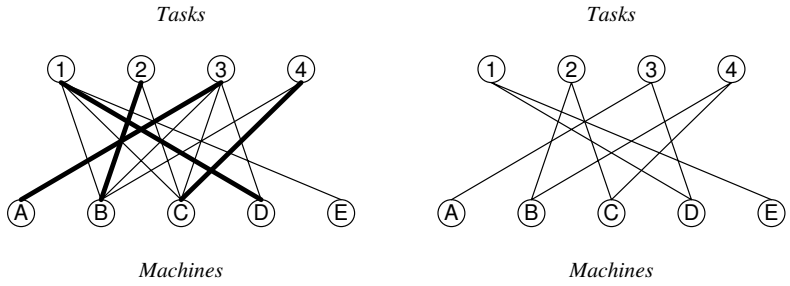
The following Theorem identifies edges that belong to a maximum-size matching. The proof follows from Petersen [1891]; see also Schrijver [2003, Theorem 16.1].

Theorem 3.22. *Let G be a graph and M a maximum-size matching in G. An edge belongs to a maximum-size matching in G if and only if it either belongs to M, or to an even M-alternating path starting at an M-free vertex, or to an M-alternating circuit.*

Proof. Let M be a maximum-size matching in $G = (V, E)$. Suppose edge e belongs to a maximum-size matching N, and $e \notin M$. The graph $G' = (V, M \cup N)$ consists of even paths (possibly of length 0) and circuits with edges alternatingly in M and N. If the paths are not of even length, M or N can be made larger by interchanging edges in M and N along this path (a contradiction because they are of maximum size).

Conversely, let M be a maximum-size matching in G. By interchanging edges in M and not in M along even M-alternating paths starting at an M-free vertex and M-alternating circuits we obtain matchings of maximum size again. □

To establish a connection between Theorem 3.20 and Theorem 3.22, consider a tight set $K \subset \{x_1, \ldots, x_n\}$ of minimum size. The edges between vertices in K and in D_K form M-alternating circuits in the value graph. By applying Theorem 3.22 we remove those edges $x_i d$ with $x_i \notin K$ and $d \in D_K$, i.e. $D_i \cap D_K = \varnothing$. This corresponds to applying Theorem 3.20.



a. A matching covering all tasks. b. The bipartite graph after propagation.

Figure 3.2. The bipartite graph for the task assignment problem of Example 3.8.

Example 3.8. Consider again the task assignment problem of Example 3.1. A solution to this problem is given in Figure 3.2.a, where bold edges denote the corresponding matching M covering the tasks. Machine E is the only M-free vertex. All even M-alternating paths starting from E belong to a solution, by Theorem 3.22. Thus, all edges on the path $\mathsf{E}, 1, \mathsf{D}, 3, \mathsf{A}$ belong to a solution. The only M-alternating circuit in the graph is $2, \mathsf{B}, 4, \mathsf{C}, 2$, and by Theorem 3.22 all edges in this circuit belong to a solution. The other edges, i.e. $(1, \mathsf{B})$, $(1, \mathsf{C})$, $(3, \mathsf{B})$ and $(3, \mathsf{C})$, should be removed to make the `alldifferent` constraint hyper-arc consistent. The result is depicted in Figure 3.2.b. which corresponds to the CSP

$$x_1 \in \{\mathsf{D}, \mathsf{E}\}, x_2 \in \{\mathsf{B}, \mathsf{C}\}, x_3 \in \{\mathsf{A}, \mathsf{D}\}, x_4 \in \{\mathsf{B}, \mathsf{C}\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

□

Using Theorem 3.22, we construct a hyper-arc consistency propagation algorithm. First we compute a maximum-size matching M in the value graph $G = (X \cup D_X, E)$. This can be done in $O(m\sqrt{n})$ time, using the algorithm by Hopcroft and Karp [1973], where $m = \sum_{i=1}^{n} |D_i|$. Next we identify the even M-alternating paths starting at an M-free vertex, and the even M-alternating circuits in the following way.

Define the directed graph $G_M = (X \cup D_X, A)$ with arc set $A = \{(v_1, v_2) \mid (v_1, v_2) \in M\} \cup \{(v_2, v_1) \mid (v_1, v_2) \notin M\}$. In other words, edges in M are being directed from X (the variables) to D_X (the domain values). Edges not in M are being directed reversely. We first compute the strongly connected components in G_M, i.e. maximal subsets of vertices S such that there exists a directed path between every two vertices in S. This can be done in $O(n + m)$ time, following Tarjan [1972]. Arcs between vertices in the same strongly connected component belong to an even M-alternating circuit in G, and are marked as "consistent". Next we search for the arcs that belong to a directed path in G_M, starting at an M-free vertex. This takes $O(m)$ time, using breadth-first search. Arcs belonging to such a path belong to an M-alternating path in G

starting at an M-free vertex, and are marked as "consistent". For all edges $x_i d$ that are not marked "consistent" and do not belong to M, we update $D_i = D_i \setminus \{d\}$. Then, by Theorem 3.22, the corresponding `alldifferent` constraint is hyper-arc consistent.

From the above follows that the `alldifferent` constraint is checked for consistency, i.e. checked to contain a solution, in $O(m\sqrt{n})$ time and made hyper-arc consistent in $O(m)$ time. We could also have applied a general algorithm to achieve hyper-arc consistency on an n-ary constraint. Such algorithm was presented by Mohr and Masini [1988]. For an `alldifferent` constraint on n variables, the time complexity of their general algorithm is $O(\frac{d!}{(d-n)!})$, where d is the maximum domain size. Clearly, the above specialized algorithm is much more efficient.

During the whole solution process of the CSP, constraints other than `alldifferent` might also be used to remove values. In such cases, we must update our `alldifferent` constraint. As indicated by Régin [1994], this can be done incrementally, i.e. we can make use of our current value graph and our current maximum-size matching to compute a new maximum-size matching. This is less time consuming than restarting the algorithm from scratch. Namely, after k modifications we need at most $O(km)$ steps to compute a new maximum-size matching. The same idea has been used by Barták [2001] to make the `alldifferent` constraint dynamic with respect to the addition of variables during the solution process.

3.4.5 Complexity Survey and Discussion

Table 3.2 present a time complexity survey of the algorithms that obtain the four notions of local consistency that have been applied to the `alldifferent` constraint in this section. Here n is again the number of variables inside the `alldifferent` constraint and m the sum of the cardinalities of the domains.

arc consistency	$O(n^2)$	Van Hentenryck [1989]
bounds consistency	$O(n \log n)$	Puget [1998], Mehlhorn and Thiel [2000], Lopez-Ortiz et al. [2003]
	$O(n)$ (special cases)	Mehlhorn and Thiel [2000], Lopez-Ortiz et al. [2003]
range consistency	$O(n^2)$	Leconte [1996]
hyper-arc consistency	$O(m\sqrt{n})$	Régin [1994]

Table 3.2. Survey of time complexity of four local consistency notions.

While increasing the strength of the local consistency notions from bounds consistency to range consistency and hyper-arc consistency, we have seen that

the underlying principle remains the same. It simply boils down to the refinement of the sets to which we apply Hall's Marriage Theorem.

In practice, none of these local consistency notions always outperforms all others. It is strongly problem-dependent which local consistency is suited best. In general, however, bounds consistency is most suitable when domains are always closed intervals. As more holes may occur in domains, hyper-arc consistency becomes more useful. Another observation concerns the implementation of an algorithm that achieves bounds consistency. Although all three variants in Table 3.2 have the same worst-case complexity, the one proposed by Lopez-Ortiz, Quimper, Tromp, and van Beek [2003] appears to be most efficient in practice.

A general comparison of bounds consistency and hyper-arc consistency with respect to the search space has been made by Schulte and Stuckey [2001]. In particular, attention is also paid to the `alldifferent` constraint.

3.5 Variants of the Alldifferent Constraint

This section presents two variants of the `alldifferent` constraint: the symmetric `alldifferent` constraint and the weighted `alldifferent` constraint.

3.5.1 The Symmetric Alldifferent Constraint

A particular case of the `alldifferent` constraint, the symmetric `alldifferent` constraint, was introduced by Régin [1999b]. We assume that the variables and their domain values represent the same set of elements. The symmetric `alldifferent` constraint states that all variables must take different values, and if the variable representing element i is assigned to the value representing element j, then the variable representing the element j must be assigned to the value representing element i. A more formal definition is presented below.

The `symm_alldifferent` constraint is particularly suitable for round-robin tournament problems. In such problems, for example a sports competition, each team has to be matched with another team. Typically, there are many more constraints involved than only the `symm_alldifferent` constraint, which makes the problems often very difficult to solve. The propagation of the `alldifferent` constraint and the `symm_alldifferent` constraint has been analyzed for practical problem instances by Henz, Müller, and Thiel [2004]. They show that constraint programming, using the `symm_alldifferent` constraint, outperforms operations research approaches with several orders of magnitude.

Definition 3.23 (Symmetric alldifferent constraint). *Let $x_1, x_2, \ldots, x_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n \subseteq \{1, 2, \ldots, n\}$. Then*

$$\texttt{symm\_alldifferent}(x_1, \ldots, x_n) = \\ \{(d_1, \ldots, d_n) \mid d_i \in D_i, d_i \neq d_j \text{ for } i \neq j, d_i = j \Leftrightarrow d_j = i \text{ for } i \neq j\}\,.$$

In a CSP, the `symm_alldifferent` constraint can also be expressed as an `alldifferent` constraint together with one or more constraints that preserve the symmetry. Another representation can be made that uses the so-called `cycle` constraint, where each cycle must contain two vertices; see Beldiceanu [2000]. In that case, a cycle on two vertices x and y indicates that x is assigned to y and vice versa. However, the `symm_alldifferent` constraint captures more global information than the common `alldifferent` constraint together with additional constraints. Hence the `symm_alldifferent` constraint can be used to obtain a stronger propagation algorithm. The following example, taken from Régin [1999b] shows exactly this.

Example 3.9. Consider a set of three people that have to be grouped in pairs. Each person can only be paired to one other person. This problem can be represented as a CSP by introducing a set of people $S = \{p_1, p_2, p_3\}$ that are pairwise compatible. These people are represented both by a set of variables x_1, x_2 and x_3 and by a set of values v_1, v_2 and v_3, where x_i and v_i represent p_i. Then the CSP

$$x_1 \in \{v_2, v_3\}, x_2 \in \{v_1, v_3\}, x_3 \in \{v_1, v_2\}, \\ x_1 = v_2 \Leftrightarrow x_2 = v_1, \\ x_1 = v_3 \Leftrightarrow x_3 = v_1, \\ x_2 = v_3 \Leftrightarrow x_3 = v_2, \\ \texttt{alldifferent}(x_1, x_2, x_3)$$

is hyper-arc consistent. However, the following CSP

$$x_1 \in \{v_2, v_3\}, x_2 \in \{v_1, v_3\}, x_3 \in \{v_1, v_2\}, \\ \texttt{symm\_alldifferent}(x_1, x_2, x_3)$$

is inconsistent. Indeed, there exists no solution to this problem, as the number of variables is odd. □

Suppose there exists a value $j \in D_i$, while $i \neq D_j$. Then we can immediately remove value j from D_i. Hence we assume that such situations do not occur in the following.

Similar to the common `alldifferent` constraint, the `symm_alldifferent` constraint can be expressed by a graph. Given a constraint $\texttt{symm\_alldifferent}(x_1, \ldots, x_n)$, construct the graph $G_{\text{symm}} = (X, E)$, with vertex set $X = \{x_1, x_2, \ldots, x_n\}$ and edge set $E = \{x_i x_j \mid x_i \in D_j, x_j \in D_i, i < j\}$. Note that we identify each variable x_i with value i for $i = 1, \ldots, n$. We denote the number of edges in G_{symm} by m, i.e. $m = \left(\sum_{i=1}^{n} |D_i|\right)/2$. An illustration of G_{symm} is given in the next example.

Example 3.10. Consider the following CSP

$x_{\mathsf{a}} \in \{\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}$, $x_{\mathsf{b}} \in \{\mathsf{a},\mathsf{c},\mathsf{d},\mathsf{e}\}$, $x_{\mathsf{c}} \in \{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{e}\}$, $x_{\mathsf{d}} \in \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{e}\}$,
$x_{\mathsf{e}} \in \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{i},\mathsf{j}\}$, $x_{\mathsf{f}} \in \{\mathsf{g},\mathsf{h}\}$, $x_{\mathsf{g}} \in \{\mathsf{f},\mathsf{h}\}$, $x_{\mathsf{h}} \in \{\mathsf{f},\mathsf{g},\mathsf{i},\mathsf{j}\}$,
$x_{\mathsf{i}} \in \{\mathsf{e},\mathsf{h},\mathsf{j}\}$, $x_{\mathsf{j}} \in \{\mathsf{e},\mathsf{h},\mathsf{i}\}$,
$\texttt{symm\_alldifferent}(x_{\mathsf{a}}, x_{\mathsf{b}}, \ldots, x_{\mathsf{j}})$.

In Figure 3.3.a the corresponding graph G_{symm} is depicted, where $x_{\mathsf{a}}, x_{\mathsf{b}}, \ldots, x_{\mathsf{j}}$ are abbreviated to $\mathsf{a}, \mathsf{b}, \ldots, \mathsf{j}$. □

Similar to Theorem 3.3, we have the following result.

Theorem 3.24. *Let $x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, D_2, \ldots, D_n$. Then*

$$(d_1, \ldots, d_n) \in \texttt{symm\_alldifferent}(x_1, \ldots, x_n)$$

if and only if $M = \{x_i d_i \mid i < d_i\}$ is a matching in G.

Proof. An edge $x_i x_j$ in G_{symm} corresponds to the assignments $x_i = d_i$ and $x_{d_i} = i$, which are equivalent with $x_j = d_j$ and $x_{d_j} = j$, because $d_i = j$ and $d_j = i$. As no edges in a matching share a vertex, all endpoints are different. Finally, as M covers X, to all variables a value is assigned. □

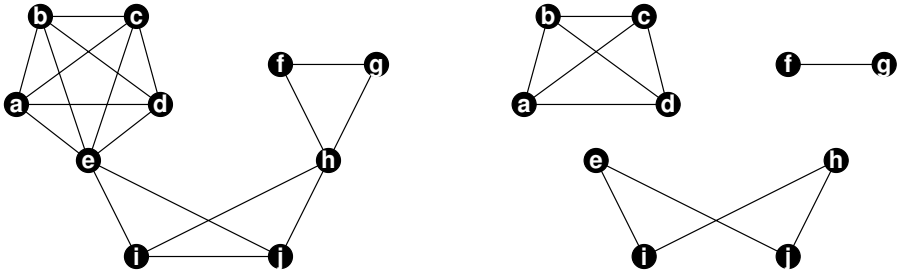
We use Theorem 3.24 to make the `symm_alldifferent` constraint hyper-arc consistent.

Theorem 3.25. *The constraint* $\texttt{symm\_alldifferent}(x_1, \ldots, x_n)$ *is hyper-arc consistent if and only if every edge in the corresponding graph G_{symm} belongs to a matching that covers $x_1, \ldots, x_n$.*

Proof. By Definition 3.6 (hyper-arc consistency) and application of Theorem 3.24. □

Example 3.11. Consider again the CSP of Example 3.10. Figure 3.3.b shows the graph of Figure 3.3.a after making the CSP hyper-arc consistent. Consider for example the subgraph induced by vertices f, g and h. Obviously vertices f and g have to be paired in order to satisfy the `symm_alldifferent` constraint. Hence, vertex h cannot be paired to f nor g, and the corresponding edges can be removed. The same holds for the subgraph induced by a, b, c, d and e, where the vertices a, b, c and d must form pairs. □

A matching in G_{symm} that covers $x_1, \ldots, x_n$ is also a maximum-size matching. From Theorem 3.22 we already know how to identify edges that belong to a maximum-size matching. Namely, given an arbitrary maximum-size matching M, they belong either to M, or to an even M-alternating path starting at a free vertex, or to an even M-alternating circuit. However, in the current case, G_{symm} does not need to be bipartite, so we cannot blindly apply the machinery from Section 3.4.4 to achieve hyper-arc consistency for the `symm_alldifferent` constraint.



a. The graph G_{symm}. b. The graph G_{symm} after propagation.

Figure 3.3. Propagation of the `symm_alldifferent` constraint of Example 3.10.

The following algorithm to achieve hyper-arc consistency was proposed by Régin [1999b]. First, we compute a maximum-size matching M in the (possibly non-bipartite) graph G_{symm}. This can be done in $O(m\sqrt{n})$ time by applying the algorithm by [Micali and Vazirani, 1980]. If $|M| < n/2$, there is no solution. Otherwise, we need to detect all edges that can never belong to a maximum-size matching. Since there are no M-free vertices, we only need to check whether an edge that does not belong to M is part of an even M-alternating circuit. This can be done as follows. If for an edge $uv \in M$ we find an M-alternating path $u, \ldots, w, v$ (with $u \neq w$), we know that the edge wv is on an even M-alternating circuit. Moreover, we can compute all possible M-alternating paths from u to v, that avoid edge uv. All edges wv that are not on such a path cannot belong to an even M-alternating circuit. Namely, all M-alternating circuits through wv should also contain the matching edge uv. Hence, by Theorem 3.22 and Theorem 3.24 we can delete such edges wv. This procedure should be repeated for all vertices in G_{symm}.

The algorithm for computing the M-alternating paths can be implemented to run in $O(m)$ time, using results of Edmonds [1965] to take care of the non-bipartiteness, and Tarjan [1972] for the data structure, as was observed by Régin [1999b]. Hence the total time complexity of the algorithm achieving hyper-arc consistency for the `symm_alldifferent` constraint is $O(nm)$.

Note that the above algorithm is not incremental, and may not be effective in all practical cases. For this reason, also an incremental algorithm was proposed by Régin [1999b], that does not ensure hyper-arc consistency, however. The algorithm computes once a maximum-size matching, and each incremental step has a time complexity of $O(m)$.

3.5.2 The Weighted Alldifferent Constraint

In this section we assume that for all variables in the `alldifferent` constraint each variable-value pair induces a fixed cost. Eventually (the problem we want to solve involves also other constraints), the goal is to find a solution

with minimum[3] total cost. In the literature, this combination is known as the constraint `MinWeightAllDiff` [Caseau and Laburthe, 1997b], or `IlcAllDiffCost` [Focacci, Lodi, and Milano, 1999b]. This section shows how to exploit the `alldifferent` constraint and the minimization problem together as an "optimization constraint". First we give a definition of the weighted `alldifferent` constraint, and then we show how we to make it hyper-arc consistent. It should be noted that the weighted `alldifferent` constraint is a special case of the weighted global cardinality constraint, which will be introduced in Section 4.5.

Definition 3.26 (Weighted alldifferent constraint). *Let $x_1, x_2, \ldots, x_n, z$ be variables with respective finite domains $D_1, D_2, \ldots, D_n, D_z$, and let $w_{ij} \in \mathbb{Q}_+$ for $i = 1, \ldots, n$ and all $j \in \bigcup_{i=1,\ldots,n} D_i$ be constants. Then*

$$\texttt{minweight\_alldifferent}(x_1, \ldots, x_n, z, w) = \left\{(d_1, \ldots, d_n, \tilde{d}) \mid d_i \in D_i, \tilde{d} \in D_z, d_i \neq d_j \text{ for } i \neq j, \textstyle\sum_{i,d_i=j} w_{ij} \leq \tilde{d}\right\}.$$

Note that for a pair (x_i, j) with $j \notin D_i$, we may define $w_{ij} = \infty$. The variable z in Definition 3.26 serves as a "cost" variable, which is minimized during the solution process. This means that admissible tuples are those instantiations of variables with total weight not more than the currently best found solution, represented by $\max D_z$. At the same time, $\min D_z$ should not be less than the currently lowest possible total weight.

In order to make the `minweight_alldifferent` constraint hyper-arc consistent, we introduce the directed graph $G_{\text{minweight}} = (V, A)$ with

$$V = \{s, t\} \cup X \cup D_X \quad \text{and} \quad A = A_s \cup A_X \cup A_t$$

where $X = \{x_1, x_2, \ldots, x_n\}$ and

$$\begin{aligned} A_s &= \{(s, x_i) \mid i = 1, 2, \ldots, n\}, \\ A_X &= \{(x_i, d) \mid d \in D_i\}, \\ A_t &= \{(d, t) \mid d \in \textstyle\bigcup_{i=1}^{n} D_i\}. \end{aligned}$$

To each arc $a \in A$, we assign a capacity $c(a) = 1$ and a weight $w(a)$. If $a = x_i j \in A_X$, then $w(a) = w_{ij}$. If $a \in A_s \cup A_t$ then $w(a) = 0$.

Theorem 3.27. *The constraint* `minweight_alldifferent`$(x_1, \ldots, x_n, z, w)$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ there exists a feasible $s-t$ flow f in $G_{\text{minweight}}$ with $f(a) = 1$, $\text{value}(f) = n$ and $\text{weight}(f) \leq \max D_z$, and
ii) $\min D_z \geq \text{weight}(f)$ for a minimum-weight $s-t$ flow f of value n in $G_{\text{minweight}}$.

[3] A maximization problem can be reformulated as a minimization problem by negating the objective function.

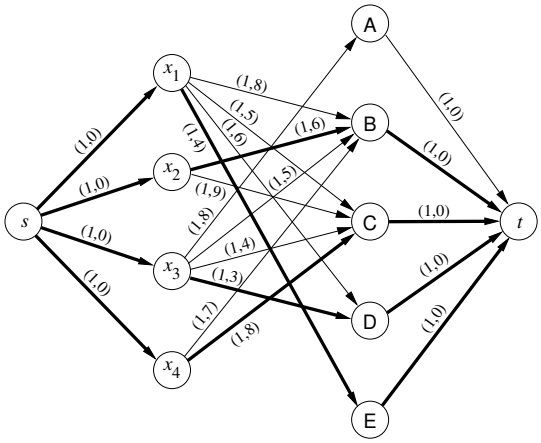


Figure 3.4. Graph $G_{\text{minweight}}$ for the `minweight_alldifferent` constraint of Example 3.12. For each arc a, $(c(a), w(a))$ is given. Bold arcs indicate a minimum-weight $s - t$ flow with weight 21.

Proof. We may assume that every feasible $s - t$ flow f of value n is integer, because c is integer. In fact, we may assume that f is $\{0, 1\}$-valued, because $c(a) = 1$ for each arc $a \in A$. As value$(f) = n$, f uses exactly n arcs in A_X. An arc $a = (x_i, j) \in A_X$ with $f(a) = 1$ corresponds to assigning $x_i = j$. Because $c(a) = 1$ for all $a \in A_t$, each value is used at most once, which enforces the `alldifferent` constraint. Hence, if weight$(f) \leq \max D_z$, the corresponding assignment is a solution to the `minweight_alldifferent` constraint. □

Example 3.12. Consider again the task assignment problem of Example 3.1. Suppose in addition that each task-machine combination induces a cost, as indicated in Table 3.3 below. The goal is to find an assignment with minimum cost. Denote the cost of assigning task i to machine j as w_{ij}. For the cost

	Machine				
Task	A	B	C	D	E
1	∞	8	5	6	4
2	∞	6	9	∞	∞
3	8	5	4	3	∞
4	∞	7	8	∞	∞

Table 3.3. Cost of task - machine combinations.

variable z we initially set $\max D_z = 33$, being the sum of the maximum assignment cost for each variable. Then we model the problem as the CSP

$z \in \{0, 1, \ldots, 33\}$,
$x_1 \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}, x_2 \in \{\mathsf{B}, \mathsf{C}\}, x_3 \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}, x_4 \in \{\mathsf{B}, \mathsf{C}\}$,
`minweight_alldifferent`$(x_1, x_2, x_3, x_4, z, w)$,
`minimize` z.

The corresponding graph $G_{\text{minweight}}$ is shown in Figure 3.4. The bold arcs in the graph denote a minimum-weight $s-t$ flow with weight 21. It corresponds to a solution to the COP, i.e. $x_1 = \mathsf{E}, x_2 = \mathsf{B}, x_3 = \mathsf{D}, x_4 = \mathsf{C}$ and $z = 21$. □

Following Theorem 3.27, we can make the `minweight_alldifferent` constraint hyper-arc consistent by the following procedure:

> For all arcs $a = x_i j \in A_X$, compute a minimum-weight $s-t$ flow f in $G_{\text{minweight}}$ with $f(a) = 1$. If $\text{value}(f) < n$ or $\text{weight}(f) > \max D_z$, update $D_i = D_i \setminus \{j\}$.
> For a minimum-weight $s-t$ flow f in $G_{\text{minweight}}$ with $\text{value}(f) = n$, update $\min D_z = \text{weight}(f)$ if $\min D_z < \text{weight}(f)$.

The drawback of the above procedure is that we have to compute a minimum-weight $s-t$ flow for each arc in A_X. It is more efficient however, to use the residual graph $(G_{\text{minweight}})_f$ and apply Theorem 2.2 as follows.

We first compute a minimum-weight flow f in $G_{\text{minweight}}$. For this we use the algorithm presented in Section 2.1.1. We need to compute n shortest $s-t$ paths in the residual graph, each of which takes $O(m + d \log d)$ time where $m = \sum_{i=1}^{n} |D_i|$ and $d = |\bigcup_{i=1}^{n} D_i|$. Thus the minimum-weight flow is computed in $O(n(m + d \log d))$ time. If $\text{weight}(f) > \max D_z$, the `minweight_alldifferent` constraint is inconsistent.

If $\text{weight}(f) \leq \max D_z$, we consider all arcs $a = x_i j \in A_X$ with $f(a) = 0$ (see Theorem 2.2). We compute a minimum-weight $j - x_i$ path P in the residual graph $(G_{\text{minweight}})_f$, if it exists. If P does not exist, we update $D_i = D_i \setminus \{j\}$. Otherwise, P, a, j forms a circuit C of minimum weight, containing a. If $\text{weight}(f) + \text{weight}(C) > \max D_z$ we update $D_i = D_i \setminus \{j\}$. Finally, we update $\min D_z = \text{weight}(f)$ if $\min D_z < \text{weight}(f)$. Then, by Theorem 3.27 and Theorem 2.2 the `minweight_alldifferent` constraint is hyper-arc consistent. These last steps involve $m - n$ shortest path computations, each taking $O(m + d \log d)$ time. Hence the total time complexity for making the `minweight_alldifferent` constraint hyper-arc consistent is $O(m(m + d \log d))$. In fact, this can be improved to $O(n(m + d \log d))$, as was shown by Régin [1999a, 2002] and Sellmann [2002].

Other work concerning the `alldifferent` constraint in conjunction with an objective function has been done by Lodi, Milano, and Rousseau [2003]. In that work, the so-called additive bounding procedure [Fischetti and Toth, 1989] and limited discrepancy search are exploited in presence of an `alldifferent` constraint; see also Section 6.3.

3.6 The Alldifferent Polytope

In this section we consider the description of the `alldifferent` constraint in integer linear programming. In this section we suppose that an integer linear programming model consists of integer variables, a set of linear constraints (inequalities) and a linear objective function to be optimized, and can be written as

$$\max \{c^{\mathsf{T}} x \mid Ax \leq b, x \text{ integral}\} \tag{3.4}$$

for rational matrix A, rational vectors b and c and variable vector x of appropriate size. The continuous relaxation of (3.4) is

$$\max \{c^{\mathsf{T}} x \mid Ax \leq b\}. \tag{3.5}$$

Note that (3.5) can be rewritten into the standard form (2.2) of a linear program.

A problem can usually be modelled as an integer linear program in more than one way. Each such model leads to a (possibly different) continuous relaxation. Ideally, one seeks for a model for which the relaxation is precisely the convex hull of all integer solutions. Namely, for such models the continuous relaxation has an integral, and hence optimal, solution. Formally, relaxation (3.5) defines a so-called *polytope*, i.e. the convex hull of a finite number of vectors, where the vectors are defined by the linear constraints $Ax \leq b$ of the problem. Our goal is to find a tight polytope, i.e. a polytope that describes the convex hull of all integer solutions to the problem, for the `alldifferent` constraint.

Suppose the problem at hand contains the structure

$$\begin{array}{l} \texttt{alldifferent}(x_1, x_2, \ldots, x_n) \\ x_1, x_2, \ldots, x_n \in \{d_1, d_2, \ldots, d_n\} \end{array} \tag{3.6}$$

for which we would like to find a tight polytope. We assume that the domain values $d_1, d_2, \ldots, d_n$ are numerical values with $d_1 < d_2 < \cdots < d_n$. A convex hull representation of (3.6) was given by Williams and Yan [2001]. We follow here the description and the proof of Hooker [2000, p. 232–233]. The idea is that the sum of any k variables must be at least $d_1 + \cdots + d_k$.

Theorem 3.28. *Let $d_1, d_2, \ldots, d_n$ be any sequence of numerical values with $d_1 < d_2 < \cdots < d_n$. Then the convex hull of all vectors x that satisfy* `alldifferent`$(x_1, x_2, \ldots, x_n)$ *where $x_i \in \{d_1, d_2, \ldots, d_n\}$ is described by*

$$\sum_{j=1}^{n} y_j = \sum_{j=1}^{n} d_j, \tag{3.7}$$

$$\sum_{j \in J} y_j \geq \sum_{j=1}^{|J|} d_j, \text{ for all } J \subset \{1, \ldots, n\} \text{ with } |J| < n. \tag{3.8}$$

for $y \in \mathbb{R}^n$.

Proof. It suffices to show that any valid inequality $a^{\mathsf{T}}y \geq \alpha$ can be obtained as a linear combination of (3.7) and (3.8) in which the coefficients of (3.8) are nonnegative. Assume, without loss of generality, that $a_{i_1} \geq a_{i_2} \geq \cdots \geq a_{i_n}$ for some permutation $i_1, \ldots, i_n$ of the indices. Then because $a^{\mathsf{T}}y \geq \alpha$ is valid, one can set $y_k = d_j$ if $i_j = k$, so that

$$\sum_{j=1}^{n} a_{i_j} d_j \geq \alpha. \tag{3.9}$$

From (3.7) and (3.8),

$$\sum_{j=1}^{n} y_{i_j} = d_1 + \cdots + d_n, \tag{3.10}$$

$$\sum_{j=1}^{k} y_{i_j} \geq d_1 + \cdots + d_k,\ k = 1, \ldots, n-1. \tag{3.11}$$

Consider a linear combination in which each inequality of 3.11 has coefficient $a_{i_k} - a_{i_{k+1}}$, and (3.10) has coefficient a_{i_n}. The result is

$$\sum_{j=1}^{n} a_{ij} y_{i_j} \geq \sum_{j=1}^{n} a_{i_j} d_j \overset{(3.9)}{\geq} \alpha$$

and the theorem follows. □

Example 3.13. The convex hull of all vectors x that satisfy

$$\texttt{alldifferent}(x_1, x_2, x_3)$$
$$x_1, x_2, x_3 \in \{7, 11, 13\}$$

can be given by

$$\begin{aligned} &y_1 + y_2 + y_3 = 31,\\ &y_1 + y_2 \geq 18,\\ &y_1 + y_3 \geq 18,\\ &y_2 + y_3 \geq 18,\\ &y_j \geq 7, \text{ for } j = 1, 2, 3, \end{aligned}$$

and is depicted in Figure 3.5. □

Unfortunately, the number of inequalities to describe the convex hull grows exponentially with the number of variables. In practice it may therefore be profitable to generate these inequalities only when they are violated by the current relaxation.

Further work on the description of the `alldifferent` constraint in integer linear programming has been done by Lee [2002]. That work proposes a representation that uses a binary encoding of the solution set. Further, Appa, Magos, and Mourtos [2004] present linear programming relaxations based upon multiple `alldifferent` constraints.

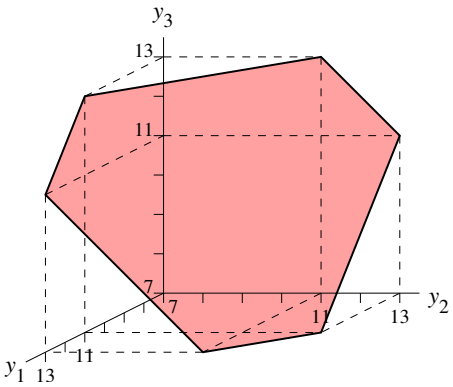


Figure 3.5. The polytope of Example 3.13.

3.7 Conclusion

We have presented a survey of the most important results over the years regarding the `alldifferent` constraint. To this end, we have first introduced the underlying combinatorial concepts on which the results are based, i.e. matchings, flows and Hall's Theorem. Using these concepts we have provided a systematic presentation of the different notions of local consistency and their corresponding propagation algorithms that have been applied to the `alldifferent` constraint.

An important observation is the following. To make constraint programming applicable to practical problems one needs propagation algorithms that are both effective and efficient. The most powerful propagation algorithm for the `alldifferent` constraint, i.e. the one obtaining hyper-arc consistency, is indeed very efficient. The reason is that we can apply matching theory from operations research. Also for the symmetric `alldifferent` constraint and the weighted `alldifferent` constraint effective and efficient propagation algorithms exist, again based on techniques from operations research.

These results show that the application of operations research techniques in constraint propagation algorithms can be very beneficial.

Chapter 4

Soft Global Constraints

In case a CSP is over-constrained, it is natural to allow some constraints, called soft constraints*, to be violated. We propose a generic method to soften global constraints that can be represented by a flow in a graph. Such constraints are softened by inserting* violation arcs *to the graph. Then we compute a minimum-weight flow in the extended graph to measure the violation. We present efficient propagation algorithms, based on different violation measures, achieving hyper-arc consistency for the* `alldifferent` *constraint, the* global cardinality *constraint, the* `regular` *constraint and the* `same` *constraint.*

4.1 Introduction

Many real-life problems are over-constrained. In personnel rostering problems for example, people often have conflicting preferences. To such problems there does not exist a feasible solution that respects all preferences. However, we still want to find *some* solution, preferably one that minimizes the total number of conflicts. In case of the personnel rostering example, we may want to construct a roster in which the number of respected preferences is equally spread among the employees.

In constraint programming, we seek for an (optimal) feasible solution to a given problem. Hence, we cannot apply constraint programming directly to over-constrained problems, because it finds no solution. As a remedy there have been proposed several methods. Most of these methods introduce so-called *soft constraints* that are allowed to be violated. Constraints that are not allowed to be violated are called *hard constraints*. Most methods then try to find a solution that minimizes the number of violated constraints, or some other measure of constraint violation.

Global constraints are often key elements in successfully modelling and solving real-life applications with constraint programming. For many *soft* global constraints, however, no efficient propagation algorithms were available, up to very recently. Moreover, it was stated that

> *"Active solving of global constraints is only applicable on the assumption that the constraint must be satisfied in any solution. Soft constraints may be violated in an admissible solution, and cannot therefore be handled by the usual techniques for global constraints."*

(quoted from Wallace, Caseau, and Puget [2003, p. 335]). In this chapter we show that "usual techniques for global constraints" *can* be used to handle

soft constraints. In particular, we apply techniques that were used earlier to handle the weighted `alldifferent` constraint.

We distinguish two main objectives with respect to soft global constraints: *useful violation measures* and *efficient propagation algorithms*. Both issues will be addressed in this chapter, depending heavily on a technique from operations research: flow theory.

In many cases we can represent a solution to a global constraint as a property in some graph representation of the constraint. For example, a solution to the `alldifferent` constraint corresponds to a matching in the associated value graph, as we have seen in Section 3.2.1. There exists a large class of such global constraints, see for example Beldiceanu [2000] for a collection. In this chapter, we focus on global constraints for which a solution can be represent by a flow in a graph.

Our method adds *violation arcs* to the graph representation of a global constraint. To these arcs we assign a cost, corresponding to some violation measure of the constraint. Each tuple in the constraint has an associated cost of violation. If the tuple satisfies the constraint, the corresponding flow does not use any violation arc, and the cost is 0. If the tuple does not satisfy the constraint, the corresponding flow must use violation arcs, whose costs represent the cost of violation of this tuple.

This approach allows us to define and implement useful violation measures for soft global constraints. Moreover, we present an efficient generic propagation algorithm for soft global constraints, making use of flow theory. We apply our method to several global constraints that are well-known to the constraint programming community: the `alldifferent` constraint, the global cardinality constraint, the `regular` constraint and the `same` constraint, which will be defined when considered in this chapter. To each of these global constraints we apply several violation measures, some of which are newly introduced.

This chapter is organized as follows. In Section 4.2 we give an overview of related literature. Then our method to soften global constraints is presented in Section 4.3. We first discuss the general concepts of constraint softening and violation measures. Then we describe the addition of violation arcs to the graph representation and present the generic hyper-arc consistency propagation algorithm.

In Section 4.4 we apply our method to the four global constraints mentioned above: the `alldifferent` constraint, the global cardinality constraint, the `same` constraint and the `regular` constraint. For each constraint we introduce useful violation measures and the corresponding graph representations. We also analyze the corresponding propagation algorithms to achieve hyper-arc consistency.

In Section 4.8 we propose to use soft global constraints to aggregate the costs of violation of different soft constraints. Finally, in Section 4.9 a conclusion is given.

4.2 Related Literature

The best-known framework to handle soft constraints is the *Partial-CSP* framework by Freuder and Wallace [1992]. This framework includes the *Max-CSP* framework that tries to maximize the number of satisfied constraints. Since in this framework all constraints are either violated or satisfied, the objective is equivalent to minimizing the number of violated constraints. It has been extended to the *Weighted-CSP* framework by Larrosa [2002] and Larrosa and Schiex [2003], associating a degree of violation (not just a boolean value) to each constraint and minimizing the sum of all weighted violations. The *Possibilistic-CSP* framework by Schiex [1992] associates a preference to each constraint (a real value between 0 and 1) representing its importance. The objective of the framework is the hierarchical satisfaction of the most important constraints, i.e. the minimization of the highest preference level for a violated constraint. The *Fuzzy-CSP* framework by Dubois, Fargier, and Prade [1993], Fargier, Lang, and Schiex [1993] and Ruttkay [1994] is somewhat similar to the Possibilistic-CSP but here a preference is associated to each tuple of each constraint. A preference value of 0 means the constraint is highly violated and 1 stands for satisfaction. The objective is the maximization of the smallest preference value induced by a variable assignment. The last two frameworks are different from the previous ones since the aggregation operator is a min/max function instead of addition. Max-CSPs are typically encoded and solved with one of two generic paradigms: valued-CSPs [Schiex, Fargier, and Verfaillie, 1995] and semi-rings [Bistarelli, Montanari, and Rossi, 1997].

Another approach to model and solve over-constrained problems was proposed by Régin, Petit, Bessière, and Puget [2000] and refined by Beldiceanu and Petit [2004]. The idea is to identify with each soft constraint S a "cost" variable z, and replace the constraint S by the disjunction

$$(S \wedge (z = 0)) \ \vee \ (\overline{S} \wedge (z > 0))$$

where $\overline{S}$ is a constraint of the type $z = \mu(S)$ for some violation measure $\mu(S)$ depending on S. The newly defined problem is not over-constrained anymore. If we ask to minimize the (weighted) sum of violation costs, we can solve the problem with a traditional constraint programming solver.

This approach also allows us to design specialized filtering algorithms for soft global constraints. Namely, if we treat the soft constraints as "optimization constraints", we can apply cost-based propagation algorithms. We have already seen an example of a cost-based propagation algorithm for the weighted `alldifferent` constraint in Section 3.5.2. Constraint propagation algorithms for soft constraints based on this method were given by Petit, Régin, and Bessière [2001] and van Hoeve [2004].

Another advantage of this framework is the applicability of "meta-constraints", as proposed by Petit, Régin, and Bessière [2000]. The idea behind this technique is to aggregate the cost variables by imposing a constraint on

them, called a *meta-constraint*. By correctly constraining these variables it is possible to replicate the previous frameworks for soft constraints, and even to extend the modeling capability to capture other types of violation measures.

Because of the above advantages, most importantly because it allows the design of specific cost-based constraint propagation algorithms for soft global constraints, we follow the scheme proposed by Régin, Petit, Bessière, and Puget [2000] to handle over-constrained CSPs.

4.3 Outline of Method

In this section we first define how we soften global constraints, and define several violation measures. Then we present a generic constraint propagation algorithm for a class of soft global constraints; those that can be represented by a flow in a graph.

4.3.1 Constraint Softening and Violation Measures

As stated above, the idea of the scheme by Régin, Petit, Bessière, and Puget [2000] is as follows. To each soft constraint we associate a violation measure and a cost variable that measures this violation. Then we transform the CSP into a constraint optimization problem (COP), where all constraints are hard, and the (weighted) sum of cost variables is minimized. If we impose an upper bound on the cost variable of each soft constraint, we can remove all domain values for which no solution exists with a violation cost below this upper bound. The method is illustrated by the following example.

Example 4.1. Consider the over-constrained CSP

$$x_1 \in \{a, b\}, x_2 \in \{a, b\}, x_3 \in \{a, b\}, x_4 \in \{b, c\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

We soften the `alldifferent` constraint as follows. Let μ denote the violation measure that counts the number of pairs (x_i, x_j) with $x_i = x_j$ for all distinct i, j. We introduce a cost variable z that represents μ. The domain of z is initially set to $\{0, \ldots, 6\}$, because at most 6 pairs of variables can be equal. Let the soft `alldifferent` be denoted by $\texttt{soft\_alldifferent}(x_1, x_2, x_3, x_4, z, \mu)$. Then we transform the CSP into the following COP:

$$z \in \{0, \ldots, 6\},$$
$$x_1 \in \{a, b\}, x_2 \in \{a, b\}, x_3 \in \{a, b\}, x_4 \in \{b, c\},$$
$$\texttt{soft\_alldifferent}(x_1, x_2, x_3, x_4, z, \mu),$$
$$\texttt{minimize}\ z.$$

This COP is not hyper-arc consistent, as there is no solution with $z < 1$. If we remove 0 from D_z, the COP is hyper-arc consistent, because there are at most 6 simultaneously violated dis-equalities.

Suppose now that during the search for a solution, we have found the tuple $(x_1, x_2, x_3, x_4, z) = (a, a, b, c, 1)$, that has one violated dis-equality. Then $z \in \{1\}$ in the remaining search. As the assignment $x_4 = b$ always leads to a solution with $z \geq 2$, b can be removed from D_4. The resulting COP is hyper-arc consistent again.

One should take into account that a simplified CSP is considered in this example. In general, a CSP can consist of many more hard and soft constraints, and also more cost-variables that together with z form an objective function to be minimized. □

Let $X = x_1, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, \ldots, D_n$. Consider a CSP that contains soft constraints, each on a subsequence of X. For each soft constraint $C(x_1, \ldots, x_k)$ we define a *violation measure* $\mu : D_1 \times \cdots \times D_k \to \mathbb{Q}$ and a "cost" variable z that represents the measure of violation of C.

We first give a general definition of *constraint softening*.

Definition 4.1 (Constraint softening). *Let $x_1, \ldots, x_n, z$ be variables with respective finite domains $D_1, \ldots, D_n, D_z$. Let $C(x_1, \ldots, x_n)$ be a constraint with a violation measure $\mu(x_1, \ldots, x_n)$. Then*

$$\texttt{soft\_}C(x_1, \ldots, x_n, z, \mu) = \left\{(d_1, \ldots, d_n, \tilde{d}) \mid d_i \in D_i, \tilde{d} \in D_z, \mu(d_1, \ldots, d_n) \leq \tilde{d}\right\}$$

is the soft version of C with respect to μ.

The cost variable z is minimized (possibly together with other cost variables) during the solution process. Thus, $\max D_z$ represents the maximum value of violation that is allowed, and $\min D_z$ represents the lowest possible value of violation, given the current state of the solution process.

There may be several natural ways to evaluate the degree to which a global constraint is violated and these are usually not equivalent. Two general measures are the *variable-based* violation measure and the *decomposition-based* violation measure.

Definition 4.2 (Variable-based violation measure). *Let C be a constraint on the variables $x_1, \ldots, x_n$ and let $d_1, \ldots, d_n$ be an instantiation of variables such that $d_i \in D_i$ for $i = 1, \ldots, n$. The* variable-based violation measure μ_{var} *of C is the minimum number of variables that need to change their value in order to satisfy C.*

For the decomposition-based violation measure we make use of the binary decomposition of a constraint; see Definition 3.11.

Definition 4.3 (Decomposition-based violation measure). *Let C be a constraint on the variables $x_1, \ldots, x_n$ for which a binary decomposition C_{dec} exists and let $d_1, \ldots, d_n$ be an instantiation of variables such that $d_i \in D_i$ for $i = 1, \ldots, n$. The* decomposition-based violation measure μ_{dec} *of C is the number of violated constraints in C_{dec}.*

Alternative measures exist for specific constraints. For the soft global cardinality constraint and the soft `regular` constraint, we introduce new violation measures, that are likely to be more effective in practical applications.

After we have assigned a violation measure to each soft constraint, the method proceeds as follows. Consider a CSP of the form $P = (X, D, C_{\text{hard}} \cup C_{\text{soft}})$, where C_{hard} and C_{soft} denote the set of hard and soft constraints of P, respectively. We soften each constraint $C_i \in C_{\text{soft}}$ using the violation measure it has been assigned, and a cost variable z_i with domain D_{z_i} $(i = 1, \ldots, |C_{\text{soft}}|)$ that represents this measure. We further define a function

$$h : D_{z_1} \times \cdots \times D_{z_{|C_{\text{soft}}|}} \to \mathbb{Q},$$

to measure the aggregated cost of violation for all soft constraints. We introduce a variable z_{agg} to represent the value of h. Then we transform P into the COP $\tilde{P} = (\tilde{X}, \tilde{D}, \tilde{C})$ where $\tilde{X}$ is the sequence of variables containing $X, z_1, \ldots, z_{|C_{\text{soft}}|}, z_{\text{agg}}$, $\tilde{D}$ contains their corresponding domains, and $\tilde{C}$ is the constraint set containing C_{hard} and the soft version of the constraints in C_{soft}, together with the "constraint" `minimize` z_{agg}.

4.3.2 Propagation of Soft Constraints

Let $x_1, \ldots, x_n$ be variables with respective finite domains $D_1, \ldots, D_n$ and let $C(x_1, \ldots, x_n)$ be a constraint. Further, let $\phi \in \mathbb{R}_+$. We assume that C can be represented by a directed graph $G = (V, A)$ with capacity function $c : A \to \mathbb{N}$ that has the following properties:

- a pair (x_i, j) is represented by at least one arc $a \in A$ for $i = 1, \ldots, n$ and all $j \in D_i$, and $c(a) = 1$,
- a tuple $(d_1, \ldots, d_n) \in C$ is represented by a feasible flow f of value ϕ in G such that $f(a) = 1$ for an arc a representing the pair (x_i, d_i) for $i = 1, \ldots, n$, and $f(a) = 0$ for all arcs a representing the pair (x_i, d) with $d \neq d_i$ for $i = 1, \ldots, n$.

In case the constraint C is violated, it is impossible to find a flow with the above mentioned properties in the corresponding digraph G. We propose to extend G with certain arcs, such that it becomes possible to find a feasible flow corresponding to a solution. We call these arcs *violation arcs*, and they are denoted by $\tilde{A}$. Violation arcs may appear anywhere in the graph. The only restriction we impose on them is that after their addition, there exists a feasible flow in $\tilde{G} = (V, A \cup \tilde{A})$ that represents a solution to C. In many cases, however, we require a specific set of solutions to be represented by a flow in G. For example, we may wish that all variable-value combinations are possible solutions to C. This should be taken into account when we define the set of violation arcs $\tilde{A}$.

The next step is to make a connection with the violation measures for a constraint. This is done by applying a "cost" function $w : A \cup \tilde{A} \to \mathbb{Q}$ to $\tilde{G}$ in

the following way. For all arcs $a \in A$ we define $w(a) = 0$, while $w(a) \geq 0$ for all arcs $a \in \tilde{A}$. Then each flow f in $\tilde{G}$ has an associated cost

$$\sum_{a \in A \cup \tilde{A}} w(a)f(a) = \sum_{a \in \tilde{A}} w(a)f(a).$$

After the addition of violation arcs, a solution to C corresponds to a feasible flow in $\tilde{G}$ with an associated cost. If the flow does not use any violation arcs, this cost is 0. Otherwise, the cost of the flow depends on the costs we impose on the violation arcs. Hence we can define a violation measure as follows. For each solution to C we define its cost of violation as the minimum-weight flow in $\tilde{G}$ that represents this solution. Conversely, for many existing violation measures it is possible to choose a particular set of violation arcs and associated costs, such that a minimum-weight flow in $\tilde{G}$ representing a solution is exactly the cost of violation of that solution. In the following sections we provide several examples for different constraints and different violation measures. We often denote the extended digraph $\tilde{G}$ as G_μ to indicate its dependence on some violation measure μ.

In other words, if C is represented by the digraph $G = (V, A)$ and we can find violation arcs $\tilde{A}$ and a cost function $w : A \cup \tilde{A} \to \mathbb{Q}_+$ that represent violation measure μ, then the soft version of C with respect to μ, i.e. $\texttt{soft\_}C(x_1, \ldots, x_n, z, \mu)$, is represented by the digraph $G_\mu = (V, A \cup \tilde{A})$ with cost function w. By construction, we have the following result.

Theorem 4.4. *The constraint* $\texttt{soft\_}C(x_1, \ldots, x_n, z, \mu)$ *is hyper-arc consistent if and only if*

i) for all $i \in \{1, \ldots, n\}$ *and all* $d \in D_i$ *there is an arc* $a \in A$ *representing* (x_i, d)*, such that there exists a feasible flow* f *in* G_μ *that represents a solution to* $\texttt{soft\_}C$ *with* $f(a) = 1$ *and* $\text{cost}(f) \leq \max D_z$*,*

ii) the minimum cost of all such flows f *is not larger than* $\min D_z$*.*

Theorem 4.4 gives rise to the following propagation algorithm, presented as Algorithm 2. For a sequence of variables $X = x_1, \ldots, x_n$, and a constraint $\texttt{soft\_}C(X, z, \mu)$ the algorithm first builds the digraph G_μ that represents the constraint. Then, for all variable-value pairs (x_i, d) we check whether the pair belongs to a solution, i.e. whether there exists a flow in G_μ that represents a solution containing $x_i = d$, with cost smaller than $\max D_z$. If this is not the case, we can remove d from D_i. Finally, we update D_z, if necessary.

The time complexity of this algorithm is $O(ndK)$ where d is the maximum domain size and K is the time complexity to compute a flow in G_μ corresponding to a solution to $\texttt{soft\_}C$. However, similar to the propagation algorithm for the weighted `alldifferent` constraint in Section 3.5.2, we can improve the efficiency by applying Theorem 2.2.

The resulting, more efficient, algorithm is as follows. We first compute an initial minimum-weight flow f in G_μ representing a solution. Then for all arcs $a = (x_i, d)$ with $f(a) = 0$, we compute a minimum-weight directed $d - x_i$

Algorithm 2 Hyper-arc consistency for $\texttt{soft\_}C(X, z, \mu)$

```
set minimum = ∞
construct G_μ = (V, A ∪ Ã)
for x_i ∈ X do
  for d ∈ D_i do
    compute a minimum-weight flow f in G_μ that represents a solution to soft_C,
    with f(a) = 1 for some a ∈ A that represents (x_i, d)
    if cost(f) > max D_z then
      remove d from D_i
    end if
    if cost(f) < minimum then
      set minimum = cost(f)
    end if
  end for
end for
if min D_z < minimum then
  set min D_z = minimum
end if
```

path in the residual graph $(G_\mu)_f$. Together with a, P forms a directed circuit. Provided that $c(b) \geq 1$ for all arcs $b \in P$, we reroute the flow over the circuit and obtain a flow f'. Then $\text{cost}(f') = \text{cost}(f) + \text{cost}(P)$, because $w(a) = 0$ for all $a \in A$. If $\text{cost}(f') > \max D_z$ we remove d from the domain of x_i.

This reduces the time complexity of the algorithm to $O(K + nd \cdot \text{SP})$ where SP denotes the time complexity to compute a minimum-weight directed path in G_μ.

It should be noted that a similar algorithm was first applied by Régin [1999a, 2002] to make the weighted global cardinality constraint hyper-arc consistent.

4.4 Soft Alldifferent Constraint

4.4.1 Definitions

To the `alldifferent` constraint we apply two measures of violation: the variable-based violation measure μ_{var} and the decomposition-based violation measure μ_{dec}. Let $X = x_1, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, \ldots, D_n$. For $\texttt{alldifferent}(x_1, \ldots, x_n)$ we have

$$\mu_{\text{var}}(x_1, \ldots, x_n) = \textstyle\sum_{d \in D_X} \max\left(|\{i \mid x_i = d\}| - 1, 0\right),$$
$$\mu_{\text{dec}}(x_1, \ldots, x_n) = |\{(i, j) \mid x_i = x_j, \text{ for } i < j\}|$$

Example 4.2. Consider again the over-constrained CSP of Example 4.1:

$$x_1 \in \{a, b\}, x_2 \in \{a, b\}, x_3 \in \{a, b\}, x_4 \in \{b, c\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

We have

$$\begin{aligned}
&\mu_{\text{var}}(a,a,b,c) = 1,\ \mu_{\text{dec}}(a,a,b,c) = 1,\\
&\mu_{\text{var}}(a,a,b,b) = 2,\ \mu_{\text{dec}}(a,a,b,b) = 2,\\
&\mu_{\text{var}}(a,a,a,b) = 2,\ \mu_{\text{dec}}(a,a,a,b) = 3,\\
&\mu_{\text{var}}(b,b,b,b) = 3,\ \ \mu_{\text{dec}}(b,b,b,b) = 6.
\end{aligned}$$

□

If we apply Definition 4.1 to the `alldifferent` constraint using the measures μ_{var} and μ_{dec}, we obtain `soft_alldifferent`$(x_1,\ldots,x_n,z,\mu_{\text{var}})$ and `soft_alldifferent`$(x_1,\ldots,x_n,z,\mu_{\text{dec}})$. Each of the violation measures μ_{var} and μ_{dec} gives rise to a different hyper-arc consistency propagation algorithm. Before we present them, we introduce the graph representation of the `alldifferent` constraint in terms of flows.

4.4.2 Graph Representation

As we have seen in Section 3.2.1, a solution to the `alldifferent` constraint corresponds to a matching in the corresponding value graph. We can give an equivalent representation in terms of flows as follows (see also Régin [1994]).

Theorem 4.5. *A solution to* `alldifferent`$(x_1,\ldots,x_n)$ *corresponds to an integer feasible $s-t$ flow of value n in the digraph $\mathcal{A}=(V,A)$ with vertex set*

$$V = X \cup D_X \cup \{s,t\}$$

and arc set

$$A = A_s \cup A_X \cup A_t,$$

where

$$\begin{aligned}
A_s &= \{(s,x_i) \mid i \in \{1,\ldots,n\}\},\\
A_X &= \{(x_i,d) \mid d \in D_i, i \in \{1,\ldots,n\}\},\\
A_t &= \{(d,t) \mid d \in D_X\},
\end{aligned}$$

with capacity function $c(a)=1$ for all $a \in A$.

Proof. With an integer feasible $s-t$ flow f of value n in $\mathcal{A}$ we associate the assignment $x_i = d$ for all arcs $a=(x_i,d) \in A_X$ with $f(a)=1$. Because $c(a)=1$ for all $a \in A_s \cup A_t$, this is indeed a solution to the `alldifferent` constraint. As value$(f)=n$, all variables have been assigned a value. Similarly, each solution to the `alldifferent` gives rise to a corresponding appropriate flow in $\mathcal{A}$. □

Example 4.3. Consider again the CSP from Example 4.1. In Figure 4.1 the corresponding graph representation of the `alldifferent` constraint is presented. □

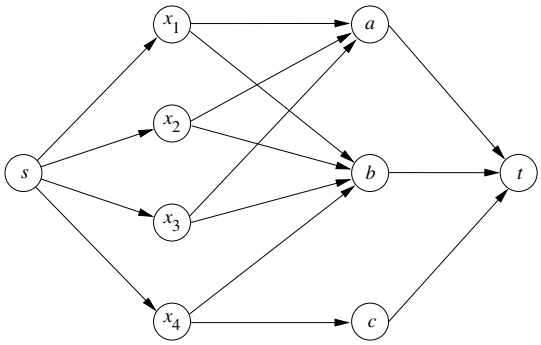


Figure 4.1. Graph representation for the `alldifferent` constraint. For all arcs the capacity is 1.

We can recognize the value graph of X in $\mathcal{A}$, being the subgraph on X and D_X. An integer feasible flow f of value n in $\mathcal{A}$ corresponds to a matching M in the value graph as follows:

$$f(a) = 1 \Leftrightarrow a \in M \quad \text{for all } a \in A_X.$$

Similarly, the graph G_M in Section 3.4.4 corresponds to the subgraph on X and D_X of the residual graph $\mathcal{A}_f$.

4.4.3 Variable-Based Violation Measure

The results in this section are originally due to Petit, Régin, and Bessière [2001]. We state their result in terms of our method, by adding violation arcs to the graph representing the `alldifferent` constraint.

To graph $\mathcal{A}$ of Theorem 4.5 we add the violation arcs $\tilde{A}_t = \{(d,t) \mid d \in D_X\}\}$ (in fact, $\tilde{A}_t$ is a copy of A_t), with demand $d(a) = 0$ and capacity $c(a) = n$ for all arcs $a \in \tilde{A}_t$. Further, we apply a cost function $w : A \cup \tilde{A}_t \to \{0,1\}$, where

$$w(a) = \begin{cases} 1 \text{ if } a \in \tilde{A}_t, \\ 0 \text{ otherwise.} \end{cases}$$

Let the resulting digraph be denoted by $\mathcal{A}_{\text{var}}$.

Example 4.4. Consider again the CSP from Example 4.1. In Figure 4.2 the corresponding graph representation of the variable-based `soft_alldifferent` constraint is presented. □

Corollary 4.6. *The constraint* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ there exists an integer feasible $s-t$ flow f of value n in $\mathcal{A}_{\text{var}}$ with $f(a) = 1$ and $\text{weight}(f) \leq \max D_z$, *and*

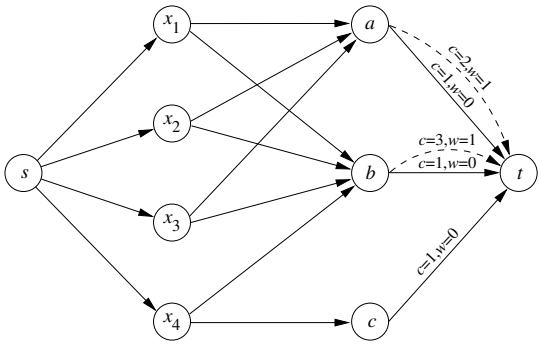


Figure 4.2. Graph representation for the variable-based `soft_alldifferent` constraint. For all arcs the capacity $c = 1$, unless specified otherwise. Dashed arcs indicate the inserted weighted arcs with weight $w = 1$.

ii) $\min D_z \geq \text{weight}(f)$ *for a feasible minimum-weight* $s-t$ *flow* f *of value* n *in* $\mathcal{A}_{\text{var}}$.

Proof. The weights on the arcs in $\tilde{A}_t$ are chosen such that the weight of a minimum-weight flow of value n is exactly μ_{var} for the corresponding solution. The result follows from Theorem 4.4. □

The constraint `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ can now be made hyper-arc consistent in the following way. First we compute a minimum-weight flow f in $\mathcal{A}_{\text{var}}$ in $O(m\sqrt{n})$ time, using the algorithm by Hopcroft and Karp [1973]. If $\text{weight}(f) > \max D_z$ or $\min D_z > \text{weight}(f)$ the constraint is inconsistent. Otherwise, we distinguish two situations: either $\text{weight}(f) < \max D_z$ or $\text{weight}(f) = \max D_z$.

Forcing a flow to use an unused arcs in A_X can only increase $\text{weight}(f)$ by 1. Hence, if $\min D_z \leq \text{weight}(f) < \max D_z$, all arcs in A_X are consistent.

If $\min D_z \leq \text{weight}(f) = \max D_z$, an unused arc $a = (x_i, d)$ in A_X is consistent if and only if there exists a $d - x_i$ path in $(\mathcal{A}_{\text{var}})_f$ with weight 0. We can find these paths in $O(m)$ time, where $m = |A_X|$. Finally, we update $\min D_z = n - |M|$ if $\min D_z < n - |M|$. Then, by Corollary 4.6, the `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{var}})$ is hyper-arc consistent.

4.4.4 Decomposition-Based Violation Measure

For the decomposition-based `soft_alldifferent` constraint, we add the following violation arcs to the graph representing the `alldifferent` constraint.

In the graph $\mathcal{A}$ of Theorem 4.5 we replace the arc set A_t by $\tilde{A}_t = \{(d, t) \mid d \in D_i, i = 1, \ldots, n\}$, with demand $d(a) = 0$ and capacity $c(a) = 1$ for all arcs $a \in \tilde{A}_t$. Note that $\tilde{A}_t$ contains parallel arcs if two or more variables share a domain value. If there are k parallel arcs (d, t) between some $d \in D_X$ and t,

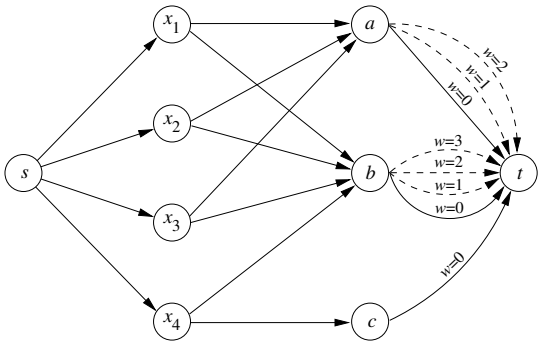


Figure 4.3. Graph representation for the decomposition-based `soft_alldifferent` constraint. For all arcs the capacity $c = 1$. Dashed arcs indicate the inserted weighted arcs with weight w as specified.

we distinguish them by numbering the arcs as $(d,t)_0, (d,t)_1, \ldots, (d,t)_{k-1}$ in a fixed but arbitrary way. One can view the arcs $(d,t)_0$ to be the original arc set A_t.

We apply a cost function $w : A \cup \tilde{A}_t \to \mathbb{N}$ as follows. If $a \in \tilde{A}_t$, so $a = (d,t)_i$ for some $d \in D_X$ and integer i, the value of $w(a) = i$. Otherwise $w(a) = 0$. Let the resulting digraph be denoted by $\mathcal{A}_{\text{dec}}$.

Example 4.5. Consider again the CSP from Example 4.1. In Figure 4.3 the corresponding graph representation of the decomposition-based `soft_alldifferent` constraint is presented. □

Corollary 4.7. *The constraint* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ *there exists an integer feasible* $s-t$ *flow* f *of value* n *in* $\mathcal{A}_{\text{dec}}$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D_z$*, and*

ii) $\min D_z \geq \text{weight}(f)$ *for a feasible minimum-weight* $s-t$ *flow* f *of value* n *in* $\mathcal{A}_{\text{dec}}$*.*

Proof. The weights on the arcs in $\tilde{A}_t$ are chosen such that the weight of a minimum-weight flow of value n is exactly μ_{dec}. Namely, the first arc entering a value $d \in D_X$ causes no violation and chooses outgoing arc with weight 0. The k-th arc that enters d causes $k-1$ violations and chooses outgoing arc with weight $k-1$. The result follows from Theorem 4.4. □

The constraint `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ can be made hyper-arc consistent by applying Algorithm 2. We first compute a minimum-weight flow f in $\mathcal{A}_{\text{dec}}$. We do this by computing n shortest $s-t$ paths in the residual graph. Because there are only weights on arcs in A_t, each shortest path takes $O(m)$ time to compute. Hence we can compute f in $O(nm)$ time. If $\text{weight}(f) > \max D_z$ we know that the constraint is inconsistent.

To identify the arcs $a = (x_i, j) \in A_X$ that belong to a flow g with value$(g) = n$ and weight$(g) \leq \max D_z$ we apply Theorem 2.2. Thus, we search for a shortest $j - x_i$ path in $(\mathcal{A}_{\text{dec}})_f$ that together with a forms a circuit C. We can compute all such shortest paths in $O(m)$ time, using the following result.

Theorem 4.8. *Let* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ *be consistent and let f be an integer feasible minimum-weight flow in $\mathcal{A}_{\text{dec}}$ of value n. Then* `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ *can be made hyper-arc consistent in $O(m)$ time.*

Proof. The complexity of the filtering algorithm depends on the computation of the minimum-weight $d - x_i$ paths in $(\mathcal{A}_{\text{dec}})_f$ for arcs $(x_i, d) \in A_X$. We make use of the fact that only arcs $a \in \tilde{A}_t$ contribute to the cost of such path.

Consider the strongly connected components of the graph $(\tilde{\mathcal{A}}_{\text{dec}})_f$ which is a copy of $(\mathcal{A}_{\text{dec}})_f$ where s and t and all their incident arcs are removed. Let P be a minimum-weight $d - x_i$ path P in $\mathcal{A}_f$. If P is equal to d, x_i then $f(x_i, d) = 1$ and cost$(P) = 0$. Otherwise, either x_i and d are in the same strongly connected component of $(\tilde{\mathcal{A}}_{\text{dec}})_f$, or not. In case they are in the same strongly connected component, P can avoid t in $\mathcal{A}_f$, and cost$(P) = 0$. In case x_i and d are in different strongly connected components, P must visit t, and we do the following.

Split t into two vertices t^{in} and t^{out} such that $\delta^{\text{in}}(t^{\text{in}}) = \delta^{\text{in}}(t)$, $\delta^{\text{out}}(t^{\text{in}}) = \varnothing$, and $\delta^{\text{in}}(t^{\text{out}}) = \varnothing$, $\delta^{\text{out}}(t^{\text{out}}) = \delta^{\text{out}}(t)$. For every vertex $v \in X \cup D_X$ we can compute the minimum-weight path from v to t^{in} and from t^{out} to v in total $O(m)$ time.

The strongly connected components of $(\tilde{\mathcal{A}}_{\text{dec}})_f$ can be computed in $O(n + m)$ time, following Tarjan [1972]. Hence the total time complexity of achieving hyper-arc consistency is $O(m)$, as $n < m$. □

Hence, we update $D_i = D_i \setminus \{j\}$ if weight(f) + weight$(C) > \max D_z$. Finally, we update $\min D_z =$ weight(f) if $\min D_z <$ weight(f). Then, by Corollary 4.7, the `soft_alldifferent`$(x_1, \ldots, x_n, z, \mu_{\text{dec}})$ is hyper-arc consistent.

4.5 Soft Global Cardinality Constraint

4.5.1 Definitions

A global cardinality constraint (`gcc`) on a sequence of variables specifies for each value in the union of their domains an upper and lower bound to the number of variables that are assigned to this value. A hyper-arc consistency propagation algorithm for the `gcc` was developed by Régin [1996], making use of network flows. A variant of the `gcc` is the `cost_gcc`, which can be seen as a weighted version of the `gcc` Régin [1999a, 2002]. For the `cost_gcc` a fixed

cost is assigned to each variable-value assignment and the goal is to satisfy the gcc with minimum total cost.

Throughout this section, we use the following notation. For a sequence of variables $X = x_1, \ldots, x_n$ with respective finite domains $D_1, \ldots, D_n$, let $l_d, u_d \in \mathbb{N}$ with $l_d \leq u_d$ for all $d \in D_X$.

Definition 4.9 (Global cardinality constraint).

$$\texttt{gcc}(X, l, u) = \{(d_1, \ldots, d_n) \mid d_i \in D_i, l_d \leq |\{d_i \mid d_i = d\}| \leq u_d \; \forall \; d \in D_X\}.$$

Note that the gcc is a generalization of the alldifferent constraint. If we set $l_d = 0$ and $u_d = 1$ for all $d \in D_X$, the gcc is equal to the alldifferent constraint.

Definition 4.10 (Weighted global cardinality constraint). *Let z be a variable with finite domain D_z and let $w_{ij} \in \mathbb{Q}$ for $i = 1, \ldots, n$ and all $j \in D_X$. Then*

$$\texttt{cost\_gcc}(X, l, u, z, w) = \Big\{(d_1, \ldots, d_n, \tilde{d}) \mid d_i \in D_i, \tilde{d} \in D_z, \\ l_d \leq |\{d_i \mid d_i = d\}| \leq u_d \; \forall \; d \in D_X, \textstyle\sum_{i,d_i=j} w_{ij} \leq \tilde{d}\Big\}.$$

Note that the cost_gcc is equal to the minweight_alldifferent constraint if we set $l_d = 0$ and $u_d = 1$ for all $d \in D_X$.

Example 4.6. Consider the CSP

$$x_1 \in \{1,2,3\}, x_2 \in \{1,2\}, x_3 \in \{2,3\}, x_4 \in \{1,3\}, \\ \texttt{gcc}(x_1, x_2, x_3, x_4, [0,1,1], [3,2,3]).$$

The gcc states that we should assign

to value 1 between 0 and 3 variables,
to value 2 between 1 and 2 variables,
to value 3 between 1 and 3 variables.

A solution to this CSP is the tuple $(1, 2, 3, 1)$. □

In order to define measures of violation for the gcc, it is convenient to introduce for each domain value a "shortage" function $s : D_1 \times \cdots \times D_n \times D_X \to \mathbb{N}$ and an "excess" function $e : D_1 \times \cdots \times D_n \times D_X \to \mathbb{N}$ as follows:

$$s(X, d) = \begin{cases} l_d - |\{x_i \mid x_i = d\}| & \text{if } |\{x_i \mid x_i = d\}| \leq l_d, \\ 0 & \text{otherwise.} \end{cases}$$

$$e(X, d) = \begin{cases} |\{x_i \mid x_i = d\}| - u_d & \text{if } |\{x_i \mid x_i = d\}| \geq u_d, \\ 0 & \text{otherwise,} \end{cases}$$

To the gcc we apply two measures of violation: the variable-based violation measure μ_{var} and the *value-based* violation measure μ_{val} that we will define in this section. The next lemma expresses μ_{var} in terms of the shortage and excess functions.

Lemma 4.11. *For* $\texttt{gcc}(X, l, u)$ *we have*

$$\mu_{\text{var}}(X) = \max\left(\sum_{d\in D_X} s(X,d), \sum_{d\in D_X} e(X,d)\right)$$

provided that

$$\sum_{d\in D_X} l_d \leq |X| \leq \sum_{d\in D_X} u_d. \tag{4.1}$$

Proof. Note that if (4.1) does not hold, there is no variable assignment that satisfies the gcc, and μ_{var} cannot be applied.

Applying μ_{var} corresponds to the minimal number of re-assignments of variables until both $\sum_{d\in D_X} s(X,d) = 0$ and $\sum_{d\in D_X} e(X,d) = 0$.

Assume $\sum_{d\in D_X} s(X,d) \geq \sum_{d\in D_X} e(X,d)$. Variables assigned to values $d' \in D_X$ with $s(X,d') > 0$ can be assigned to values $d'' \in D_X$ with $e(X,d'') > 0$, until $\sum_{d\in D_X} e(X,d) = 0$. In order to achieve $\sum_{d\in D_X} s(X,d) = 0$, we still need to re-assign the other variables assigned to values $d' \in D_X$ with $s(X,d') > 0$. Hence, in total we need to re-assign exactly $\sum_{d\in D_X} s(X,d)$ variables.

Similarly when we assume $\sum_{d\in D_X} s(X,d) \leq \sum_{d\in D_X} e(X,d)$. Then we need to re-assign exactly $\sum_{d\in D_X} e(X,d)$ variables. □

Without assumption (4.1), the variable-based violation measure for the gcc cannot be applied. Therefore, we introduce the following violation measure for the gcc, which can also be applied when assumption (4.1) does not hold.

Definition 4.12 (Value-based violation measure). *For* $\texttt{gcc}(X, l, u)$ *the* value-based violation measure *is*

$$\mu_{\text{val}}(X) = \sum_{d\in D_X} \left(s(X,d) + e(X,d)\right).$$

Example 4.7. Let $X = x_1, x_2, x_3, x_4$ be a sequence of variables. Consider the over-constrained CSP

$$x_1 \in \{1,2\}, x_2 \in \{1\}, x_3 \in \{1,2\}, x_4 \in \{1\},$$
$$\texttt{gcc}(x_1, x_2, x_3, x_4, [1,3], [2,5]).$$

We have

tuple	$\sum s$	$\sum e$	μ_{var}	μ_{val}
$(1,1,1,1)$	3	2	3	5
$(2,1,1,1)$	2	1	2	3
$(1,1,2,1)$	2	1	2	3
$(2,1,2,1)$	1	0	1	1

where $\sum s$ denotes $\sum_{d\in D_X} s(X,d)$ and $\sum e$ denotes $\sum_{d\in D_X} e(X,d)$. □

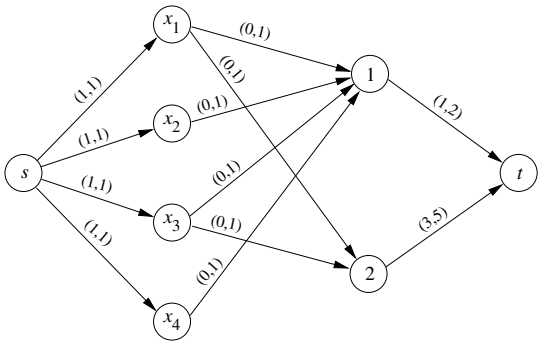


Figure 4.4. Graph representation for the gcc. Demand and capacity are indicated between parentheses for each arc a as $(d(a), c(a))$.

4.5.2 Graph Representation

A graph representation for the gcc was given by Régin [1996].

Theorem 4.13. *A solution to* $\texttt{gcc}(X, l, u)$ *corresponds to an integer feasible* $s - t$ *flow of value* n *in the digraph* $\mathcal{G} = (V, A)$ *with vertex set*

$$V = X \cup D_X \cup \{s, t\}$$

and arc set

$$A = A_s \cup A_X \cup A_t,$$

where

$$\begin{aligned} A_s &= \{(s, x_i) \mid i \in \{1, \ldots, n\}\}, \\ A_X &= \{(x_i, d) \mid d \in D_i, i \in \{1, \ldots, n\}\}, \\ A_t &= \{(d, t) \mid d \in D_X\}, \end{aligned}$$

with demand function

$$d(a) = \begin{cases} 1 \text{ if } a \in A_s, \\ 0 \text{ if } a \in A_X, \\ l_d \text{ if } a = (d, t) \in A_t, \end{cases}$$

and capacity function

$$c(a) = \begin{cases} 1 \text{ if } a \in A_s, \\ 1 \text{ if } a \in A_X, \\ u_d \text{ if } a = (d, t) \in A_t. \end{cases}$$

Example 4.8. Consider again the CSP of Example 4.7. In Figure 4.4 the corresponding graph representation of the gcc is presented. □

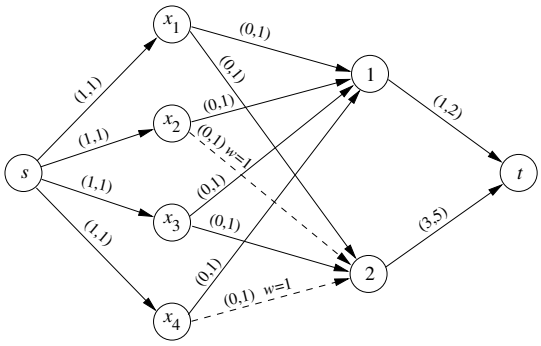


Figure 4.5. Graph representation for the variable-based `soft_gcc`. Demand and capacity are indicated between parentheses for each arc a as $(d(a), c(a))$. Dashed arcs indicate the inserted weighted arcs with weight $w = 1$.

4.5.3 Variable-Based Violation Measure

For the variable-based violation measure, we adapt the graph $\mathcal{G}$ of Theorem 4.13 in the following way. We add the violation arcs

$$\tilde{A}_X = \{(x_i, d) \mid d \notin D_i, i \in \{1, \ldots, n\}\},$$

with demand $d(a) = 0$, capacity $c(a) = 1$ for all arcs $a \in \tilde{A}_X$. Further, we apply a cost function $w : A \cup \tilde{A}_X \to \{0, 1\}$, where

$$w(a) = \begin{cases} 1 \text{ if } a \in \tilde{A}_X, \\ 0 \text{ otherwise.} \end{cases}$$

Let the resulting digraph be denoted by $\mathcal{G}_{\text{var}}$.

Example 4.9. We transform the over-constrained CSP of Example 4.7 into the following COP by softening the `gcc` using μ_{var} and cost variable z:

$$x_1 \in \{1, 2\}, x_2 \in \{1\}, x_3 \in \{1, 2\}, x_4 \in \{1\}, z \in \{0, 1, \ldots, 4\}$$
$$\texttt{soft\_gcc}(x_1, x_2, x_3, x_4, [1, 3], [2, 5], z, \mu_{\text{var}}),$$
$$\texttt{minimize } z.$$

In Figure 4.5 the corresponding graph representation of the `soft_gcc` is presented. □

Corollary 4.14. *The constraint* $\texttt{soft\_gcc}(X, l, u, z, \mu_{\text{var}})$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ *there exists an integer feasible* $s - t$ *flow* f *of value* n *in* $\mathcal{G}_{\text{var}}$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D_z$*, and*

ii) $\min D_z \geq \text{weight}(f)$ *for a feasible minimum-weight* $s - t$ *flow* f *of value* n *in* $\mathcal{G}_{\text{var}}$.

Proof. An assignment $x_i = d$ corresponds to the arc $a = (x_i, d)$ with $f(a) = 1$. By construction, all variables need to be assigned to a value and the cost function exactly measures the variable-based cost of violation. The result follows from Theorem 4.4. □

The digraph $\mathcal{G}_{\text{var}}$ corresponds to a particular instance of the `cost_gcc`. Namely, let $D_i = D_X$ for $i = 1, \ldots, n$ and define $w_{ij} = w(a)$ for all arcs $a = (x_i, j) \in A_X \cup \tilde{A}_X$. Then

$$\texttt{cost\_gcc}(X, l, u, z, w)$$

corresponds to $\texttt{soft\_gcc}(X, l, u, z, \mu_{\text{var}})$. Hence, we can apply the propagation procedures developed for that constraint directly to the `soft_gcc` in this case. The `soft_gcc` also inherits from the `cost_gcc` the time complexity of achieving hyper-arc consistency, being $O(n(m + n \log n))$ where $m = \sum_{i=1}^{n} |D_i|$ and $n = |X|$. It should be noted that this algorithm is also based on Theorem 2.2.

Beldiceanu and Petit [2004] also consider the variable-based cost measure for a different version of the soft `gcc`. Their version considers the parameters l and u to be variables too. Hence, the variable-based cost evaluation becomes a rather poor measure, as we trivially can change l and u to satisfy the `gcc`. They fix this by restricting the set of variables to consider to be the set X, which corresponds to our situation. However, they do not provide a propagation algorithm for that case.

4.5.4 Value-Based Violation Measure

For the value-based violation measure, we adapt the graph $\mathcal{G}$ of Theorem 4.13 in the following way. We add the violation arcs

$$A_{\text{shortage}} = \{(s, d) \mid d \in D_X\} \text{ and } A_{\text{excess}} = \{(d, t) \mid d \in D_X\},$$

with demand $d(a) = 0$ for all $a \in A_{\text{shortage}} \cup A_{\text{excess}}$ and capacity

$$c(a) = \begin{cases} l_d \text{ if } a = (s, d) \in A_{\text{shortage}}, \\ \infty \text{ if } a \in A_{\text{excess}}. \end{cases}$$

Further, we again apply a cost function $w : A \cup A_{\text{shortage}} \cup A_{\text{excess}} \to \{0, 1\}$, where

$$w(a) = \begin{cases} 1 \text{ if } a \in A_{\text{shortage}} \cup A_{\text{excess}}, \\ 0 \text{ otherwise.} \end{cases}$$

Let the resulting digraph be denoted by $\mathcal{G}_{\text{val}}$.

Example 4.10. We transform the over-constrained CSP of Example 4.7 into the following COP by softening the `gcc` using μ_{val} and cost variable z:

$$x_1 \in \{1, 2\}, x_2 \in \{1\}, x_3 \in \{1, 2\}, x_4 \in \{1\}, z \in \{0, 1, \ldots, 4\}$$
$$\texttt{soft\_gcc}(x_1, x_2, x_3, x_4, [1, 3], [2, 5], z, \mu_{\text{val}}),$$
$$\texttt{minimize } z.$$

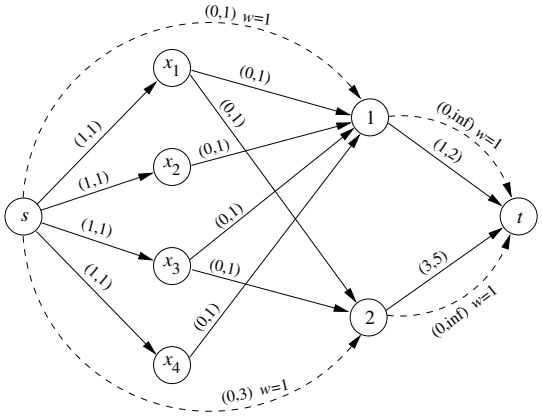


Figure 4.6. Graph representation for the value-based `soft_gcc`. Demand and capacity are indicated between parentheses for each arc a as $(d(a), c(a))$. Dashed arcs indicate the inserted weighted arcs with weight $w = 1$.

In Figure 4.6 the corresponding graph representation of the `soft_gcc` is presented. □

Corollary 4.15. *The constraint* $\texttt{soft\_gcc}(X, l, u, z, \mu_{\text{val}})$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X$ *there exists an integer feasible* $s - t$ *flow* f *of value* n *in* $\mathcal{G}_{\text{val}}$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D_z$*, and*

ii) $\min D_z \geq \text{weight}(f)$ *for a feasible minimum-weight* $s - t$ *flow* f *of value* n *in* $\mathcal{G}_{\text{val}}$*.*

Proof. Similar to the proof of Theorem 4.14. □

Unfortunately, the graph $\mathcal{G}_{\text{val}}$ does not preserve the structure of the `cost_gcc` because of the arcs A_{shortage}. Therefore we cannot blindly apply the same propagation algorithms. However, it is still possible to design an efficient propagation algorithm for the value-based `soft_gcc`, based on Algorithm 2 and Theorem 2.2.

First, we compute a minimum-weight feasible flow in $\mathcal{G}_{\text{val}}$. For this we first need to compute n shortest paths to satisfy the demand of the arcs in A_s. In order to meet the demand of the arcs in A_t, we need to compute at most another k shortest paths, where $k = |D_X|$. Hence the total time complexity is $O((n + k)(m + n \log n))$, where again $m = \sum_{i=1}^{n} |D_i|$.

In order to make the `soft_gcc` with respect to μ_{val} hyper-arc consistent, we need to check $m - n$ arcs for consistency. For this we compute $m - n$ shortest paths in the residual graph, which takes $O((m - n)(m + n \log n))$ time. Alternatively, we may compute a shortest path from every vertex in D_X to every vertex in X, which takes in total $O(k(m + n \log n))$ time.

When $l_d = 0$ for all $d \in D_X$, the arc set A_{shortage} is empty. In that case, $\mathcal{G}_{\text{val}}$ has a particular structure, i.e. the costs only appear on arcs from D_X to t. Then, similar to the reasoning for the `soft_alldifferent` constraint with respect to μ_{dec}, we can compute a minimum-weight flow of value n in $\mathcal{G}_{\text{val}}$ in $O(mn)$ time and achieve hyper-arc consistency in $O(m)$ time

4.6 Soft Regular Constraint

4.6.1 Definitions

The `regular` constraint was introduced by Pesant [2004]. It is defined on a fixed-length sequence of finite-domain variables and it states that the corresponding sequence of values taken by these variables belong to a given so-called regular language. A hyper-arc consistency propagation algorithm for this constraint was also provided by Pesant [2004]. Particular instances of the `regular` constraint can for example be applied in rostering problems or sequencing problems.

Before we introduce the `regular` constraint we need the following definitions, following Hopcroft and Ullman [1979]. An *alphabet* Σ is a finite, nonempty set of symbols. A *string* over an alphabet is a finite sequence of symbols from that alphabet. We denote the empty string by "ϵ". The set of all strings over an alphabet Σ is denoted by $\Sigma^\star$. Any subset of $\Sigma^\star$ is called a *language*. The *Kleene closure* of a language L is defined as the set of all strings that can be formed by concatenating any number of strings from L. It is denoted by $L^\star$. Note that the empty string is also in $L^\star$ because we may "concatenate" zero strings from L.

A *regular expression* over an alphabet Σ is built from Σ and the symbols "(", ")", "ϵ", "+", and "*", according to the following recursive definition:

- ϵ and each member of Σ is a regular expression,
- if α and β are regular expressions, then so is $(\alpha\beta)$
- if α and β are regular expressions, then so is $(\alpha + \beta)$
- if α is a regular expression, then so is $\alpha^\star$.

We say that every regular expression represents a *regular language* in $\Sigma^\star$, according to the interpretation of "+" as set union and "*" as Kleene closure, respectively.

A *deterministic finite automaton* (DFA) is described by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. Given an input string, the automaton starts in the initial state q_0 and processes the string one symbol at the time, applying the transition function δ at each step to update the current state. The string is *accepted* if and only if the last state reached belongs to the set of final states F. Strings processed by M that are accepted are said to belong to the

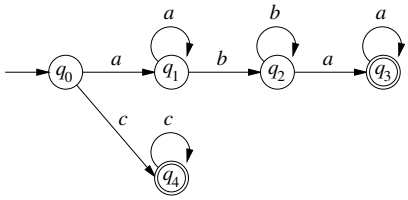


Figure 4.7. A DFA corresponding to $aa^\star bb^\star aa^\star + cc^\star$. A state is represented by a circle, a final state is represented by a double circle.

language defined by M, denoted by $L(M)$. The languages recognized by DFAs are precisely regular languages.

Given a sequence of variables $X = x_1, x_2, \ldots, x_n$ with respective finite domains $D_1, D_2, \ldots, D_n \subseteq \Sigma$, there is a natural interpretation of the set of possible instantiations of X, i.e. $D_1 \times D_2 \times \cdots \times D_n$, as a subset of all strings of length n over Σ.

Definition 4.16 (Regular language membership constraint). *Let $M = (Q, \Sigma, \delta, q_0, F)$ denote a DFA and let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D_1, D_2, \ldots, D_n \subseteq \Sigma$. Then*

$$\texttt{regular}(X, M) = \{(d_1, \ldots, d_n) \mid d_i \in D_i, d_1 d_2 \cdots d_n \in L(M)\}\,.$$

Example 4.11. Let $\Sigma = \{a, b, c\}$ be an alphabet and let $aa^\star bb^\star aa^\star + cc^\star$ be a regular expression over Σ. The corresponding DFA is depicted in Figure 4.7. Let M denote the DFA and let $L(M)$ denote the regular language defined by M. For example, $aaabaa \in L(M)$ and $cc \in L(M)$, but $aacbba \notin L(M)$.

Consider the CSP

$$x_1 \in \{a, b, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, b, c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M).$$

A solution to this CSP is for example (a, b, a, a). The CSP is not hyper-arc consistent. For example, value b can never be assigned to x_1. If we make the CSP hyper-arc consistent we obtain

$$x_1 \in \{a, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M).$$

□

To the `regular` constraint we apply two measures of violation: the variable-based violation measure μ_{var} and the *edit-based* violation measure μ_{edit} that we will define in this section.

Let s_1 and s_2 be two strings of the same length. The *Hamming distance* $H(s_1, s_2)$ is the number of positions in which they differ. The variable-based violation measure can be expressed in terms of the Hamming distance. We associate with a tuple $(d_1, d_2, \ldots, d_n)$ the string $d_1 d_2 \cdots d_n$.

Lemma 4.17. *For* `regular`(X, M) *we have*

$$\mu_{\text{var}}(X) = \min\{H(D, X) \mid D = d_1 \cdots d_n \in L(M)\}.$$

Proof. The minimum number of positions in which an assignment differs from some solution to the `regular` constraint is exactly μ_{var}. □

Another distance function that is often used for two strings is the following. Let s_1 and s_2 be two strings of the same length. The *edit distance* $E(s_1, s_2)$ is the smallest number of insertions, deletions, and substitutions required to change one string into another. It captures the fact that two strings that are identical except for one extra or missing symbol should be considered close to one another.

Definition 4.18 (Edit-based violation measure). *For* `regular`(X, M) *the* edit-based violation measure *is*

$$\mu_{\text{edit}}(X) = \min\{E(D, X) \mid D = d_1 \cdots d_n \in L(M)\}.$$

Example 4.12. Consider again the initial CSP from Example 4.11:

$$x_1 \in \{a, b, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, b, c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M).$$

We have $\mu_{\text{var}}(b, a, a, b) = 3$, because we have to change at least 3 variables. A corresponding valid string with Hamming distance 3 is for example *abba*.

On the other hand, we have $\mu_{\text{edit}}(b, a, a, b) = 2$, because we can delete the value b at the front an add the value a at the end, thus obtaining the valid string *aaba*. □

4.6.2 Graph Representation

A graph representation for the `regular` constraint was presented by Pesant [2004]. Recall that $M = (Q, \Sigma, \delta, q_0, F)$.

Theorem 4.19. *A solution to* `regular`(X, M) *corresponds to an integer feasible $s - t$ flow of value* 1 *in the digraph $\mathcal{R} = (V, A)$ with vertex set*

$$V = V_1 \cup V_2 \cup \cdots \cup V_{n+1} \cup \{s, t\}$$

and arc set

$$A = A_s \cup A_1 \cup A_2 \cup \cdots \cup A_n \cup A_t,$$

where

$$V_i = \{q_k^i \mid q_k \in Q\} \quad \text{for } i = 1, \ldots, n+1,$$

and

$$A_s = \{(s, q_0^1\},$$
$$A_i = \{(q_k^i, q_l^{i+1}) \mid \delta(q_k^i, d) = q_l^{i+1} \text{ for } d \in D_i\} \quad \text{for } i = 1, \ldots, n,$$
$$A_t = \{(q_k^{n+1}, t) \mid q_k \in F\},$$

with capacity function $c(a) = 1$ for all $a \in A$.

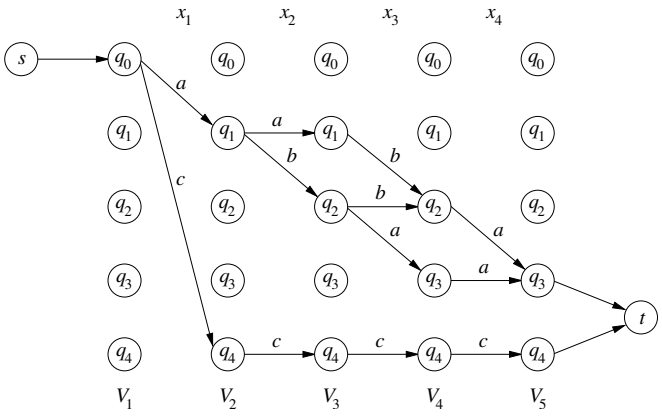


Figure 4.8. Graph representation for the `regular` constraint. For all arcs the capacity is 1.

Proof. Each arc in A_i corresponds to a variable-value pair: there is an arc from q_k^i to q_l^{i+1} if and only if there exists some $d \in D_i$ such that $\delta(q_k, d) = q_l$. If an arc belongs to an integer $s - t$ flow of value 1, it belongs to a path from q_0^1 to a member of F in the last V_{n+1}. Hence the assignment $x_i = d$ belongs to a solution to the `regular` constraint. □

Example 4.13. Consider again the hyper-arc consistent CSP from Example 4.11. In Figure 4.8 the corresponding graph representation of the `regular` constraint is presented. □

4.6.3 Variable-Based Violation Measure

For the variable-based `soft_regular` constraint, we add the following violation arcs to the graph representing the `regular` constraint.

To the graph $\mathcal{R}$ of Theorem 4.19 we add the violation arcs $\tilde{A} = \{a \mid a \in A_i, i = 1, \ldots, n\}\}$ (in fact, $\tilde{A}$ is a copy of $\cup_i A_i$), with capacity $c(a) = 1$ for all arcs $a \in \tilde{A}$. With an arc $(q_k^i, q_l^{i+1}) \in \tilde{A}$ we associate the transition $\delta(q_k, d) = q_l$ for all $d \in \Sigma$. Further, we apply a cost function $w : A \cup \tilde{A} \to \{0, 1\}$, where

$$w(a) = \begin{cases} 1 \text{ if } a \in \tilde{A}, \\ 0 \text{ otherwise.} \end{cases}$$

Let the resulting digraph be denoted by $\mathcal{R}_{\text{var}}$.

Example 4.14. Consider again the hyper-arc consistent CSP from Example 4.11. In Figure 4.9 the corresponding graph representation of the variable-based `soft_regular` constraint is presented. □

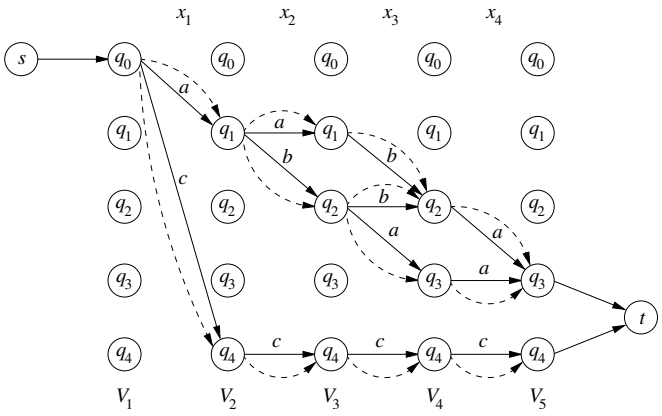


Figure 4.9. Graph representation for the variable-based `soft_regular` constraint. For all arcs the capacity is 1. Dashed arcs indicate the inserted weighted arcs with weight 1.

Corollary 4.20. *The constraint* $\texttt{soft\_regular}(X, M, z, \mu_{\text{var}})$ *is hyper-arc consistent if and only if*

i) *for every arc* $a \in A_1 \cup \cdots \cup A_n$ *there exists an integer feasible* $s-t$ *flow* f *of value* 1 *in* $\mathcal{R}_{\text{var}}$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D_z$*, and*
ii) $\min D_z \geq \text{weight}(f)$ *for a minimum-weight* $s-t$ *flow* f *of value* 1 *in* $\mathcal{R}_{\text{var}}$*.*

Proof. The weight function measures exactly μ_{var}. The result follows from Theorem 4.4. □

Note that a minimum-weight $s-t$ flow of value 1 is in fact a shortest $s-t$ path with respect to w. The constraint propagation algorithm thus must ensure that all arcs corresponding to a variable-value assignment are on an $s-t$ path with cost smaller than $\max D_z$. Computing shortest paths from the initial state in the first layer to every other node and from every node to a final state in the last layer can be done in $O(n\,|\delta|)$ time through topological sorts because of the special structure of the graph, as observed by Pesant [2004]. Here $|\delta|$ denotes the number of transitions in the corresponding DFA. The computation can also be made incremental in the same way as in Pesant [2004].

Note that this result has been obtained independently by Beldiceanu, Carlsson, and Petit [2004a].

4.6.4 Edit-Based Violation Measure

For the edit-based `soft_regular` constraint, we add the following violation arcs to the graph $\mathcal{R}$ representing the `regular` constraint.

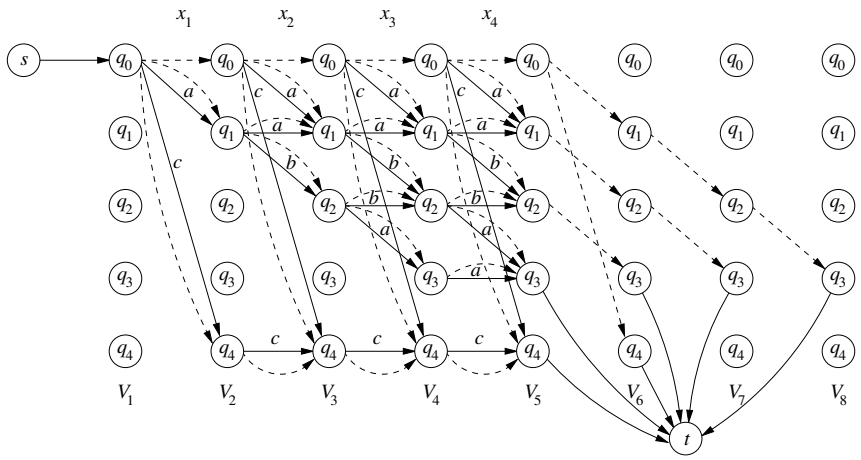


Figure 4.10. Graph representation for the edit-based `soft_regular` constraint. For all arcs the capacity is 1. Dashed arcs indicate the inserted weighted arcs with weight 1.

Similar to the previous section, we add the violation arcs $\tilde{A} = \{a \mid a \in A_i, i = 1, \ldots, n\}\}$ to allow the substitution of a value. To allow insertions and deletions, we add vertex sets $V_i = \{q_k^i \mid q_k \in Q\}$ for $i = n+2, \ldots, 2n+2$, and the arc sets

$$A_{\text{del}} = \{(q_k^i, q_k^{i+1}) \mid i = 1, \ldots, n\}\} \setminus A,$$

$$A_{\text{ins}} = \{(q_k^i, q_l^{i+1}) \mid \delta(q_k, d) = q_l \text{ for } d \in \Sigma, k \neq l, q_k \notin F, i = n{+}1, \ldots, 2n{+}2\}\},$$

and

$$\tilde{A}_t = \{(q_k^i, t) \mid q_k \in F, i = n+1, \ldots, 2n+2\}.$$

We set $c(a) = 1$ for each arc $a \in \tilde{A} \cup A_{\text{del}} \cup A_{\text{ins}} \cup \tilde{A}_t$. With an arc $(q_k^i, q_l^{i+1}) \in \tilde{A} \cup A_{\text{del}} \cup A_{\text{ins}}$ we associate the transition $\delta(q_k, d) = q_l$ for all $d \in \Sigma$. Further, we again apply a cost function $w : A \cup \tilde{A} \cup A_{\text{del}} \cup A_{\text{ins}} \cup \tilde{A}_t \to \{0, 1\}$, where

$$w(a) = \begin{cases} 1 \text{ if } a \in \tilde{A} \cup A_{\text{del}} \cup A_{\text{ins}}, \\ 0 \text{ otherwise.} \end{cases}$$

Let the resulting digraph be denoted by $\mathcal{R}_{\text{edit}}$.

Example 4.15. Consider again the hyper-arc consistent CSP from Example 4.11. In Figure 4.10 the corresponding graph representation of the edit-based `soft_regular` constraint is presented.

For example, consider the tuple $(x_1, x_2, x_3, x_4) = (b, a, a, b)$. As we have seen in Example 4.12, $\mu_{\text{edit}}(b, a, a, b) = 2$. Indeed, a shortest $s - t$ path in Figure 4.10 corresponding to this tuple has cost 2, using the directed path $s, q_0^1, q_0^2, q_1^3, q_1^4, q_2^5, q_3^6, t$:

arc (s, q_0^1) corresponds to no symbol, at cost 0,
arc (q_0^1, q_0^2) corresponds to symbol b, at cost 1,
arc (q_0^2, q_1^3) corresponds to symbol a, at cost 0,
arc (q_1^3, q_1^4) corresponds to symbol a, at cost 0,
arc (q_1^4, q_2^5) corresponds to symbol b, at cost 0,
arc (q_2^5, q_3^6) corresponds to no symbol, at cost 1,
arc (q_3^6, t) corresponds to no symbol, at cost 0. □

Corollary 4.21. *The constraint* $\texttt{soft\_regular}(X, M, z, \mu_{\text{edit}})$ *is hyper-arc consistent if and only if*

i) *for every arc* $a \in A_1 \cup \cdots \cup A_n$ *there exists an integer feasible* $s-t$ *flow* f *of value* 1 *in* $\mathcal{R}_{\text{edit}}$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D_z$, *and*
ii) $\min D_z \geq \text{weight}(f)$ *for a minimum-weight* $s-t$ *flow* f *of value* 1 *in* $\mathcal{R}_{\text{edit}}$.

Proof. The weight function measures exactly μ_{edit}. The result follows from Theorem 4.4. □

The complexity of the corresponding propagation algorithm is equal to the one for the variable-based `soft_regular` constraint. Namely, the size of $\mathcal{R}_{\text{edit}}$ is of the same order as the size of $\mathcal{R}_{\text{edit}}$, and it maintains the same structure.

4.7 Soft Same Constraint

4.7.1 Definitions

The `same` constraint is defined on two sequences of variables and states that the variables in one sequence use the same values as the variables in the other sequence. The constraint was introduced by Beldiceanu [2000]. One can also view the `same` constraint as demanding that one sequence is a permutation of the other. A hyper-arc consistency algorithm for the `same` constraint was presented by Beldiceanu, Katriel, and Thiel [2004b], making use of flow theory. The `same` constraint can be applied to rostering problems where we need to assign two types of personnel to each other. An example is the assignment of the same number of doctors and nurses on a particular date.

Definition 4.22 (Same constraint). *Let* $X = x_1, \ldots, x_n$ *and* $Y = y_1, \ldots, y_n$ *be sequences of variables with respective finite domains* $D_1, \ldots, D_n$ *and* $D'_1, \ldots, D'_n$. *Then*

$$\texttt{same}(X, Y) = \left\{ (d_1, \ldots, d_n, d'_1, \ldots, d'_n) \mid d_i \in D_i, d'_i \in D'_i, \bigcup_{i=1}^{n} \{d_i\} = \bigcup_{i=1}^{n} \{d'_i\} \right\}.$$

Note that in the above definition $\bigcup_{i=1}^{n}\{d_i\}$ and $\bigcup_{i=1}^{n}\{d'_i\}$ are families, in which elements may occur more than once.

To the `same` constraint we apply the variable-based violation measure μ_{var}. Denote the *symmetric difference* of two sets S and T by $S \Delta T$, i.e. $S \Delta T = (S \setminus T) \cup (T \setminus S)$. For $\texttt{same}(X, Y)$ we have

$$\mu_{\text{var}}(X, Y) = \left| \left(\bigcup_{i=1}^{n} \{x_i\} \right) \Delta \left(\bigcup_{i=1}^{n} \{y_i\} \right) \right| /2.$$

Example 4.16. Consider the following over-constrained CSP:

$$\begin{array}{l} x_1 \in \{a, b, c\}, x_2 \in \{c, d, e\}, x_3 \in \{c, d, e\}, \\ y_1 \in \{a, b\}, y_2 \in \{a, b\}, y_3 \in \{c, d\}, \\ \texttt{same}(x_1, x_2, x_3, y_1, y_2, y_3). \end{array}$$

We have $\mu_{\text{var}}(a, c, c, a, b, c) = 1$ because $\{a, c, c\} \Delta \{a, b, c\} = \{b, c\}$, and $|\{b, c\}| /2 = 1$. □

4.7.2 Graph Representation

A graph representation for the `same` constraint was given by Beldiceanu, Katriel, and Thiel [2004b].

Theorem 4.23. Beldiceanu et al. [2004b] *A solution to* $\texttt{same}(X, Y)$ *corresponds to an integer feasible* $s - t$ *flow of value* n *in the digraph* $\mathcal{S} = (V, A)$ *with vertex set*

$$V = X \cup (D_X \cap D'_Y) \cup Y \cup \{s, t\}$$

and arc set

$$A = A_s \cup A_X \cup A_Y \cup A_t,$$

where

$$\begin{array}{rl} A_s &= \{(s, x_i) \mid i \in \{1, \ldots, n\}\}, \\ A_X &= \{(x_i, d) \mid d \in D_i \cap D_Y, i \in \{1, \ldots, n\}\}, \\ A_Y &= \{(d, y_i) \mid d \in D'_i \cap D_X, i \in \{1, \ldots, n\}\}, \\ A_t &= \{(y_i, t) \mid i \in \{1, \ldots, n\}\}, \end{array}$$

with capacity function $c(a) = 1$ *for all* $a \in A$.

Example 4.17. Consider again the CSP from Example 4.16. In Figure 4.11 the corresponding graph representation of the `same` constraint is presented. □

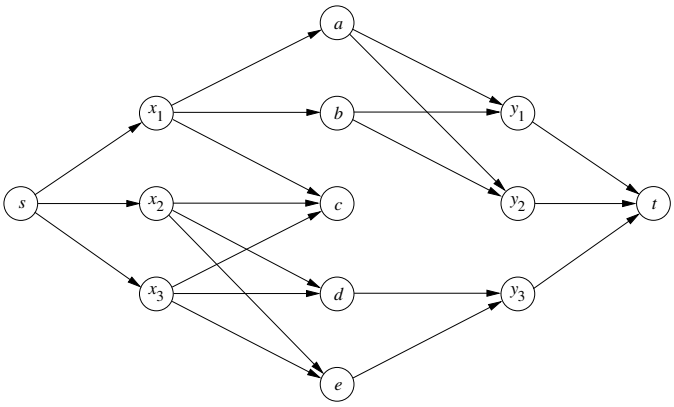


Figure 4.11. Graph representation for the `same` constraint. For all arcs the capacity is 1.

4.7.3 Variable-Based Violation Measure

To the graph $\mathcal{S}$ of Theorem 4.23 we add the arc sets

$$\tilde{A}_X = \{(x_i, d) \mid d \in D_Y \setminus D_i, i = 1, \ldots, n\}, \text{ and}$$
$$\tilde{A}_Y = \{(d, y_i) \mid d \in D_X \setminus D'_i, i = 1, \ldots, n\}$$

with capacity $c(a) = n$ for all arcs $a \in \tilde{A}_X \cup \tilde{A}_Y$. As before, we apply a cost function $w : A \cup \tilde{A}_X \cup \tilde{A}_Y \to \{0, 1\}$, where

$$w(a) = \begin{cases} 1 \text{ if } a \in \tilde{A}_X \cup \tilde{A}_Y, \\ 0 \text{ otherwise.} \end{cases}$$

Let the resulting digraph be denoted by $\mathcal{S}_{\text{var}}$.

Example 4.18. Consider again the CSP from Example 4.16. In Figure 4.12 the corresponding graph representation of the variable-based `soft_same` constraint is presented. □

Corollary 4.24. *The constraint* $\texttt{soft\_same}(X, Y, z, \mu_{\text{var}})$ *is hyper-arc consistent if and only if*

i) for every arc $a \in A_X \cup A_Y$ *there exists a feasible* $s - t$ *flow* f *of value* n *in* $\mathcal{S}_{\text{var}}$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D_z$*, and*
ii) $\min D_z \geq \text{weight}(f)$ *for a minimum-weight* $s - t$ *flow* f *of value* n *in* $\mathcal{S}_{\text{var}}$.

Proof. An assignment $x_i = d$ corresponds to the arc $a = (x_i, d)$ with $f(a) = 1$. By construction, all variables need to be assigned to a value and the cost function exactly measures the variable-based cost of violation. The result follows from Theorem 4.4. □

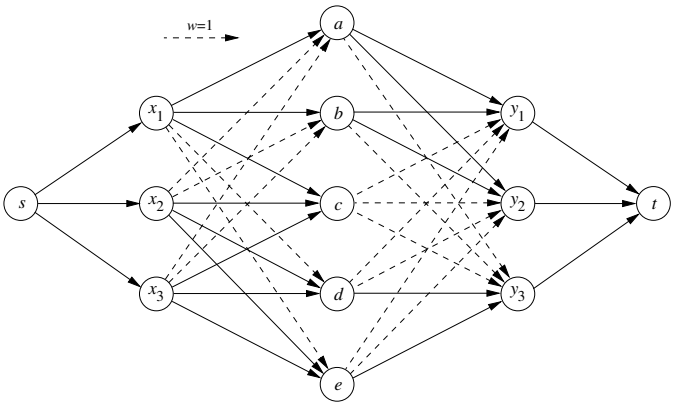


Figure 4.12. Graph representation for the variable-based `soft_same` constraint. For all arcs the capacity is 1. Dashed arcs indicate the inserted weighted arcs with weight 1.

The constraint $\texttt{soft\_same}(X, Y, z, \mu_{\text{var}})$ can be made hyper-arc consistent by applying Algorithm 2. Consistency can again be checked by computing an initial flow in $O(n(m + n \log n))$ time, and hyper-arc consistency can be achieved in $O(m(m+n \log n))$ time by applying Theorem 2.2. Here m denotes the number of arcs in $\mathcal{S}_{\text{var}}$.

4.8 Aggregating Soft Constraints

The previous sections have introduced constraint propagation algorithms based on different violation measures for several soft global constraints. If these propagation techniques are to be effective, especially in the presence of soft constraints of a different nature, they must be able to cooperate and communicate. Even though there are many avenues for combining soft constraints, the objective almost always remains to minimize constraint violations.

To combine soft constraints more effectively, Petit, Régin, and Bessière [2000] introduced "meta-constraints" on the cost variables of the soft constraints. We propose an extension of this approach by stating *soft meta-constraints* on the cost variables. We illustrate our approach with the `soft_gcc`.

Definition 4.25 (Soft global cardinality aggregator). *Let $C = \{C_1, \ldots, C_k\}$ be a set of soft constraints and let $Z = z_1, \ldots, z_k$ be a sequence of variables with respective domains $D_{z_1}, \ldots, D_{z_k}$ such that z_i represents the violation cost of C_i for $i = 1, \ldots, k$. Let $l_d, u_d \in \mathbb{N}$ with $l_d \leq u_d$ for all $d \in D_Z$. The* soft global cardinality aggregator (SGCA) *is defined as*

$$\texttt{soft\_gcc}(Z, l, u, z_{\text{agg}}, \mu)$$

where $z_{\text{agg}} \in D_{z_{\text{agg}}} \subseteq \mathbb{N}$ *is a cost variable representing some violation measure* μ*.*

We can use the SGCA to encode existing frameworks. For example, when all constraints $C_i \in C$ are either satisfied ($z_i = 1$) or violated ($z_i = 0$), the Max-CSP approach[1] is obtained by setting

$$l_0 = 0,\, u_0 = 0,$$
$$l_1 = 0,\, u_1 = k,$$

and using the violation measure

$$\mu_{\max}(Z) = \sum_{d \in D_Z} e(Z, d) = e(Z, 0),$$

where e again denotes the "excess" function (see Section 4.5).

Similar to the approach of Petit et al. [2000], the SGCA can also be used to enforce homogeneity (in a soft manner) among the violation of the soft constraints. Another possibility is to define violation measures that restrict the number of "highly violated" constraints. For example, we could wish to impose that no more than a certain number of constraints are highly violated. Since we cannot guarantee that this is possible, the use of the SGCA allows to state this wish without risking to create an inconsistent problem.

Another approach could be to set $l_i, u_i = 0$ for all $C_i \in C$ and define a violation measure that penalizes higher violation costs more, for example

$$\mu_{\text{higher}}(Z) = \sum_{d \in D_Z} d \cdot e(Z, d).$$

In the original meta-constraint framework, a similar behaviour can be established by applying a `cost_gcc` to Z. We can define a cost $w_{id} = d$ for every pair (z_i, d) where d is in the domain of z_i. Then `cost_gcc`$(Z, l, u, z_{\text{agg}}, w)$ with $l_i = 0$ and $u_i = k$ for all $C_i \in C$, is similar to the SGCA using μ_{higher}.

However, as for this variant of the `soft_gcc` we have $l = \mathbf{0}$, the `soft_gcc` is more efficient than the `cost_gcc`, as was discussed at the end of Section 4.5. In fact, the SGCA can be checked for consistency in $O(km)$ time and made hyper-arc consistent in $O(m)$ time (where $k = |C|$ and $m = \cup_i |D_{z_i}|$ whenever $l = \mathbf{0}$ and $\mu(Z) = \sum_{d \in D_Z} F(d) \cdot e(Z, d)$ for any cost function $F : D_Z \to \mathbb{Q}_+$.

4.9 Conclusion

Many real-life problems are over-constrained and cannot be solved by existing methods that seek for a feasible solution. In constraint programming, we can

[1] Recall that the Max-CSP framework tries to maximize the number of satisfied constraints.

overcome this problem by softening constraints: for each soft constraint a violation measure is defined and we seek for a solution with minimum total violation.

Until recently there were no efficient propagation algorithms for soft global constraints. In this chapter we have proposed a generic constraint propagation algorithm for a large class of global constraints; those that can be represented by a flow in a graph. To allow solutions that were originally infeasible, we have added violation arcs to the graph, with an associated cost. Hence, a flow that represents an originally infeasible solution induces a cost. This cost corresponds to a violation measure of the soft global constraint.

We have applied our method to soften several global constraints that are well-known to the constraint programming community: the `alldifferent` constraint, the global cardinality constraint, the `regular` constraint, and the `same` constraint. For these constraints we have presented the graph representations corresponding to different violation measures, some of which we have newly introduced. For all these soft global constraints efficient hyper-arc consistency algorithms have been presented, inferred from our generic algorithm.

In our propagation algorithms we have applied techniques from flow theory to compute an initial solution and to make the soft global constraints hyper-arc consistent. The application of these techniques make our algorithms very efficient. This shows that the application of operations research techniques in constraint programming is also beneficial for propagation algorithms for soft global constraints.

Part II

Search

Chapter 5

Postponing Branching Decisions

We present an effective strategy in case a value ordering heuristic produces a tie*: postponing the branching decision. To implement this strategy, we introduce the search method* domain partitioning*. We provide a theoretical and experimental analysis of our method. In Chapter 6 we show how domain partitioning can improve the computation of bounds for optimization problems.*

5.1 Introduction

In Chapter 2 we presented the general concept of a search tree to solve CSPs (or COPs), by iteratively splitting a CSP into smaller CSPs. The search tree is constructed using variable and value ordering heuristics. The efficiency of the solution process highly depends on these heuristics. In this chapter we focus on the value ordering heuristic.

A value ordering heuristic ranks domain values in such a way that the "most promising" value is selected first. If the value ordering heuristic regards two or more domain values equally promising, we say that the heuristic produces a *tie*, consisting of equally ranked domain values. The definition of ties can be extended to the concept of *heuristic equivalence* [Gomes, Selman, and Kautz, 1998] that considers equivalent all domain values that receive a rank within a given percentage from a value taken as reference.

A similar situation occurs when different value ordering heuristics can be applied simultaneously. Often a problem is composed of different aspects, for instance optimization of profit, resource balance, or feasibility of some problem constraints. For each of those aspects a value ordering heuristic may be available. However, applying only one such heuristic often does not lead to a globally satisfactory solution. The goal is then to combine these heuristics into one global value ordering heuristic. Many combinations are used in practice:

(*i*) to follow the heuristic that is regarded most important, and apply a different heuristic on values belonging to a tie,

(*ii*) to define a new heuristic (that might still contain ties) as the (weighted) sum of the ranks that each heuristic assigns to a domain value, or

(*iii*) to rank the domain values through a *multi-criteria* heuristic. In this third case, a domain value has a higher rank than another domain value if it has a higher rank with respect to all heuristics. With respect to the multi-criteria heuristic, some values may be incomparable. These incomparable values together form a tie.

As we already discussed in Chapter 2, there are several possibilities when we want to select a single domain value from a tie. Traditionally, values are chosen according to a deterministic rule, for instance lexicographic order. More recently, randomization has been applied successfully to these choices, see Gomes, Selman, and Kautz [1998] and Gomes [2003]. Once a single domain value has been selected, we can for example apply the labelling procedure to split the domain.

We propose a simple, yet effective method that improves the efficiency of the search process in case of ties: avoid making a choice and postpone the branching decision. To this end, we group together values in a tie, branch on this subdomain, and defer the decision among them to lower levels of the search tree. We call this method *domain partitioning*, or *partitioning* in short, as we split the domain according to a partition instead of single values. We show theoretically and experimentally that domain partitioning is to be preferred over labelling in case of ties. Moreover, as we will see in Chapter 6, domain partitioning in combination with techniques from operations research allows us to improve the bound computation for optimization problems.

Partitioning the domain of a variable is a well-known technique when solving CSPs or COPs. For example, the bisection splitting procedure partitions a domain in two parts. However, to our knowledge this is the first time that partitioning is applied to the ties of a value ordering heuristic, and that an analysis of partitioning with respect to labelling is presented.

The outline of this chapter is as follows. In Section 5.2 we present a detailed description of our method: domain partitioning in based on ties. A theoretical analysis of domain partitioning with respect to labelling is given in Section 5.3. This is followed by an experimental analysis in Section 5.4. We conclude with a discussion in Section 5.5.

5.2 Outline of Method

This section describes how we postpone a branching decision in a search tree. We first define the corresponding domain splitting procedure. Then we discuss the application of search strategies to the resulting search tree.

Consider a domain D and a value ordering heuristic that induces a partial order $\preceq$ on D. Define the $\preceq$-*partition* of D as a partition $D^{(1)}, D^{(2)}, \ldots, D^{(m)}$ of D, such that

- all elements in $D^{(i)}$ are incomparable with respect to $\preceq$ for $i = 1, \ldots, m$,
- $d_i \preceq d_{i+1}$ for all $d_i \in D^{(i)}$ and all $d_{i+1} \in D^{(i+1)}$, for $i = 1, \ldots, m-1$.

In other words, each subdomain $D^{(i)}$ $(1 \leq i \leq m)$ corresponds to a tie of the value ordering heuristic. It contains all values that belong to that tie. Further,

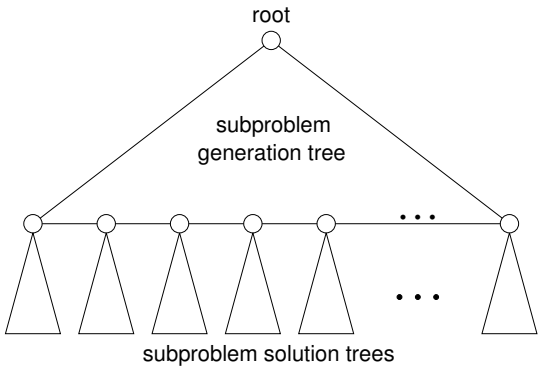


Figure 5.1. The structure of the search tree when applying domain partitioning.

$D^{(1)}$ contains all least elements of D with respect to $\preceq$. If a value ordering heuristic induces a total order, all ties consist of a single value.

We define the search method *domain partitioning*, or *partitioning* in short, as a two-phase procedure: *subproblem generation* and *subproblem solution*. The first phase, the subproblem generation, is defined as follows. For a domain D and a value ordering heuristic that induces a partial order $\preceq$ on D, we apply the domain splitting procedure that splits D into

$$D^{(1)}, D^{(2)}, \ldots, D^{(m)},$$

where $D^{(1)}, D^{(2)}, \ldots, D^{(m)}$ is the $\preceq$-partition of D. In principle one may apply any partition of D, but we will only use the $\preceq$-partition in this thesis.

The second phase deals with the subproblem solution. For this we may in principle apply any available value ordering heuristic and splitting procedure. In this thesis however, we will restrict ourselves to the lexicographic value ordering heuristic in combination with labelling to solve the subproblems. Note that the above domain splitting procedure is equivalent to labelling if the value ordering heuristic induces a total order.

In fact, partitioning constructs a *subproblem generation tree* that has subproblems as its leaves, and *subproblem solution trees*. This is depicted in Figure 5.1.

Next we consider the traversal of the resulting search tree. We may apply different search strategies to the subproblem generation tree and the subproblem solution trees. When we say that we apply a particular search strategy in combination with partitioning, we mean that we apply this strategy only to the subproblem generation tree. In this thesis we will always apply depth-first search to the subproblem solution trees. This choice is motivated by the fact that all leafs of a subproblem are heuristically equivalent, i.e. all leaves are equally likely to be a solution (with respect to the used heuristic). Hence we

choose the most efficient search strategy available, depth-first search, to visit the leaves.

Finally, we always perform a complete search on the subproblem before we backtrack to a node in the subproblem generation tree. Again this is motivated by the fact that all leafs of a subproblem are equally likely to be a solution.

5.3 Theoretical Analysis

This section shows, on a probabilistic basis, that partitioning is more beneficial than labelling in case the (combined) value ordering heuristic produces ties. We compare two search trees. One tree is built through labelling, the other is built through partitioning. Both trees apply the same variable and value ordering heuristics, that are fixed throughout this section. We assume that the total order of the descendants of a node is based on the value ordering heuristic. None of the trees applies constraint propagation.

Let a node P of a search tree have a set of descendant S. The elements of S arise from P by splitting the domain of a variable x in P. For $s, t \in S$, we denote by $s + t$ a new descendant that replaces s and t by the CSP that contains the union of s and t. That is, $s+t$ contains the same set of variables, domains and constraints as s and t, and for variable x the domain is the union of its domain in s and in t.

Similar to the analysis of LDS by Harvey and Ginsberg [1995], we assign a probability to the value ordering heuristic making a correct choice, based on a set of values. This defines a probability distribution on the set of values. Let the search tree consist of good and bad nodes. A node is called *good* if it is on a path to a leaf that is an (optimal) solution to the CSP (COP). Otherwise, the node is called *bad*. Given that the current node P is a good node and has a set of descendants S, the *heuristic probability distribution* of P is a function $h_P : S \to [0, 1]$ such that

$$\begin{aligned} &h_P(s) \geq 0 \text{ for all } s \in S, \\ &h_P(s) + h_P(t) = h_P(s + t) \text{ for all } s, t \in S, \\ &\textstyle\sum_{s \in S} h_P(s) = 1. \end{aligned}$$

For $s \in S$, $h_P(s)$ represents the probability that descendant s is also a good node. For $s, t \in S$, we assume that $h_P(s) > h_P(t)$ if the value ordering heuristic prefers s over t. If the heuristic considers s and t equally likely, then $h_P(s) = h_P(t)$.

Given a heuristic probability distribution h_P for each node P in the search tree that is not a leaf, we can compute the probability of a node being successful as follows. To each node in the search tree leads a unique path from the root P_0. Let this path be denoted as $P_0, P_1, \ldots, P_k$ for a node P_k. Then the probability of P_k being successful is

$$\text{prob}(P_k) = \prod_{i=1,\ldots,k} h_{P_{i-1}}(P_i).$$

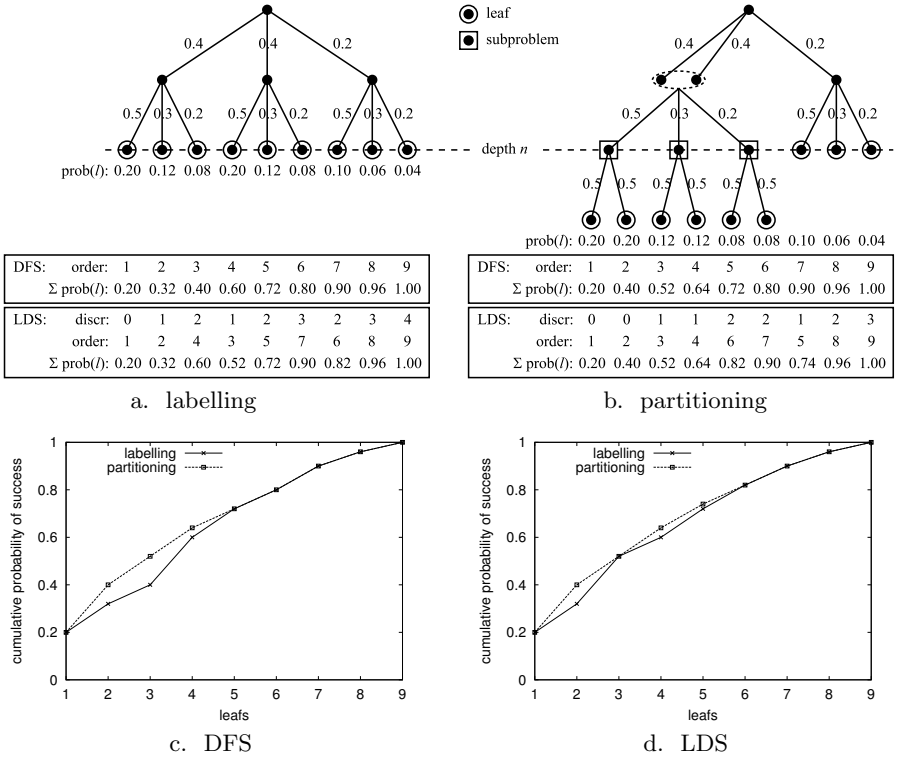


Figure 5.2. Cumulative probability of success using DFS and LDS for a CSP with $n = 2$ variables, both having 3 domain values.

Each node P in the search tree yields a different heuristic probability distribution h_P. For simplicity reasons, Harvey and Ginsberg [1995] assume that h_P remains constant throughout a binary search tree[1] with fixed depth when they analyse LDS. To analyse DDS, Walsh [1997] also uses binary search trees with fixed depth. As DDS assumes that the performance of the value ordering heuristic is better at lower parts of the tree, it is assumed that the probability of the first descendant being successful increases with the depth.

For our analysis of partitioning with respect to labelling, we assume that for all nodes P at the same depth, h_P is constant. Note that because of the fixed variable ordering heuristic, all nodes at the same depth have the same number of descendants that are ordered equally because of the fixed value ordering heuristic. The assumption implies that h_P is independent of the choices made along the path from the root to P.

When we apply a certain search strategy to the trees defined by labelling and partitioning, leaves are visited in a different order. An example of the

[1] In a binary search tree all nodes, except leaves, have exactly two descendants.

probability distribution along the leaves of the different search trees is given in Figure 5.2. The trees in Figures 5.2.a and 5.2.b correspond to 2 variables, both having 3 domain values. The descendants are ordered from left to right following the value ordering heuristic's choice. The heuristic probability distribution is shown near the branch leading to each descendant. Note that the value ordering heuristic produces a tie, consisting of two values, for the first variable. Labelling follows the heuristic on single values, while partitioning groups together values in the tie.

For DFS and LDS, the order in which the leaves are visited is given, together with the cumulative probability of success. The cumulative probability of success is also visualized in the Figures 5.2.c and 5.2.d for DFS and LDS, respectively. Observe that for every leaf, partitioning always has a higher (or equal) cumulative probability of success than labelling. This will be formalized in Theorem 5.1.

Note that in a subproblem generated by partitioning all leaves have the same probability of success. This property follows immediately from the construction of the subproblems. As a consequence, any search strategy applied to this subproblem will be equally likely to be successful. In practice, we will therefore use DFS to solve the subproblems.

Theorem 5.1. *For a fixed variable ordering and a domain value ordering heuristic, let* T_{label} *be the search tree defined by labelling, and let* $T_{\text{partition}}$ *be the search tree defined by partitioning. Let the set of the first* k *leaf nodes visited by labelling and partitioning be denoted by* L^k_{label} *and* $L^k_{\text{partition}}$ *respectively. If* T_{label} *and* $T_{\text{partition}}$ *are traversed using the same depth-first based search strategy then*

$$\sum_{l \in L^k_{\text{partition}}} \text{prob}(l) \geq \sum_{l \in L^k_{\text{label}}} \text{prob}(l). \tag{5.1}$$

Proof. For $k = 1$, (5.1) obviously holds. Let k increase until labelling and partitioning visit a leaf with a different probability of success, say l_k^{label} and $l_k^{\text{partition}}$ respectively. If such leaves do not exist, (5.1) holds with equality for all k.

Assume next that such leaves do exist, and let l_k^{label} and $l_k^{\text{partition}}$ be the first leaves with a different probability of success. As the leaves are different, there is at least one different branching decision between the two. The only possibility for this different branching decision is that we have encountered a tie, because partitioning and labelling both follow the same depth-first based search strategy. This tie made partitioning create a subproblem S, with $l_k^{\text{partition}} \in S$, and $l_k^{\text{label}} \notin S$. If labelling made a branching decision different from partitioning, with a higher probability of being successful, then partitioning would have made the same decision. Namely, partitioning and labelling follow the same strategy, and the heuristic prefers values with a higher probability. So it must be that a different branching decision made by labelling

has a smaller or equal probability of being successful with respect to the corresponding decision made by partitioning. However, as we have assumed that $\text{prob}(l_k^{\text{partition}}) \neq \text{prob}(l_k^{\text{label}})$, there must be at least one different branching decision made by labelling, that has a strictly smaller probability of being successful. Thus for the current k, (5.1) holds, and the inequality is strict.

As we let k increase further, partitioning will visit first all leaves inside S, and then continue with l_k^{label}. On the other hand, labelling will visit leaves l that are either in S or not, all with $\text{prob}(l) \leq \text{prob}(l_k^{\text{partition}})$. However, as partitioning follows the same search strategy as labelling, partitioning will either visit a leaf of a subproblem, or a leaf that labelling has already visited (possibly simultaneously). In both cases, $\sum_{l \in L^k_{\text{partition}}} \text{prob}(l) \geq \sum_{l \in L^k_{\text{label}}} \text{prob}(l)$. □

Next we measure the effect that the number of ties has on the performance of partitioning with respect to labelling. This is done by comparing the cumulative probability of success of partitioning and labelling on search trees with a varying number of ties.

We consider search trees with fixed maximum depth 30 and a *branch-width* of 3, i.e. every non-leaf node has exactly 3 descendants. This branch-width allows ties, and a larger branch-width would make it impractical to measure effectively the performance of labelling, because then the cumulative probability of success of labelling remains close to zero for a large number of visited leaves. Depending on the occurrence of a tie, the heuristic probability distribution $h_P(i)$ of descendant $i = 1, 2, 3$ of a node P will be chosen either

$h_P(1) = 0.95$, $h_P(2) = 0.04$, $h_P(3) = 0.01$ (no tie), or
$h_P(1) = 0.495$, $h_P(2) = 0.495$, $h_P(3) = 0.01$ (tie).

Our method assumes a fixed variable ordering in the search tree, and uniformly distributes the ties among them. This is reasonable, since in practice ties can appear unexpectedly. We have investigated the appearance of 10%, 33% and 50% ties out of the 30 branching decisions that lead to a leaf.

In Figures 5.3 and 5.4, we report the cumulative probability of success for labelling and partitioning using DFS and LDS until 50000 leaves. Note that in Figure 5.3 the graphs for labelling with 33% and 50% ties almost coincide along the x-axis. The figures show that in the presence of ties partitioning may be much more beneficial than labelling, i.e. the strict gap in (5.1) can be very large.

5.4 Computational Results

This section presents computational results of two applications for which we have compared partitioning and labelling. The first is the Travelling Salesman Problem (TSP), the second is the Partial Latin Square Completion Problem

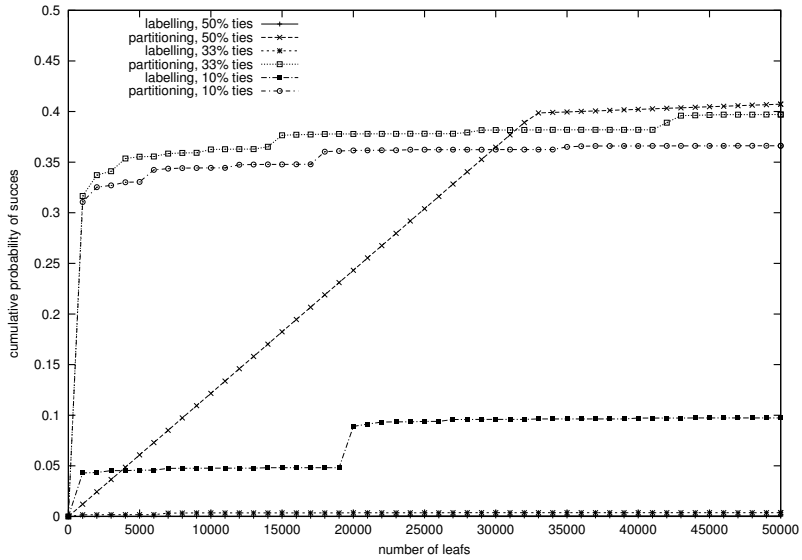


Figure 5.3. Partitioning versus labelling on search trees with maximum depth 30 and branch-width 3 using DFS.

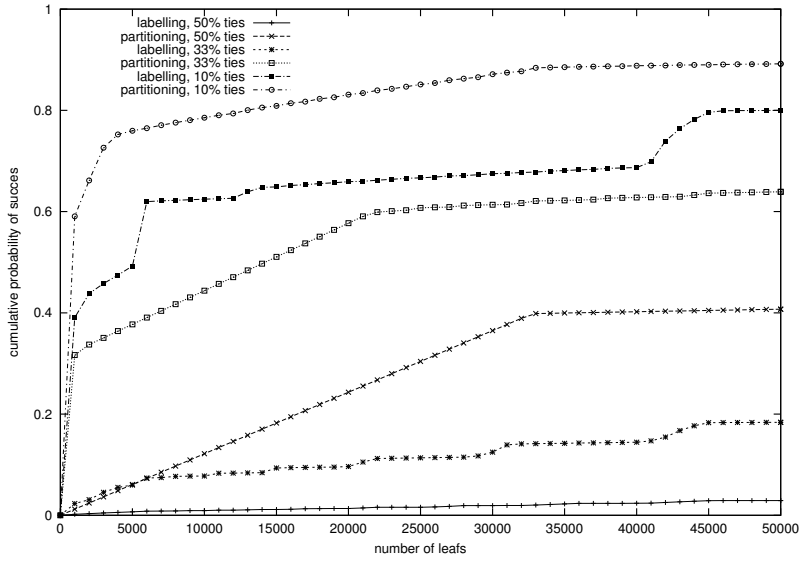


Figure 5.4. Partitioning versus labelling on search trees with maximum depth 30 and branch-width 3 using LDS.

(PLSCP). We first explain the reason why we chose these two problems to test our method.

The TSP is a constraint *optimization* problem for which the propagation is quite poor and the value ordering heuristic used is very informative and often produces ties. Instead, PLSCP is a constraint *satisfaction* problem whose model contains many `alldifferent` constraints that has an effective and efficient propagation algorithm. The value ordering heuristic used is fairly good and sometimes produces ties. Therefore, the two problems have opposite structure and characteristics.

For the TSP, partitioning is likely to be very suitable since the only drawback of the method, i.e., the decreased effect of propagation, does not play any role. On the contrary, the PLSCP is a problem whose characteristics are not likely to be suitable for partitioning. Therefore, we also analyze both the strength and the weakness of our method.

For both applications we state the problem, define the applied value ordering heuristic and report the computational results. For both problems we apply LDS as search strategy. In case of ties, labelling uses lexicographic ordering to select a value. As stated before, partitioning uses DFS to solve the subproblems.

The applications are implemented on a Pentium 1Ghz with 256 MB RAM, using ILOG Solver 4.4 as constraint programming solver and and ILOG Cplex 6.5 as linear programming solver.

5.4.1 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a well-known NP-hard constraint optimization problem. Given a set of cities with distances between them, the problem is to find a closed tour of minimal length visiting each city exactly once.

For the TSP, we have used a constraint programming model and a value ordering heuristic similar to the one that is presented in Chapter 6. We refer to that chapter for details on the model and implementation. The value ordering heuristic is based on reduced costs that are obtained from the linear relaxation of the problem. As pointed out in Chapter 6, this heuristic is very accurate. Furthermore, it is often the case that two or more domain values have the same reduced cost and a tie is produced.

To compare labelling and partitioning fairly, we stop the search as soon as an optimal solution has been found. For the considered instances, the optimal values are known in advance. The proof of optimality should not be taken into account, because it is not directly related to the probability of a leaf being an optimal solution.

The results of our comparison are presented in Table 5.1. The instances are taken from TSPLIB [Reinelt, 1991] and represent symmetric TSPs. For labelling and partitioning, the table shows the time and the number of fails (backtracks) needed to find an optimum. For labelling, the discrepancy of the

instance		labelling			partitioning		
name	n	time (s)	fails	discr	time (s)	fails	discr
gr17	17	0.08	36	2	0.02	3	0
gr21	21	0.16	52	3	0.01	1	0
gr24	24	0.49	330	5	0.01	4	0
fri26	26	0.16	82	2	0.01	0	0
bayg29	29	8.06	4412	8	0.07	82	1
bays29	29	2.31	1274	5	0.07	43	1
dantzig42	42	0.98	485	1	0.79	1317	1
swiss42	42	6.51	2028	4	0.08	15	0
hk48	48	190.96	35971	11	0.23	175	1
brazil58	58	N.A.			0.72	770	1

Table 5.1. Results for finding optima of TSP instances on n cities (not proving optimality). N.A. means "not applicable" due to time limit (300 s).

leaf node that represents the optimum is given. For partitioning, the discrepancy of the subproblem that contains the optimum is reported. Observe that we find an optimal solutions in subproblems corresponding to discrepancy 0 or 1. This indicates the accuracy of reduced costs as branching heuristic.

For all instances but one, partitioning performs much better than labelling. Both the number of fails and the computation time are substantially less for partitioning. Observe that for the instance 'dantzig42' labelling needs less fails than partitioning, but uses more time. This is because partitioning solves the subproblems using DFS. Partitioning can visit almost three times more nodes in less time, because it lacks the LDS overhead inside the subproblems.

5.4.2 Partial Latin Square Completion Problem

The *Partial Latin Square Completion Problem* (PLSCP) is a well known NP-complete constraint satisfaction problem. A *Latin square* of order $n > 0$ is an $n \times n$ square in which each row and each column is a permutation of the numbers $\{1, \ldots, n\}$. For example

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

is a Latin square of order 4. A partial Latin square is a partially pre-assigned square. The PLSCP is the problem of extending a partial Latin square to a feasible (completely filled) Latin square.

We have used the following CSP for the PLSCP of order n, where we denote a pre-assignment of value v in row i and column j by $a_{ij} = v$:

$$\begin{aligned}
\text{for all } i,j \in \{1,\ldots,n\} &: \begin{cases} x_{ij} \in \{v\} & \text{if } a_{ij} = v, \\ x_{ij} \in \{1,\ldots,n\} & \text{else,} \end{cases} \\
\text{for all } i \in \{1,\ldots,n\} &: \texttt{alldifferent}(x_{i1}, x_{i2}, \ldots, x_{in}), \\
\text{for all } j \in \{1,\ldots,n\} &: \texttt{alldifferent}(x_{1j}, x_{2j}, \ldots, x_{nj}).
\end{aligned}$$

instance	labelling			partitioning		
	time (s)	fails	discr	time (s)	fails	discr
b.o25.h238	2.36	668	5	1.09	746	5
b.o25.h239	0.49	15	1	0.42	2	1
b.o25.h240	1.17	179	4	0.86	893	4
b.o25.h241	3.31	772	3	4.70	3123	4
b.o25.h242	2.41	537	3	1.80	1753	4
b.o25.h243	4.06	1082	4	3.96	2542	4
b.o25.h244	1.33	214	3	2.99	2072	4
b.o25.h245	9.40	2308	6	10.66	12906	7
b.o25.h246	2.01	401	5	2.22	1029	4
b.o25.h247	258.91	69105	6	11.66	5727	4
b.o25.h248	33.65	6969	5	0.68	125	2
b.o25.h249	212.76	60543	11	101.46	85533	8
b.o25.h250	2.45	338	2	0.83	687	3
u.o30.h328	273.53	32538	4	82.00	14102	3
u.o30.h330	21.79	2756	3	25.15	5019	3
u.o30.h332	235.40	30033	5	56.94	9609	3
u.o30.h334	4.18	256	2	6.09	843	2
u.o30.h336	1.73	69	2	0.76	12	1
u.o30.h338	49.17	5069	3	29.41	8026	3
u.o30.h340	1.68	91	2	0.81	66	2
u.o30.h342	28.40	3152	3	5.41	600	2
u.o30.h344	9.05	605	2	8.35	1103	2
u.o30.h346	2.15	101	2	3.76	482	2
u.o30.h348	43.80	2658	2	32.86	2729	2
u.o30.h350	1.16	46	1	0.80	12	1
u.o30.h352	5.10	288	2	0.95	32	1
sum	1211.45	220793	91	396.62	159773	81
mean	46.59	8492.04	3.50	15.25	6145.12	3.12

Table 5.2. Results for PLS completion problems.

We use the maximal constraint propagation for the `alldifferent` constraints, i.e. achieving hyper-arc consistency; see Section 3.4.4. With less powerful propagation, the considered instances are practically unsolvable.

We have used the following value ordering heuristic. We rank the values according to their occurrence in the partial Latin square. Values that occur most often are to be considered first, because they are most constrained. This value ordering heuristic changes as more domains become singletons during search. A tie consists of all values that are ranked equally. Labelling selects the value with the highest rank, and uses lexicographic ordering in case of ties.

In Table 5.2 we report the performance of labelling and partitioning on a number of partial Latin square completion problems. It follows the same format as Table 5.1. The instances are generated with the PLS-generator kindly provided to us by Carla Gomes. Following remarks made by Gomes and Shmoys [2002], our generated instances are such that they are difficult to solve, i.e. they appear in the "transition phase" of the problem. The instances 'b.o25.hm' are balanced 25×25 partial Latin squares, with m unfilled entries (around 38%). Instances 'u.o30.hm' are unbalanced 30×30 partial Latin squares, with m unfilled entries (around 38%).

Although partitioning performs much better than labelling on average, the results are not homogeneous. For some instances labelling has a better

performance than partitioning. This can be explained by the propagation of the `alldifferent` constraint. Since partitioning branches on subdomains of cardinality larger than one, the `alldifferent` constraint will remove less values that are inconsistent compared to branching on single values, as is the case with labelling. Using partitioning, such values will only be removed inside the subproblems. However, even in instances where partitioning is less effective, the difference between the two strategies is not so high, while on many instances partitioning is much more effective.

As was already mentioned in Section 5.4.1, partitioning effectively applies DFS inside the subproblems. For a number of instances, partitioning finds a solution earlier than labelling, although making use of a higher number of fails.

5.5 Discussion and Conclusion

We have presented an effective method in case a value ordering heuristic produces ties: postponement of the branching decision. The method has been described in terms of domain partitioning based on the ties. We have shown both theoretically and experimentally that partitioning is to be preferred over labelling in case of ties. The experiments have also exposed a drawback of our method. Namely, the branching on subdomains instead of single values decreases the effect of constraint propagation.

There is a connection between our method and the search method *iterative broadening* by Ginsberg and Harvey [1992]. Given a branch cut-off k, iterative broadening applies depth-first search restricted to the first k descendants of each node. Then it restarts using a larger branch cut-off. If we define k dynamically for each node to be the cardinality of the (possible) tie, our first subproblem corresponds to the first run of iterative broadening. However, iterative broadening behaves differently on backtracking.

Domain partitioning can be used in several ways to exploit operations research methods in constraint programming. For example, the subproblems that are created by domain partitioning may be subject to any applicable solution method. In particular one may apply a suitable operations research method to solve a subproblem more efficiently.

In Chapter 6 we will see another effective combination of domain partitioning and operations research techniques. We will see that domain partitioning in combination with linear programming and limited discrepancy search is very useful to improve the bound computation of optimization problems.

Chapter 6

Reduced Costs as Branching Heuristic

In Chapter 2 and Chapter 5 we have seen that a search tree depends on variable ordering and value ordering heuristics. In this chapter we propose to use a value ordering heuristic based on reduced costs*, obtained from a linear relaxation of the problem at hand. Further, we apply the method of domain partitioning of Chapter 5, which allows us to improve the quality of the lower bound using* additive bounding *procedures. Experimental results show the effectiveness of our approach.*

6.1 Introduction

For many NP-hard[1] combinatorial optimization problems, one or more linear programming relaxations exist. For example, any integer linear program can be relaxed by removing the integrality constraints on the variables. When we solve linear programs to optimality, we obtain a (fractional) solution, a bound on the optimal objective value and reduced costs, as we have seen in Section 2.1.2. We recall that the reduced cost of a non-basic variable corresponds to the additional cost to be paid if this variable is inserted in the basis.

It is rare that the linear programming solution is also a solution to the original problem we want to solve. However, the reduced costs corresponding to the linear programming solution give an intuitive indication. Namely, when the problem at hand is a minimization problem, variables with a low reduced cost value are more likely to be part of the optimal solution. It should be noted that the validity of this intuition highly depends on the nature of the problem at hand. For problems that aim at feasibility rather than optimality (for example certain scheduling problems), the reduced costs may not be very informative.

In this chapter we consider COPs for which a linear programming relaxation exists. We propose to use the reduced costs obtained from the relaxation as a value ordering heuristic during the search for a solution to the COP. Furthermore, we use the domain partitioning method of Chapter 5 applied to the ties of this heuristic to find a solution earlier. Domain partitioning also allows us to improve the bound of the objective value dramatically during the search process.

[1] A combinatorial optimization problem belongs the class of NP-hard problems if the corresponding decision problem is NP-complete.

Our method is described as follows. Recall that domain partitioning consists of two phases: subproblem generation and subproblem solution. At the root of our search tree we solve the linear programming relaxation, and we order the domains of the variables by their corresponding reduced costs. We define two ties by means of a given threshold. The *good* tie consists of values corresponding to reduced costs below the threshold, the *bad* tie consists of values corresponding to reduced costs above the threshold. Doing so, partitioning splits each domain into a *good* subdomain and a *bad* subdomain, according to the ties.

We apply LDS to the subproblem generation tree. Hence, the first leaf of the subproblem generation tree consists of the subproblem in which all variables are restricted to their good domains. If the reduced costs provide an accurate value ordering heuristic, it is likely that an optimal solution will be found in a subproblem with low discrepancy. This claim is supported by the experimental results that we provide in Section 6.5. Subproblems at higher discrepancies are supposed to contain worse candidate solutions. We continue the search process either until we have proved optimality of a solution or until we have reached a given limit on the discrepancy of the subproblems.

A surprising aspect of this method is that even by using good subdomains of low cardinality, we almost always find the optimal solution in the first generated subproblem. Thus, reduced costs provide very useful information indicating for each variable which values are the most promising. Solving only the first subproblem, we obtain an effective incomplete method that finds the optimal solution in almost all test instances.

In order to be complete, the method should solve all subproblems for all discrepancies to prove optimality. Clearly, even if each subproblem could be efficiently solved this approach would not be applicable. We propose to improve the bound with considerations based on the discrepancy of the subproblems. This will allow us to prove optimality earlier.

We have applied our method to the Travelling Salesman Problem and its time-constrained variant. The latter is a typical example of a problem that is difficult for both constraint programming and integer linear programming. Namely, the problem is composed of an optimization part and a scheduling part. Constraint programming is suitable for the scheduling part, but has problems with optimization. The converse holds for integer linear programming. It has been shown by Focacci, Lodi, and Milano [2002] that a hybrid approach, combining the strengths of both, is very effective in this case.

This chapter is organized as follows: in Section 6.2 we describe the proposed method in detail. The improvement of the lower bound is presented in Section 6.3. In Section 6.4 we provide a constraint programming model and a linear relaxation for the Travelling Salesman Problem with Time Windows. The computational results on this problem are presented in Section 6.5. We discuss our results in Section 6.6 and give a conclusion.

6.2 Solution Framework

The basis of our framework is the constraint programming search tree, using the domain partitioning splitting procedure of Chapter 5. We use the reduced costs obtained from the solution of a linear programming relaxation as value ordering heuristic. In this section we first show how to extract a linear programming relaxation from a constraint programming model. Then we describe the application of reduced costs as value ordering heuristic in more detail. Finally, we propose to use a discrepancy constraint instead of standard LDS as a search strategy. Without loss of generality, we assume that we are dealing with a minimization problem in this chapter.

6.2.1 Building a Linear Programming Relaxation

Consider a COP consisting of a sequence of variables $v_1, \ldots, v_n$, corresponding finite domains $D_1, \ldots, D_n$, a set of constraints and an objective function to be optimized. For simplicity, we restrict ourselves to objective functions that are weighted sums of variable-value pairs. This means that the assignment $v_i = j$ induces a cost c_{ij} for $i = 1, \ldots, n$ and all $j \in D_i$. Then the objective function has the form

$$\sum_{i=1}^{n} c_{i\,v_i}.$$

From this COP we need to extract a linear programming relaxation with the property that a reduced cost value gives an indication of a variable-value pair to be in an optimal solution. This can be done in the following way.

We first transform the variables v_i and the domains D_i into corresponding binary variables x_{ij} for $i = 1, \ldots, n$ and all $j \in D_i$:

$$\begin{aligned} v_i = j &\Leftrightarrow x_{ij} = 1, \\ v_i \neq j &\Leftrightarrow x_{ij} = 0. \end{aligned} \tag{6.1}$$

To ensure that each variable v_i is assigned to a single value in its domain we state the linear constraint

$$\sum_{j \in D_i} x_{ij} = 1 \quad \text{for } i = 1, \ldots, n.$$

The objective function is transformed as

$$\sum_{i=1}^{n} c_{i\,v_i} = \sum_{i=1}^{n} \sum_{j \in D_i} c_{ij} x_{ij}.$$

The next, most difficult, task is to transform (some of) the problem constraints into linear constraints using the binary variables. This is problem dependent, and no general recipe exists. However, for many problems such descriptions

are known and described in the literature. For example, for an `alldifferent` constraint we may add the linear constraint

$$\sum_{i=1}^{n} x_{ij} \leq 1 \text{ for all } j \in \bigcup_{i=1}^{n} D_i$$

to ensure that every domain value is assigned to at most one variable.

Finally, in order to obtain a linear programming relaxation, we remove the integrality constraint on the binary variables and state

$$0 \leq x_{ij} \leq 1 \text{ for } i = 1, \ldots, n \text{ and all } j \in D_i.$$

The above linear programming relaxation has the desired property: the reduced cost value $\overline{c}_{ij}$ gives an indication of the assignment $v_i = j$ to be in an optimal solution for $i = 1, \ldots, n$ and all $j \in D_i$. Namely, if we enforce the assignment $v_i = j$, the objective function value will increase by (at least) $\overline{c}_{ij}$. Hence, if $\overline{c}_{ij}$ is small, the assignment $v_i = j$ is more likely to be in an optimal solution of the original problem we want to solve.

6.2.2 Domain Partitioning using Reduced Costs

We consider again the COP of the previous section, containing variables $v_1, \ldots, v_n$ with corresponding finite domains $D_1, \ldots, D_n$. We assume that we have constructed a linear programming relaxation (LP) of the problem with variables x_{ij} for $i = 1, \ldots, n$ and all $j \in D_i$ according to the transformation (6.1).

First we solve the LP to optimality. As we have seen in Section 2.1.2, we can extract a reduced cost $\overline{c}_{ij}$ for all i, j. We define the value ordering heuristic based on reduced costs as follows. Given variable v_i for some i and a threshold T, we order its domain D_i into two ties D_i^{good} and D_i^{bad} such that

$$D_i^{\text{good}} = \{j \mid j \in D_i, \overline{c}_{ij} \leq T\},$$
$$D_i^{\text{bad}} = \{j \mid j \in D_i, \overline{c}_{ij} > T\},$$

and the values in D_i^{good} are ordered smaller than the values in D_i^{bad}. In other words, we regard all values in D_i^{good} heuristically equivalent but more promising than values in D_i^{bad}, which are also regarded heuristically equivalent.

Next we apply the reduced cost-based value ordering heuristic to the domain partitioning procedure of Chapter 5. Recall that domain partitioning induces a search tree consisting of two layers: the subproblem generation tree and subproblem solution trees. As we expect the value ordering heuristic to be accurate, we apply LDS to the resulting search tree. Note that this means that we apply LDS to subproblem generation tree only. To the subproblems we apply DFS, as motivated in Chapter 5.

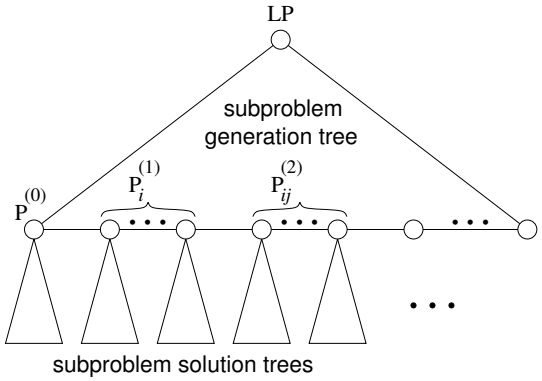


Figure 6.1. The structure of the search tree. The linear programming relaxation LP is located at the root. The subproblems $\mathrm{P}^{(k)}$ correspond to discrepancy k.

The resulting search tree is depicted in Figure 6.1. At the root, we solve the linear programming relaxation LP which provides us the reduced costs. Then we generate the first subproblem, $\mathrm{P}^{(0)}$, corresponding to discrepancy 0. For $\mathrm{P}^{(0)}$ holds that

$$v_i \in D_i^{\text{good}} \quad \text{for } i = 1, \ldots, n.$$

We perform a complete DFS search on $\mathrm{P}^{(0)}$. Hopefully, $\mathrm{P}^{(0)}$ contains an optimal solution, but it is possibly not yet proven optimal. Then we generate and search the next n subproblems, $\mathrm{P}_1^{(1)}, \ldots, \mathrm{P}_n^{(1)}$, corresponding to discrepancy 1. For $\mathrm{P}_i^{(1)}$ holds that

$$\begin{aligned} &v_i \in D_i^{\text{bad}}, \\ &v_j \in D_j^{\text{good}} \quad \text{for all } j \in \{1, \ldots, n\} \setminus \{i\}. \end{aligned}$$

For the next generated $n(n-1)/2$ subproblems $\mathrm{P}_{ij}^{(2)}$ corresponding to discrepancy 2 holds that

$$\begin{aligned} &v_i \in D_i^{\text{bad}}, \\ &v_j \in D_j^{\text{bad}}, \\ &v_k \in D_k^{\text{good}} \quad \text{for all } k \in \{1, \ldots, n\} \setminus \{i, j\}, \end{aligned}$$

for distinct i, j. The search is continued until we have reached a given limit on the discrepancy of the subproblems, or until we have proved optimality of a solution. The proof of optimality will be discussed in the Section 6.3.

The threshold T will influence the performance of our method. If T is too small, the optimal solution may not be in the first subproblem. If T is too large, a complete search on the subproblems may become impractical due to time restrictions. Hence, the "optimal" threshold should be experimentally tuned for each problem.

6.2.3 Discrepancy Constraint

In this section we propose to use a discrepancy *constraint* instead of standard LDS tree search. Then the propagation of this constraint ensures that we visit only leaves of a given discrepancy. This idea was first introduced by Focacci [2001, p. 171]. Our motivation to use a constraint is twofold. The most important reason is that the discrepancy constraint allows us to improve the bound of the linear programming relaxation, as we will see in Section 6.3. Further, if the propagation for this constraint is efficient, a discrepancy constraint may have a better performance than traditional LDS tree search. We provide experimental results to compare the performance of both.

The *discrepancy constraint* takes as input a value of discrepancy k, a sequence of variables, and their corresponding "bad" subdomains. The constraint holds if exactly k variables take their values in their bad subdomain. Formally, we define the discrepancy constraint as follows.

Definition 6.1 (Discrepancy constraint). *Let $v_1, v_2, \ldots, v_n$ be variables with respective finite domains $D_1, D_2, \ldots, D_n$ and let $D_i^{\text{bad}} \subseteq D_i$ for all i. Further, let $k \in \{0, \ldots, n\}$. Then*

$$\begin{aligned}&\texttt{discrepancy}(v_1, \ldots, v_n, D_1^{\text{bad}}, \ldots, D_n^{\text{bad}}, k) = \\ &\left\{(d_1, \ldots, d_n) \mid d_i \in D_i, \left|\{d_j \mid d_j \in D_i^{\text{bad}}\}\right| = k\right\}.\end{aligned}$$

Operationally, the `discrepancy` constraint keeps track of the number of variables that take their value in either the good or the bad domain. If during the search for a solution in the current subproblem the number of variables ranging on their bad domain is k, all other variables are forced to range on their good domain. Equivalently, if the number of variables ranging on their good domain is $n - k$, the other variables are forced to range on their bad domain.

Using the `discrepancy` constraint, the subproblem generation is defined as follows (in pseudo-code):

```
for (k=0..n) {
  add( discrepancy(v, D_bad, k) );
  solve subproblem;
  remove( discrepancy(v, D_bad, k) );
}
```

where `k` is the level of discrepancy, `v` is the array containing the variables v_i, and `D_bad` is the array containing D_i^{bad} for $i = 1, \ldots, n$. The command `solve subproblem` is shorthand for solving the subproblem.

We have compared the performance of the `discrepancy` constraint to the traditional implementation of LDS. Both methods have been applied to solve Asymmetric Travelling Salesman Problem with Time Windows (ATSPTW) instances to optimality. The instances are taken from Ascheuer [1995]. We

instance	standard LDS		discrepancy constraint	
	time (s)	fails	time (s)	fails
rbg010a	0.06	7	0.03	7
rbg016a	0.08	44	0.04	44
rbg016b	0.15	57	0.10	43
rbg017.2	0.05	14	0.05	14
rbg017	0.10	69	0.09	47
rbg017a	0.09	42	0.09	42
rbg019a	0.06	30	0.06	30
rbg019b	0.12	71	0.12	71
rbg019c	0.20	152	0.19	158
rbg019d	0.06	6	0.06	6
rbg020a	0.07	5	0.06	5
rbg021.2	0.13	66	0.13	66
rbg021.3	0.22	191	0.20	158
rbg021.4	0.11	85	0.09	40
rbg021.5	0.10	45	0.16	125
rbg021.6	0.19	110	0.20	110
rbg021.7	0.23	70	0.22	70
rbg021.8	0.15	88	0.15	88
rbg021.9	0.17	108	0.17	108
rbg021	0.19	152	0.19	158
rbg027a	0.22	53	0.21	53
sum	2.75	1465	2.61	1443
mean	0.13	69.76	0.12	68.71
median	0.12	66	0.12	53

Table 6.1. Comparison of standard tree search implementation of LDS and the discrepancy constraint on ATSPTW instances.

have used the same models and implementation as for the computational experiments described in Section 6.5. The results are presented in Table 6.1. In the table, the traditional tree search implementation of LDS is referred to as "standard LDS". The `discrepancy` constraint is referred to as "discrepancy constraint". Both time (in seconds) and fails (number of backtracks) are reported for each instance. The results indicate that the `discrepancy` constraint and the traditional tree search implementation of LDS have a comparable performance on these instances.

6.3 Discrepancy-Based Bound Improvement

In this section we improve the lower bound provided by the solution of the linear programming relaxation. This is done by taking into consideration the discrepancy of a subproblem.

First, we introduce the concept of *additive bounding*, introduced by Fischetti and Toth [1989, 1992]. The additive bounding scheme is visualized in Figure 6.2, taken from Lodi and Milano [2003].

Let P be a problem of the form

$$\min \{c^{\mathsf{T}} x \mid Ax = b, x \geq 0\} \tag{6.2}$$

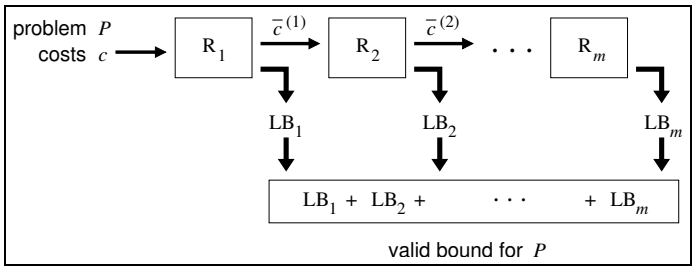


Figure 6.2. The additive bounding scheme.

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$.

In many cases, several relaxations for P are available, each exploiting a different substructure of P. Let $R_1, \ldots, R_m$ denote the m relaxations available for P. We suppose that relaxation R_i applied to P and cost vector c returns a lower bound value LB_i as well as a reduced cost vector $\overline{c}^{(i)} \in \mathbb{R}^n$. The result of Fischetti and Toth [1989] is the following:

> Apply R_1 to P with cost vector c. Next, apply sequentially $R_2, \ldots, R_m$ to P such that R_{i+1} uses the cost vector $\overline{c}^{(i)}$ for $i = 1, \ldots m-1$. Then $\sum_{i=1}^{m} \mathrm{LB}_i$ is a valid lower bound for P.

We will apply the additive bounding procedure using the following relaxations of the original problem. Here we follow the notation of the previous sections, i.e. the original problem is stated on variables $v_1, \ldots, v_n$ with corresponding domains $D_1, \ldots, D_n$. The linear programming relaxation is stated using the variables x_{ij} for $i = 1, \ldots, n$ and all $j \in D_i$.

The first relaxation will be the linear programming relaxation, which indeed provides a lower bound and a reduced cost vector. Then, for all subproblems that are generated for discrepancy $k = 1, \ldots, n$, we consider the linear description of the `discrepancy` constraint as relaxation R_k:

$$\sum_{j \in D_i^{\mathrm{bad}}} x_{ij} \leq 1 \qquad \text{for } i = 1, \ldots, n \tag{6.3}$$

$$\sum_{i=1}^{n} \sum_{j \in D_i^{\mathrm{bad}}} x_{ij} = k \tag{6.4}$$

$$x_{ij} \in \{0, 1\} \quad \text{for } i = 1, \ldots, n \text{ and all } j \in D_i. \tag{6.5}$$

The following result was observed by Focacci [2001, p. 171–172] and Lodi [2002]. We follow here the proof provided by Lodi and Milano [2003].

Theorem 6.2. *Let R be a relaxation for a problem P, providing a lower bound* LB *on P and a reduced cost vector $\overline{c}$. Then relaxation R_k:*

$$\begin{aligned} &\min z^{(k)} = \textstyle\sum_{i=1}^{n} \sum_{j \in D_i} \overline{c}_{ij} x_{ij} \\ &\text{subject to} \quad (6.3), (6.4), (6.5), \end{aligned}$$

can be solved to optimality in $O(nd)$ time, where $d = \max_{i=1,\ldots,n} |D_i|$. Moreover, $\mathrm{LB} + z^{(k)}$ is a valid lower bound for P when

$$\texttt{discrepancy}(v_1, \ldots, v_n, D_1^{\mathrm{bad}}, \ldots, D_n^{\mathrm{bad}}, k)$$

is imposed.

Proof. The optimal solution of R_k is obtained by the following algorithm:

i) compute the smallest reduced cost $\min_{j \in D_i^{\mathrm{bad}}} \overline{c}_{ij}$ for $i = 1, \ldots, n$,
ii) sort the obtained reduced costs non-decreasingly,
iii) select and sum the k smallest reduced costs.

The time complexity of *i*) is $O(nd)$, while *ii*) and *iii*) take $O(n \log n)$ time and $O(k)$ time, respectively.

Concerning the validity of the bound, this is guaranteed by the additive bounding procedure where we take as first relaxation R and as second relaxation R_k for every $k \in \{1, \ldots, n\}$. □

Note that we do not need to compute R_k from scratch for every k. In fact, we compute once a list L that contains the n smallest reduced costs $\min_{j \in D_i^{\mathrm{bad}}} \overline{c}_{ij}$ for $i = 1, \ldots, n$ and is non-decreasingly sorted. Then

$$z^{(k)} = \sum_{i=1}^{k} L[i] \text{ for } k = 1, \ldots, n.$$

Hence, we add $L[k]$ to the lower bound whenever the k-`discrepancy` constraint is imposed.

The idea of additive bounding based on discrepancy considerations has been generalized by Lodi and Milano [2003] and Lodi, Milano, and Rousseau [2003].

6.4 The Travelling Salesman Problem

In this section we present models of the Travelling Salesman Problem (TSP) and the Asymmetric Travelling Salesman Problem with Time Windows (ATSPTW) on which we have tested our method. We provide a constraint programming model for the ATSPTW and an integer linear programming model for the TSP. From the latter we infer a linear programing relaxation.

6.4.1 Constraint Programming Model

The constraint programming model of the ATSPTW makes use of the `nocycle` constraint, introduced by Caseau and Laburthe [1997a]. Before we introduce this constraint, we need the following definition.

Consider an ordered sequence $S = s_1, \ldots, s_n$ with $s_i \in \{1, \ldots, n\}$ for $i = 1, \ldots, n$. Define the set C with respect to S as follows:

$$\{1\} \in C,$$
$$i \in C \Rightarrow s_i \in C.$$

We say that S is *cyclic* if $|C| = n$.

Definition 6.3 (No cycle constraint). *Let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D_i \subseteq \{1, 2, \ldots, n\}$ for $i = 1, 2, \ldots, n$. Then*

$$\texttt{nocycle}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \in D_i, d_1, \ldots, d_n \text{ is not cyclic}\}.$$

Corresponding to the variables in Definition 6.3 we can define the digraph $G = (X, A)$ with arc set $A = \{(x_i, x_j) \mid j \in D_i, i \in \{1, \ldots, n\}\}$. An assignment $x_1 = d_1, \ldots, x_n = d_n$ corresponds to the subset of arcs $\tilde{A} = \{(x_i, x_{d_i}) \mid i \in \{1, \ldots, n\}\}$. The `nocycle` constraint ensures that $\tilde{A}$ does not contain a cycle.

Note that hyper-arc consistency for the `nocycle` constraint has a worst-case running time that is exponential in the number of variables. Namely, finding a solution to the `nocycle` constraint corresponds to finding a Hamiltonian path in the corresponding graph, which is an NP-complete problem. Therefore, we apply a weaker propagation algorithm described by Caseau and Laburthe [1997a] and Pesant, Gendreau, Potvin, and Rousseau [1998] that runs in polynomial time. Given a partial assignment of variables, it computes all paths in the corresponding digraph. Based on the starting vertex, the ending vertex and the length of a path, the algorithm removes values from unassigned variable domains.

Next we provide the constraint programming model for the ATSPTW, following Pesant, Gendreau, Potvin, and Rousseau [1998] and Focacci, Lodi, and Milano [2002].

Let $G = (V, A)$ be a digraph with vertex set $V = \{0, 1, \ldots, n, n+1\}$. Vertices 0 and $n+1$ represent the origin and destination depots, while vertices $1, \ldots, n$ represent the cities to be visited. In fact, the origin and destination depots are the same. For modeling convenience however, we have split the depot into two vertices. This transforms the original Travelling Salesman Problem into the problem of finding a directed Hamiltonian path from 0 to $n+1$. Each vertex $i \in V \setminus \{0, n+1\}$ has an associated *time window* $[a_i, b_i]$, with $a_i, b_i \in \mathbb{R}$, during which the service at city i must be executed. The duration of the service at city i is represented by $\texttt{dur}_i$. We also apply a "cost" function $c : A \to \mathbb{Q}_+$. For each pair of vertices $i, j \in V$, c_{ij} represents the travel cost from city i to city j.

We model the ATSPTW as a CSP as follows. To each vertex $i \in V$, we associate the variables

$\texttt{next}_i$: This variable represents the next city to visit after city i. The initial domain of $\texttt{next}_i$ is $\{1, \ldots, n+1\}$ for $i = 0, \ldots, n$, while we set $\texttt{next}_{n+1} = 0$.

$\texttt{cost}_i$: This variable represents the travel cost if we go from i to $\texttt{next}_i$, i.e. $c_{i\,\texttt{next}_i}$. The initial domain of $\texttt{cost}_i$ is $\{0, \ldots, \max_{i \neq j} c_{ij}\}$ for $i = 0, \ldots, n$, while we set $\texttt{cost}_{n+1} = 0$.

$\texttt{start}_i$: This variable represents the time at which the service at city i starts. The initial domain of $\texttt{start}_i$ is $\{0, \ldots, K\}$, where K is chosen appropriately large, for $i = 1, \ldots, n+1$, while we set $\texttt{start}_0 = 0$.

Finally, we introduce a cost variable z that represents the objective function, i.e. the cost of the tour, to be minimized. The initial domain of z is bounded from below by 0 and from above by $\sum_{i \in V} \max_{j \in D_i} c(i,j)$.

With these variables, we model the ATSPTW as

$$\texttt{minimize}\ z = \sum_{i \in V} \texttt{cost}_i \tag{6.6}$$

$$\texttt{alldifferent}(\texttt{next}_0, \texttt{next}_1, \ldots, \texttt{next}_{n+1}) \tag{6.7}$$

$$\texttt{nocycle}(\texttt{next}_0, \texttt{next}_1, \ldots, \texttt{next}_n) \tag{6.8}$$

$$\texttt{next}_i = j \Rightarrow \texttt{cost}_i = c_{ij} \qquad \forall i, j \in V \tag{6.9}$$

$$\texttt{next}_i = j \Rightarrow \texttt{start}_i + \texttt{dur}_i + \texttt{cost}_i \leq \texttt{start}_j \qquad \forall i, j \in V \tag{6.10}$$

$$a_i \leq \texttt{start}_i \leq b_i \ \forall i \in V \setminus \{0, n+1\} \tag{6.11}$$

The `alldifferent` constraint (6.7) states that we visit and leave each city exactly once. To forbid subtours, we impose the `nocycle` constraint (6.8) on the cities $0, \ldots, n$. Together, they ensure that we find a Hamiltonian path from vertex 0 to vertex $n+1$. The connection between the variables `next` and `cost` is made by constraint (6.9). Thus far we have only considered the TSP part of the problem. The scheduling part of the problem, represented by the `start` variables, is linked to the TSP part by the constraints (6.10). Finally, constraints (6.11) ensure that the time windows are respected.

6.4.2 Integer Linear Programming Model

Let $G = (V, A)$ be a digraph with vertex set $V = \{0, 1, \ldots, n\}$. Vertex 0 represents the city from which we start the tour and in which we end. Vertices $1, \ldots, n$ represent the cities to be visited. To G, we apply a cost function $c : A \to \mathbb{Q}_+$.

To model the TSP as an integer linear programming model, we make use of a classical formulation. For each pair of cities $i, j \in V$, we introduce a variable $x_{ij} \in \{0, 1\}$. If $x_{ij} = 1$, the arc from i to j is included in the tour, otherwise $x_{ij} = 0$. We also introduce a cost variable z that represents the objective function, i.e. the cost of the tour, to be minimized.

With these variables, we model the TSP as

$$\min z = \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} \tag{6.12}$$

$$\text{subject to} \quad \sum_{i \in V} x_{ij} = 1 \qquad \forall j \in V, \tag{6.13}$$

$$\sum_{j \in V} x_{ij} = 1 \qquad \forall i \in V, \tag{6.14}$$

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \ \forall S \subset V, S \neq \varnothing. \tag{6.15}$$

The constraints (6.13) and (6.14) ensure that we visit and leave each city exactly once. They correspond to the `alldifferent` constraint (6.7) of the constraint programming model of the ATSPTW. To forbid subtours, we impose the *subtour elimination constraints* (6.15). For each nonempty subset of vertices $S \subset V$, a subtour elimination constraint ensures that there is at least one arc going from S to $V \setminus S$. These constraints correspond to the `nocycle` constraint (6.8) of the constraint programming model of the ATSPTW. Note that there exponentially many subtour elimination constraints, which makes this model impractical to apply.

6.4.3 Linear Programming Relaxation

Let $G = (V, A)$ be the digraph from the previous section, with cost function $c : A \to \mathbb{Q}_+$.

A linear programming relaxation for both the ATSPTW and the TSP is obtained from the integer linear programming formulation of the TSP. From this model, we first remove the subtour elimination constraints (6.15). Secondly, we relax the integrality constraints on the variables. The resulting model is called the *assignment problem*, or AP in short:

$$\begin{aligned}
\min z \ &= \textstyle\sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} && && (6.16)\\
\text{subject to} \quad & \textstyle\sum_{i \in V} x_{ij} = 1 && \forall j \in V, && (6.17)\\
& \textstyle\sum_{j \in V} x_{ij} = 1 && \forall i \in V, && (6.18)\\
& x_{ij} \geq 0 && \forall i, j \in V. && (6.19)
\end{aligned}$$

An important aspect of the AP is that an optimal integer solution can be computed in $O(n^3)$ time by the so-called *Hungarian method*; see for example Schrijver [2003, p. 188]. This method is also incremental, i.e. each recomputation requires at most $O(n^2)$ time; see Carpaneto, Martello, and Toth [1988]. Moreover, the reduced cost values can be obtained without extra computational effort. All together, this makes the AP particularly useful for our purposes.

Next we establish the mapping between the constraint programming model for the ATSPTW and the AP:

$$\begin{aligned}
\texttt{next}_i = j &\Leftrightarrow\ x_{ij} = 1,\\
\texttt{next}_i \neq j &\Leftrightarrow\ x_{ij} = 0,
\end{aligned}$$

for all $i, j \in V$. Note that the vertex set V used for the AP model does not contain the vertex $n{+}1$ which does occur in the constraint programming model for the ATSPTW. This is not a problem, because vertex $n + 1$ is identified with vertex 0.

A solution to the AP provides a lower bound for the original problem and reduced costs $\overline{c}$. So this linear programming relaxation as well as its mapping with the constraint programming model follows the description of our method as presented in Section 6.2.1.

In general, the AP solution does not provide a tight lower bound. To improve the bound, we can add linear constraints to the AP that are valid for the ATSPTW or the TSP, so-called *cutting planes*. Many different kinds of cutting planes for these problems have been proposed. Following Focacci, Lodi, and Milano [2002], we use *subtour elimination cuts* for the TSP, which can be found in polynomial time. However, adding linear inequalities to the AP formulation changes the structure of the relaxation, which is no longer an AP. Because of the benefits of the AP structure we choose to maintain it by relaxing the cuts in a Lagrangian way, as done by Focacci, Lodi, and Milano [2002]. This means that a cut is added to the objective function with an associated "penalty" cost. The resulting relaxation still has an AP structure, but provides a tighter lower bound than the initial AP.

6.5 Computational Results

6.5.1 Implementation

We have implemented and tested our method using ILOG Solver 4.4 and ILOG Scheduler 4.4 as constraint programming solvers, and ILOG Cplex 6.5 as linear programming solver. Our experiments are performed on a PC Pentium 1GHz with 256 MB RAM. The test instances are taken from the TSPLIB by Reinelt [1991][2] and the collection of ATSPTW instances by Ascheuer [1995][3].

Our implementation is built on the work by Focacci, Lodi, and Milano [2002]. Their code is implemented using the same models and relaxations that have been described in this chapter. To this code we added the ingredients of our method:

i) the definition of the good domains based on reduced costs,
ii) the generation of subproblems using the discrepancy constraint, and
iii) the discrepancy-based improved bounding technique.

All subproblems have been solved using the original code of Focacci, Lodi, and Milano [2002].

The goal of the experiments is to analyze the quality of reduced costs as value ordering heuristic, and to analyze the efficiency of our solution framework.

6.5.2 Quality of Heuristic

In this section we evaluate the quality of reduced costs as branching heuristic. In Section 5.4.1 we already observed that reduced costs are indeed very accurate for TSP instances: even when we only included the values corresponding

[2] See `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.
[3] See `http://elib.zib.de/pub/Packages/mp-testdata/tsp/atsptw/`.

to the lowest reduced costs in the good domain we already found an optimal solution in subproblems corresponding to discrepancy 0 or 1. However, in Section 5.4.1 the optimal objective value was given in advance and we did not need to prove optimality. In this chapter we do need to take into account the proof of optimality.

On the one hand, we would like the optimal solution to be in the first subproblem, corresponding to discrepancy 0. This subproblem should be as small as possible, in order to be able to solve it fast. On the other hand, we need to prove fast the optimality of the solution found. For this we need relatively large reduced costs in the bad domains, in order to apply Theorem 6.2 effectively. This would typically induce a larger first subproblem. Consequently, we should make a trade-off between finding a good first solution and proving optimality. This is done by tuning the size of the good domains.

We tune the size of the good domains by specifying a preferred ratio r with $0 \leq r \leq 1$, such that $|D_i^{\text{good}}| \approx r \cdot n$ for $i = 1, \ldots, n$. The good domains are computed at the root node of tree search tree. First we solve the AP relaxation, and apply constraint propagation until we have reached the specified local consistency for each constraint. During this process the AP solution is updated, if necessary. Then we select, for each domain D_i, the values corresponding to smallest reduced costs and insert them in D_i^{good}. This is done until $|D_i^{\text{good}}| \geq r \cdot n$. We add all values corresponding to possible ties, hence the actual size of D_i^{good} may be larger than $r \cdot n$.

We have tuned the "optimal" value of ratio r for TSP and ATSPTW instances. For increasing values of r, we have collected various problem characteristics:

- the average relative size of the good domains (after propagation), i.e. $\frac{1}{n} \sum_{i=1}^{n} |D_i^{\text{good}}|/|D_i|$,
- the average discrepancy of the subproblem that contains the optimal solution,
- the average discrepancy of the proof of optimality with respect to Theorem 6.2,
- the average total number of backtracks.

The aim is to find a smallest ratio that has a satisfactory effect on these characteristics.

For the TSP, we have solved the instances gr17, gr21, gr24, fri26, bayg29 and bays29 from TSPLIB to optimality for $r = 0.025, 0.05, 0.015, 0.1$. The average results are reported in Figure 6.3. The average relative size of the good domains is depicted in the upper-left figure. For example, for $r = 0.025$, the average size of a good domain is 17% of the initial domain (after propagation). The upper-right figure reports the average discrepancy of the subproblem that contains the optimal solution. For ratio $r \geq 0.05$, we always find an optimal solution in the first subproblem. In the lower-left figure we see that we prove optimality of this solution in almost all cases immediately (i.e. the discrepancy is close to 1) for $r \geq 0.05$. For $r \geq 0.075$ this property holds in all cases. The

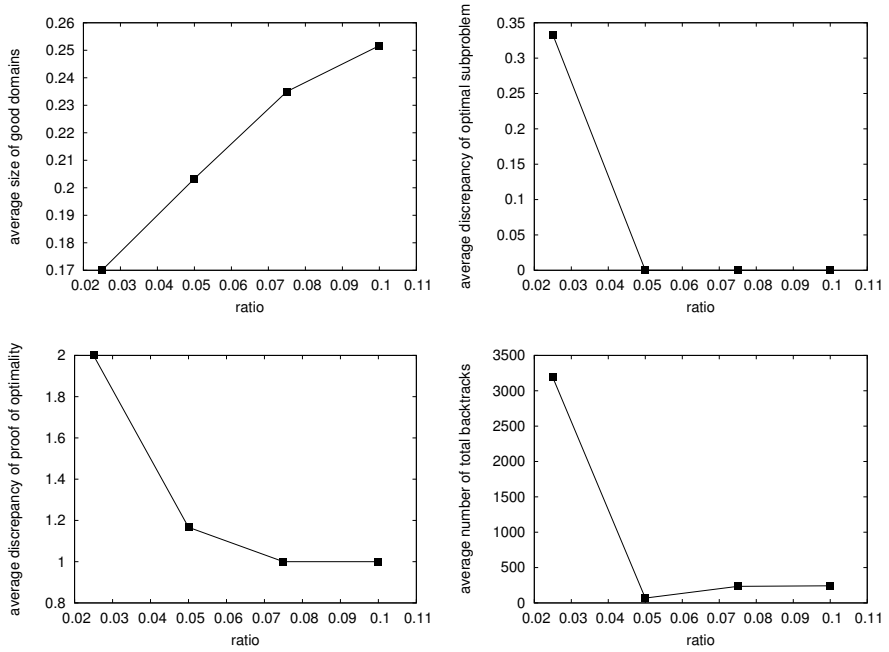


Figure 6.3. Quality of reduced costs as branching heuristic with respect to the ratio for the TSP.

figure in the lower-right corner depicts the average total number of backtracks. Here we clearly see that a smaller ratio has a negative effect on the solution process. Namely, for $r = 0.025$ the average number of total backtracks is much larger than for $r \geq 0.05$. This is mainly due to the weaker proof of optimality. From these figures we conclude that $0.05 \leq r \leq 0.075$ is a good ratio for the TSP.

For the ATSPTW, we have solved the instances rbg010a, rbg016a, rbg016b, rbg017.2, rbg017, rbg017a, rbg019a, rbg019b, rbg019c, rbg019d, rbg020a, rbg021.2, rbg021.3, rbg021.4, rbg021.5, rbg021.6, rbg021.7, rbg021.8, rbg021.9, rbg021 and rbg027a from Ascheuer [1995] to optimality for $r = 0.05$, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35. The average results are reported in Figure 6.4 which follows the same format as Figure 6.3. A remarkable difference between ATSPTW and TSP instances can be observed concerning the average relative size of the good domains. This difference is due to the addition of scheduling constraints. The propagation of these constraints often remove a large number of domain values. Therefore the good domains may be relatively large. For example, for the ATSPTW the average relative size of the good domains ranges from 0.57 to 0.95 for $r = 0.05$ to 0.35 as depicted in the upper-left figure of Figure 6.4. Nevertheless, the resulting subproblems are small enough to solve, using only a small number of backtracks; see the lower-right figure. From these figures

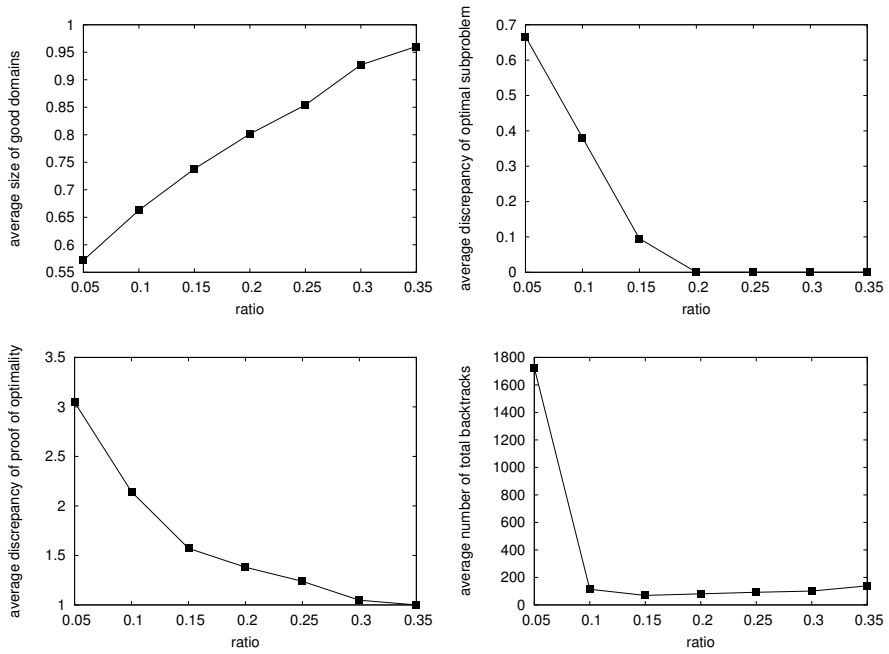


Figure 6.4. Quality of reduced costs as branching heuristic with respect to the ratio for the ATSPTW.

we conclude that $0.15 \leq r \leq 0.2$ is a good ratio for the ATSPTW. For this ratio we find an optimal solution in the first subproblem in almost all cases, and optimality is proved immediately (discrepancy close to 1) in many cases.

6.5.3 Symmetric TSP Instances

We have tested our method on a set of symmetric TSP instances from TSPLIB. It should be noted however, that constraint programming is not the preferred method for these problems. Specific TSP solvers are much faster than solvers based on constraint programming. However, it is still interesting to evaluate the performance of different constraint programming methods on pure TSP instances. Namely, most real-life applications do not consist only of a "pure" combinatorial structure, but also contain side constraints that make the problem much more difficult to solve. For example, the ATSPTW cannot be solved with a pure TSP solver. On the other hand, using a constraint programming solver, one "just" needs to add the additional side constraints while the solution process remains the same. Hence, when the problem at hand contains a TSP structure and is suitable for a constraint programming, one typically wants to apply the constraint programming approach that performs well on pure TSP instances.

instance		Concorde	FLM2002		our method	
name	n	time	time	fails	time	fails
gr17	17	0.04	0.12	34	0.12	1
gr21	21	0.01	0.07	19	0.06	1
gr24	24	0.02	0.18	29	0.18	1
fri26	26	0.03	0.17	70	0.17	4
bayg29	29	0.05	0.33	102	0.28	28
bays29	29	0.04	0.30	418	0.20	54
dantzig42*	42	0.09	1.43	524	1.21	366
hk48	48	0.08	12.61	15k	1.91	300
gr48*	48	0.18	21.05	25k	19.10	22k
brazil58	58	0.24	N.A.	N.A.	81.19	156k

* ratio $r = 0.075$.

Table 6.2. Computational results for solving TSP instances on n cities to optimality. Time is measured in seconds. N.A. means "not applicable" due to time limit (300 s).

We compare our method with the performance of the method by Focacci, Lodi, and Milano [2002]. This has two reasons. The first reason is that the method by Focacci, Lodi, and Milano [2002] is the fastest known constraint programming approach for pure TSP instances, to our knowledge. The second reason is that our method is an adaption of the method by Focacci, Lodi, and Milano [2002], which makes it possible to evaluate the results in view of these adaptations. In other words, all positive or negative observations are due to our proposed method. The results of Focacci, Lodi, and Milano [2002] were obtained by executing their code on the same machine as our method.

For completeness, we also compare our method with the state-of-the-art solver for pure TSP instances, Concorde, by D. Applegate, R. Bixby, W. Cook and V. Chvátal. The reason is to show the difference between a constraint programming approach and specialized solvers for the TSP. The results for Concorde were obtained on a PC Intel Xeon 2.8GHz with 2 GB memory[4].

The results are reported in Table 6.2. The three approaches are indicated by 'Concorde' for the pure TSP solver, 'FLM2002' for the method by Focacci, Lodi, and Milano [2002] and 'our method' for the method proposed in this chapter. The results for our method are obtained with ratio $r = 0.05$, except for two instances for which ratio $r = 0.075$ was considerably better. For each instance, n denotes the number of cities. We report the total solution time in seconds to find a solution and to prove optimality, while 'fails' indicates the total number of backtracks. The time limit is set to 300 seconds.

The results show that our method improves the method of Focacci, Lodi, and Milano [2002], both concerning the total number of backtracks and the time. For larger instances the difference is most obvious. In the next section we will see whether this improvement extrapolates to ATSPTW instances.

A last remark concerns the comparison with the results of Section 5.4.1, where we applied a similar method on TSP instances without proving optimality. Consider for example instance 'gr17' in Table 5.1 and Table 6.2. It

[4] See `http://www.tsp.gatech.edu/concorde.html`.

seems strange that we can find a solution and prove optimality in 0.12 seconds using 1 backtrack (Table 6.2), while we find a single optimal solution without proving optimality in 0.02 seconds using 3 backtracks (Table 5.1). This difference is caused by the different application of reduced costs as branching heuristic. In the present case we have applied the reduced cost-based value ordering *statically*. It means that the good domains are selected at the root and fixed throughout the whole search tree. This is necessary to apply the `discrepancy` constraint and the discrepancy-based bounding technique, i.e. Theorem 6.2. In Section 5.4.1 we have applied the reduced cost-based value ordering *dynamically*. A dynamic value ordering heuristic takes into consideration the current status of a domain. This explains the difference in results for instance 'gr17'.

6.5.4 Asymmetric TSP with Time Windows Instances

For the ATSPTW, we have compared our method with the approaches that are currently the fastest for these instances, to our knowledge. The first is the method of Balas and Simonetti [2000], which applies a dynamic programming algorithm. Their results are obtained using a Sun Ultra I Workstation with 64MB memory. The second is the method of Ascheuer, Fischetti, and Grötschel [2001], which is a branch-and-cut approach. Their results are obtained using a Sun Sparc Station 10. The third is the method by Focacci, Lodi, and Milano [2002] on which we have based our method. Their results are obtained using a PC Pentium III 700MHz with 128MB memory. For our method, we have applied a ratio $r = 0.15$ in all cases, following the analysis in Section 6.5.2.

The results for the ATSPTW instances are shown in Table 6.3. It follows the same format as Table 6.2. The method of Balas and Simonetti [2000] is indicated by 'BS2000', the method of Ascheuer, Fischetti, and Grötschel [2001] is indicated by 'AFG2001', the method of Focacci, Lodi, and Milano [2002] is indicated by 'FLM2002' while 'our method' indicates the method we propose in this chapter. We have only included instances for which at least one of the methods could find a provably optimal solution. One should take into account that the results are obtained on faster machines from left to right. The time limit for FLM2002 and our method is 1800 seconds, which compares to the time limit of 5 hours on the machine of AFG2001. The time limit of BS2000 is unknown.

In general, our method behaves quite well. It improves the method of Focacci, Lodi, and Milano [2002] in almost all cases, both concerning the total number of backtracks and the total solution time. This indicates that the results for the pure TSP instances of the previous section indeed do extrapolate to ATSPTW instances. Our method also outperforms the method by Ascheuer, Fischetti, and Grötschel [2001] in many cases. However, the method by Balas and Simonetti [2000] is particularly effective on larger instances. This is where the dynamic programming algorithm approach pays off. It is

instance		BS2000	AFG2001	FLM2002		our method	
name	n	time	time	time	fails	time	fails
rbg010a	12	0.95	0.12	0.0	6	0.03	4
rbg016a	18	1.72	0.20	0.1	21	0.04	8
rbg016b	18	2.71	8.80	0.1	27	0.09	32
rbg017.2	17	9.80	0.03	0.0	17	0.04	2
rbg017	17	2.51	0.82	0.1	27	0.08	11
rbg017a	19	3.23	0.12	0.1	22	0.05	1
rbg019a	21	2.03	0.03	0.0	14	0.05	2
rbg019b	21	3.42	54.67	0.2	80	0.09	22
rbg019c	21	7.64	8.72	0.3	81	0.14	74
rbg019d	21	2.39	0.75	0.0	32	0.07	4
rbg020a	22	3.59	0.20	0.0	9	0.06	3
rbg021.2	21	9.00	0.22	0.2	44	0.08	15
rbg021.3	21	9.60	27.15	0.4	107	0.14	80
rbg021.4	21	11.52	5.82	0.3	121	0.09	32
rbg021.5	21	127.97	6.63	0.2	55	0.12	60
rbg021.6	21	161.66	1.38	0.7	318	0.16	50
rbg021.7	21	N.A.	4.30	0.6	237	0.21	43
rbg021.8	21	N.A.	17.40	0.6	222	0.10	27
rbg021.9	21	N.A.	26.12	0.8	310	0.11	28
rbg021	21	7.82	8.75	0.3	81	0.14	74
rbg027a	29	N.A.	2.25	0.2	50	0.16	23
rbg031a	33	11.13	1.70	2.7	841	0.68	119
rbg033a	35	5.66	1.85	1.0	480	0.73	55
rbg034a	36	18.03	0.98	55.2	13k	0.93	36
rbg035a.2	37	N.A.	64.80	36.8	5k	8.18	4k
rbg035a	37	7.67	1.83	3.5	841	0.83	56
rbg038a	40	8.64	4232.23	0.2	49	0.36	3
rgb040a	42	20.08	751.82	738.1	136k	1200.62	387k
rbg041a	43	24.57	N.A.	N.A.		N.A.	
rbg042a	44	47.38	N.A.	149.8	19k	70.71	24k
rbg050a	52	N.A.	18.62	180.4	19k	4.21	1.5k
rbg055a	57	25.56	6.40	2.5	384	4.50	133
rbg067a	69	29.14	5.95	4.0	493	25.69	128
rbg086a	88	18.70	N.A.	N.A.		N.A.	
rbg092a	94	48.13	N.A.	N.A.		N.A.	
rbg125a	127	31.93	229.82	N.A.		N.A.	
rbg132	132	1135.49	N.A.	N.A.		N.A.	
rbg152	152	37.90	N.A.	N.A.		N.A.	

Table 6.3. Computational results for solving ATSPTW instances to optimality. Time is measured in seconds. N.A. means "not applicable" due to time limit.

less sensitive to the increase of the number of cities n while the other three approaches need to maintain search trees that become too large for larger n.

6.6 Discussion and Conclusion

We have applied reduced costs as branching heuristic in a constraint programming search tree. The reduced costs are obtained from a linear programming relaxation of the COP. To improve the bound of this relaxation we have used the domain partitioning framework of Chapter 5 in combination with a discrepancy constraint.

Computational results on TSP and asymmetric TSPTW instances have shown that reduced costs are very accurate. In almost all cases an optimal solution has been found in the first subproblem of the partitioning framework. Also the bound improvement has been shown to be effective. In many cases we were able to proof optimality after generating only a small number of subproblems.

The method proposed in this chapter is a tight combination of operations research techniques and constraint programming. It shows that cross-fertilization of the two fields can be very beneficial. Not only do reduced costs provide an accurate value ordering heuristic, it is also possible to improve the bound computations of operations research based using constraint programming, i.e. the discrepancy constraint.

A possible direction for further research is the following. We have explicitly focused on reduced costs, and therefore our value ordering heuristic is optimization-driven. On the other hand, one could also infer relaxations from the scheduling component of the problem and apply these as a value ordering heuristic. For several problem instances it is likely that such heuristics would be more effective.

Chapter 7

Semidefinite Relaxation as Branching Heuristic

In this chapter we investigate the applicability of semidefinite relaxations in constraint programming. We use the solution of a semidefinite relaxation of a CSP to guide the traversal of the search tree. Furthermore, the (often tight) bound of the semidefinite relaxation helps to prune suboptimal branches. Experimental results on stable set problem instances and maximum clique problem instances show that constraint programming can indeed benefit from semidefinite relaxations.

7.1 Introduction

In this chapter we investigate the possibility of using semidefinite relaxations in constraint programming. This investigation involves the extraction of semidefinite relaxations from a constraint programming model, and the actual use of the relaxation inside the solution scheme. We propose to use the solution of a semidefinite relaxation to define variable and value ordering heuristics in combination with limited discrepancy search (LDS). Effectively, this means that our search starts at the suggestion made by the semidefinite relaxation, and gradually scans a wider area around this solution. Moreover, we use the solution value of the semidefinite relaxation as a bound for the objective function, which results in stronger pruning of suboptimal branches. By applying a semidefinite relaxation in this way, we hope to speed up the constraint programming solver significantly.

We implemented our method and provide experimental results on the stable set problem and the maximum clique problem, two classical combinatorial optimization problems. We compare our method with a standard constraint programming solver, and with specialized solvers for maximum clique problems. As computational results will show, our method obtains far better results than a standard constraint programming solver. However, on maximum clique problems, the specialized solvers appear to be much faster than our method.

The outline of this chapter is as follows. The next section gives a motivation for the approach that we propose. Then, in Section 7.3, we present a description of our solution framework. In Section 7.4 we formulate the stable set problem as an integer optimization problem and provide a semidefinite relaxation. Section 7.5 presents the computational results. In Section 7.6 we discuss our results and give a conclusion.

7.2 Motivation

NP-hard combinatorial optimization problems are often solved with the use of a polynomially solvable relaxation. Often (continuous) linear relaxations are chosen for this purpose. Also within constraint programming, linear relaxations are widely used, see Focacci, Lodi, and Milano [2003] for an overview. For example, we have exploited a linear relaxation in a constraint programming framework in Chapter 6.

Let us first motivate why in this paper a semidefinite relaxation is used rather than a linear relaxation. For some problems, for example for the stable set problem, standard linear relaxations are not very tight and not informative. One way to overcome this problem is to identify and add linear constraints that strengthen the relaxation. But it may be time-consuming to identify such constraints, and by enlarging the model the solution process may slow down.

On the other hand, several papers on approximation theory, following Goemans and Williamson [1995], have shown the tightness of semidefinite relaxations. However, being tighter, semidefinite programs are more time-consuming to solve than linear programs in practice. Hence one has to trade strength for computation time. For some (large scale) applications, semidefinite relaxations are well-suited to be used within a branch and bound framework (see for example Karisch, Rendl, and Clausen [2000]). Moreover, our intention is not to solve a relaxation at every node of the search tree. Instead, we propose to solve only once a relaxation, before entering the search tree. Therefore, we are willing to make the trade-off in favour of the semidefinite relaxation.

Finally, investigating the possibility of using semidefinite relaxations in constraint programming is worthwhile in itself. To our knowledge the cross-fertilization of semidefinite programming and constraint programming has not yet been investigated. Hence, the work in this chapter should be seen as a first step toward the cooperation of constraint programming and semidefinite programming.

7.3 Solution Framework

The skeleton of our solution framework is formed by the search tree of the constraint programming solver. Within this skeleton, we want to use the solution of a semidefinite relaxation to define the variable and value ordering heuristics. In this section we first show how to extract a semidefinite relaxation from a constraint programming model. Then we give a description of the usage of the relaxation within the enumeration scheme.

7.3.1 Building a Semidefinite Relaxation

Consider a COP consisting of a sequence of variables $v_1, \ldots, v_n$, corresponding finite domains $D_1, \ldots, D_n$, a set of constraints and an objective function. From

this model we need to extract a semidefinite relaxation. In general, a relaxation is obtained by removing or replacing one or more constraints such that all solutions are preserved. If it is possible to identify a subset of constraints for which a semidefinite relaxation is known, this relaxation can be used inside our framework.

Unfortunately, it is not a trivial task to obtain a computationally efficient semidefinite program that provides a tight solution for a given problem. However, for a number of combinatorial optimization problems such semidefinite relaxations do exist, for example the stable set problem, the maximum cut problem, quadratic programming problems, the maximum satisfiability problem, and many other problems; see Laurent and Rendl [2004] for an overview. If such semidefinite relaxations are not available, we need to build up a relaxation from scratch. This can be done in the following way.

If all domains $D_1, \ldots, D_n$ are binary, a semidefinite relaxation can be extracted using a method proposed by Laurent, Poljak, and Rendl [1997], which is explained below. In general, however, the domains are non-binary. In that case, we transform the variables v_i and the domains D_i into corresponding binary variables x_{ij} for $i = 1, \ldots, n$ and $j \in D_i$, similar to the mapping defined in Section 6.2.1:

$$\begin{aligned} v_i = j &\Leftrightarrow x_{ij} = 1, \\ v_i \neq j &\Leftrightarrow x_{ij} = 0. \end{aligned} \tag{7.1}$$

We will then use the binary variables x_{ij} to construct a semidefinite relaxation. Of course, the transformation has consequences for the constraints also, which will be discussed below.

The method to transform a model with binary variables into a semidefinite relaxation, presented by Laurent, Poljak, and Rendl [1997], is the following. Let $d \in \{0,1\}^N$ be a vector of binary variables, where N is a positive integer. Construct the $(N+1) \times (N+1)$ variable matrix X as

$$X = \begin{pmatrix} 1 \\ d \end{pmatrix} (1 \; d^{\mathsf{T}}) = \begin{pmatrix} 1 & d^{\mathsf{T}} \\ d & dd^{\mathsf{T}} \end{pmatrix}.$$

Then X can be constrained to satisfy

$$X \succeq 0 \tag{7.2}$$

$$X_{ii} = X_{0i} \; \forall i \in \{1, \ldots, N\} \tag{7.3}$$

where the rows and columns of X are indexed from 0 to N. Condition (7.3) expresses the fact that $d_i^2 = d_i$, which is equivalent to $d_i \in \{0,1\}$. Note however that the latter constraint is relaxed by requiring X to be positive semidefinite.

The matrix X contains the variables to model our semidefinite relaxation. Obviously, the diagonal entries (as well as the first row and column) of this matrix represent the binary variables from which we started. Using these variables, we need to rewrite (a part of) the original constraints into the form of program (2.3) in order to build the semidefinite relaxation.

In case the binary variables are obtained from transformation (7.1), not all constraints may be trivially transformed accordingly. Especially because the original constraints may be of any form. The same holds for the objective function. On the other hand, as we are constructing a relaxation, we may choose among the set of constraints an appropriate subset to include in the relaxation. Moreover, the constraints itself are allowed to be relaxed. Although there is no 'recipe' to transform any given original constraint into the form of program (2.3), one may use results from the literature; see for example Laurent and Rendl [2004]. For example, for linear constraints on binary variables a straightforward translation is given in Section 2.1.3.

7.3.2 Applying the Semidefinite Relaxation

At this point, we have either identified a subset of constraints for which a semidefinite relaxation exists, or built up our own relaxation. Now we show how to apply the solution to the semidefinite relaxation inside the constraint programming framework, also depicted in Figure 7.1. In general, the solution to the semidefinite relaxation yields fractional values for its variable matrix. For example, the diagonal variables X_{ii} of the above matrix will be assigned to a value between 0 and 1. These fractional values serve as an indication for the original constraint programming variables. Consider for example again the above matrix X, and suppose it is obtained from non-binary original variables, by transformation (7.1). Assume that variable X_{ii} corresponds to the binary variable x_{jk} (for some integer j and k), which corresponds to $v_j = k$, where v_j is a constraint programming variable and $k \in D_j$. If variable X_{ii} is close to 1, then also x_{jk} is supposed to be close to 1, which corresponds to assigning $v_j = k$.

Hence, our variable and value ordering heuristics for the constraint programming variables are based upon the fractional solution values of the corresponding variables in the semidefinite relaxation. Our variable ordering heuristic is to select first the constraint programming variable for which the corresponding fractional solution is closest to the corresponding integer solution. Our value ordering heuristic is to select first the corresponding suggested value. For example, consider again the above matrix X, obtained from non-binary variables by transformation (7.1). We select first the variable v_j for which X_{ii}, representing the binary variable x_{jk}, is closest to 1, for some $k \in D_j$ and corresponding integer i. Then we assign value k to variable v_j. We have also implemented a randomized variant of the above variable ordering heuristic. In the randomized case, the selected variable is accepted with a probability proportional to the corresponding fractional value.

We expect the semidefinite relaxation to provide promising values. Therefore the resulting search tree will be traversed using limited discrepancy search, defined in Section 2.2.3. A last remark concerns the solution value of the semidefinite relaxation, which is used as a bound on the objective function in the constraint programming model. If this bound is tight, which is the

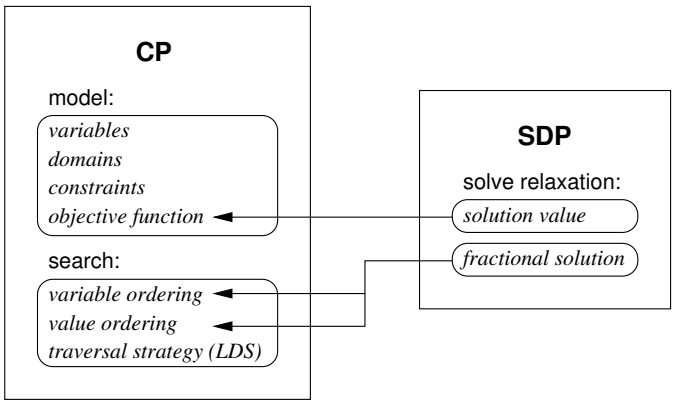


Figure 7.1. Communication between constraint programming (CP) and semidefinite programming (SDP).

case in our experiments, it leads to more propagation and a smaller search space.

7.4 The Stable Set Problem

We have applied our method to the stable set problem and the maximum clique problem (these problems are defined in Section 2.1.1). Recall that a maximum clique problem can be translated into a stable set problem in the complement graph. We will do exactly this in our implementation, and focus on the stable set problem, for which tight semidefinite relaxations exist. This section describes the models and the semidefinite relaxation of the stable set problem that we have used.

7.4.1 Integer Programming Models

We first consider an integer linear programming formulation of the stable set problem. Recall that we denote the value of a maximum-weight stable set in a graph G by $\alpha(G)$.

Consider a graph $G = (V, E)$ where $V = \{1, \ldots, n\}$, with "weight" function $w : E \to \mathbb{Q}$. Without loss of generality, we can assume all weights to be nonnegative. We introduce binary variables to indicate whether or not a vertex belongs to the stable set S in G. So, for n vertices, we have n integer variables x_i indexed by $i \in V$, with initial domains $\{0, 1\}$. In this way, $x_i = 1$ if vertex i is in S, and $x_i = 0$ otherwise. We state the objective function, being the sum of the weights of vertices that are in S, as $\sum_{i=1}^{n} w_i x_i$. Finally, we define the

constraints that forbid two adjacent vertices to be both inside S as $x_i+x_j \leq 1$, for all edges $(i,j) \in E$. Hence the integer linear programming model becomes:

$$\begin{aligned} \alpha(G) = \max \ & \textstyle\sum_{i=1}^{n} w_i x_i \\ \text{s.t. } & x_i + x_j \leq 1 \ \forall (i,j) \in E \\ & x_i \in \{0,1\} \ \forall i \in V. \end{aligned} \tag{7.4}$$

Another way of describing the same solution set is presented by the following integer quadratic program

$$\begin{aligned} \alpha(G) = \max \ & \textstyle\sum_{i=1}^{n} w_i x_i \\ \text{s.t. } & x_i x_j = 0 \ \forall (i,j) \in E \\ & x_i^2 = x_i \ \forall i \in V. \end{aligned} \tag{7.5}$$

Note that here the constraint $x_i \in \{0,1\}$ is replaced by $x_i^2 = x_i$, similar to condition (7.3) in Section 7.3. This quadratic formulation will be used below to infer a semidefinite relaxation of the stable set problem.

In fact, both model (7.4) and model (7.5) can be used as a constraint programming model. We have chosen the first model, because the quadratic constraints take more time to propagate than the linear constraints, while the propagation is the same. To infer the semidefinite relaxation, however, we will use the equivalent model (7.5).

7.4.2 Semidefinite Programming Relaxation

The integer quadratic program (7.5) gives rise to a well-known semidefinite relaxation introduced by Lovász [1979]. The value of the objective function of this relaxation has been named the *theta number* of a graph G, indicated by $\vartheta(G)$. For its derivation into a form similar to program (2.3), we will follow the same idea as in Section 7.3 for the general case.

As our constraint programming model uses binary variables already, we can immediately define the $(n+1)\times(n+1)$ matrix variable X of our relaxation as

$$X = \begin{pmatrix} 1 & x^{\mathsf{T}} \\ x & xx^{\mathsf{T}} \end{pmatrix}$$

where the binary vector x again represents the stable set, as in Section 7.4.1. First we impose the constraints

$$X \succeq 0 \tag{7.6}$$

$$X_{ii} = X_{0i} \ \forall i \in \{1, \ldots, n\} \tag{7.7}$$

as described in Section 7.3. Next we translate the edge constraints $x_i x_j = 0$ from program (7.5) into $X_{ij} = 0$, because X_{ij} represents $x_i x_j$. In order to translate the objective function, we first define the $(n+1) \times (n+1)$ weight matrix W as

$$W_{ii} = w_i \ \forall i \in \{1, \ldots, n\},$$
$$W_{ij} = 0 \quad \forall i \in \{0, \ldots, n\}, j \in \{0, \ldots, n\}, i \neq j.$$

Then the objective function translates into tr(WX). The semidefinite relaxation thus becomes

$$\begin{aligned} \vartheta(G) = \max \quad & \mathrm{tr}(\mathrm{WX}) \\ \text{s.t. } & X_{ii} = X_{0i} \ \forall i \in V \\ & X_{ij} = 0 \quad \forall (i,j) \in E \\ & X \succeq 0. \end{aligned} \tag{7.8}$$

Note that program (7.8) can easily be rewritten into the general form of program (2.3). Namely, $X_{ii} = X_{0i}$ is equal to $\mathrm{tr}(A_i X) = 0$ where the $(n+1) \times (n+1)$ matrix A_i consists of all zeroes, except for $(A_i)_{ii} = 1$, $(A_i)_{i0} = -\frac{1}{2}$ and $(A_i)_{0i} = -\frac{1}{2}$, which makes the corresponding right-hand side (b_i entry) equal to 0 (similarly for the edge constraints).

The theta number also arises from other formulations, different from the above, see Grötschel, Lovász, and Schrijver [1988]. In our implementation we have used the formulation that has been shown to be computationally most efficient among those alternatives as shown by Gruber and Rendl [2003]. Let us introduce that particular formulation (called ϑ_3 by Grötschel, Lovász, and Schrijver [1988]). Again, let $x \in \{0,1\}^n$ be a vector of binary variables representing a stable set. Define the $n \times n$ matrix $\tilde{X} = \xi\xi^{\mathsf{T}}$ where

$$\xi_i = \frac{\sqrt{w_i}}{\sqrt{\sum_{j=1}^n w_j x_j}} x_i.$$

Furthermore, let the $n \times n$ cost matrix U be defined as $U_{ij} = \sqrt{w_i w_j}$ for $i, j \in V$. Observe that in these definitions we exploit the fact that $w_i \geq 0$ for all $i \in V$. The following semidefinite program

$$\begin{aligned} \max \quad & \mathrm{tr}(U\tilde{X}) \\ \text{s.t. } & \mathrm{tr}(\tilde{X}) = 1 \\ & \tilde{X}_{ij} = 0 \ \forall (i,j) \in E \\ & \tilde{X} \succeq 0 \end{aligned} \tag{7.9}$$

has been shown to also give the theta number of G, see Grötschel, Lovász, and Schrijver [1988]. When (7.9) is solved to optimality, the diagonal element $\tilde{X}_{ii}$ serves as an indication for the value of x_i ($i \in V$) in a maximum stable set. Again, it is not difficult to rewrite program (7.9) into the general form of program (2.3).

Program (7.9) uses matrices of dimension n and $m+1$ constraints, while program (7.8) uses matrices of dimension $n+1$ and $m+n$ constraints. This gives an indication why program (7.9) is computationally more efficient.

7.5 Computational Results

We have tested our method on random and structured instances representing stable set and maximum clique problems. The goal of our experiments is to verify whether a semidefinite relaxation is applicable in constraint programming. Hence, we first analyze problem characteristics to identify possibly suitable instances. Then we compare our method with a standard constraint programming approach and specialized solvers for maximum-clique problems.

7.5.1 Implementation

All our experiments are performed on a Sun Enterprise 450 (4 X UltraSPARC-II 400MHz) with maximum 2048 Mb memory size, on which our algorithms only use one processor of 400MHz at a time. As constraint programming solver we use ILOG Solver 5.1. As semidefinite programming solver, we use CSDP version 4.1 by Borchers [1999], with the optimized ATLAS 3.4.1 [Whaley, Petitet, and Dongarra, 2001] and LAPACK 3.0 [Anderson, Bai, Bischof, Blackford, Demmel, Dongarra, Croz, Greenbaum, Hammarling, McKenney, and Sorensen, 1999] libraries for matrix computations. The reason for our choices is that both solvers are among the fastest in their field, and because ILOG Solver is written in C++, and CSDP is written in C, they can be hooked together relatively easy.

We distinguish two algorithms to perform our experiments. The first algorithm is a sole constraint programming solver, which uses a standard enumeration strategy. This means we use a lexicographic variable ordering, and we select domain value 1 before value 0. The resulting search tree is traversed using a depth-first search strategy. After each branching decision, its effect is directly propagated through the constraints. As constraint programming model we have used model (7.4), as was argued in Section 7.4.1.

The second algorithm is the one proposed in Section 7.3. It first solves the semidefinite program (7.9), and then calls the constraint programming solver. In this case, we use the randomized variable ordering heuristic, defined by the solution of the semidefinite relaxation. The resulting search tree is traversed using a limited discrepancy search strategy. In fact, in order to improve our starting solution, we repeat the search for the first solution n times, (where n is the number of vertices in the graph), and the best solution found is the heuristic solution to be followed by the limited discrepancy search strategy.

7.5.2 Characterization of Problem Instances

We will first identify general characteristics of the constraint programming solver and the semidefinite programming solver applied to stable set problems. It appears that both solvers are highly dependent on the edge density of the graph, i.e. $2m/(n^2-n)$ for a graph with n vertices and m edges. We therefore generated random graphs on 30, 40, 50 and 60 vertices, with density ranging

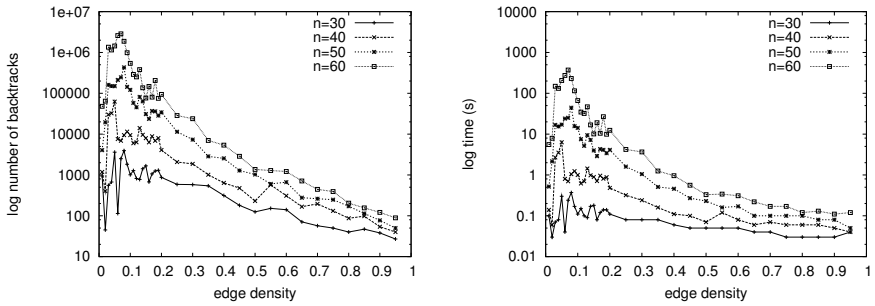


Figure 7.2. Performance of the constraint programming solver on random instances with n vertices.

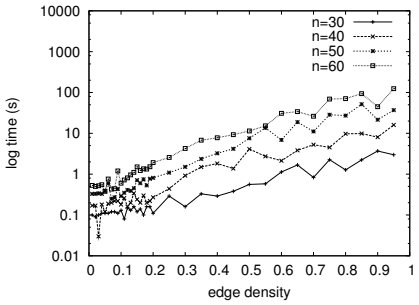


Figure 7.3. Performance of the semidefinite programming solver on random instances with n vertices.

from 0.01 up to 0.95. Our aim is to identify the hardness of the instances for both solvers, parametrized by the density. Based upon this information, we can identify the kind of problems our algorithm is suitable for.

We have plotted the performance of both solvers in Figure 7.2 and Figure 7.3. Here the constraint programming solver solves the problems to optimality, while the semidefinite programming solver only solves the semidefinite relaxation. For the constraint programming solver, we depict both the number of backtracks and the time needed to prove optimality. For the semidefinite programming solver we only plotted the time needed to solve the relaxation. Namely, this solver does not use a tree search, but a so-called primal-dual interior point algorithm. Note that we use a log-scale for time and number of backtracks in these pictures.

From these figures, we can conclude that the constraint programming solver has the most difficulties with instances up to density around 0.2. Here we see the effect of constraint propagation. As the number of constraints increases, the search tree can be heavily pruned. On the other hand, our semidefinite relaxation suffers from every edge that is added. As the density

increases, the semidefinite program increases accordingly, as well as its computation time. Fortunately, for the instances up to 0.2, the computation time for the semidefinite relaxation is very small. Consequently, our algorithm is expected to behave best for graphs that have edge density up to around 0.2. For graphs with a higher density, the constraint programming solver is expected to use less time than the semidefinite programming solver, which makes the application of our method unnecessary.

7.5.3 Random Weighted and Unweighted Graphs

Our first experiments are performed on random weighted and unweighted graphs. We generated graphs with 50, 75, 100, 125 and 150 vertices and edge density from 0.05, 010 and 0.15, corresponding to the interesting problem area. The results are presented in Table 3.1. Unweighted graphs on n vertices and edge density r are named 'gndr'. Weighted graphs are similarly named 'wg$n$dr'.

The first five columns of the table are dedicated to the instance, reporting its name, the number of vertices n and edges m, the edge density and the (best known) value of a stable set, α. The next three columns (CP) present the performance of the constraint programming solver, reporting the best found estimate of α, the total time and the total number of backtracks needed to prove optimality. The last five columns (SDP+CP) present the performance of our method, where also a column has been added for the discrepancy of the best found value (best discr), and a column for the time needed by the semidefinite programming solver (sdp time).

Table 3.1 shows that our approach always finds a better (or equally good) estimate for α than the standard constraint programming approach. This becomes more obvious for larger n. However, there are two (out of 30) instances in which our method needs substantially more time to achieve this result (g75d015 and wg75d010). A final observation concerns the discrepancy of the best found solutions. Our method appears to find those (often optimal) solutions at rather low discrepancies.

7.5.4 Graphs Arising from Coding Theory

The next experiments are performed on structured (unweighted) graphs arising from coding theory, obtained from the website of N.J.A. Sloane[1]. We have used those instances that were solvable in reasonable time by the semidefinite programming solver (here reasonable means within 1000 seconds). For these instances, the value of α happened to be known already.

The results are reported in Table 3.2, which follows the same format as Table 3.1. It shows the same behaviour as the results on random graphs. Namely,

[1] See `http://www.research.att.com/~njas/doc/graphs.html`.

instance				CP			SDP + CP				
			edge		total	back-		best	sdp	total	back-
name	n	m	density	α	time	tracks	α	discr	time	time	tracks
g50d005	50	70	0.06	27	5.51	50567	27	0	0.26	0.27	0
g50d010	50	114	0.09	22	28.54	256932	22	0	0.35	0.36	0
g50d015	50	190	0.16	17	5.83	48969	17	0	0.49	0.49	0
g75d005	75	138	0.05	$\geqslant$ 35	N.A.		36	0	0.72	0.73	0
g75d010	75	282	0.10	$\geqslant$ 25	N.A.		$\geqslant$ 25	5	1.4	N.A.	
g75d015	75	426	0.15	21	170.56	1209019	21	0	2.81	664.92	1641692
g100d005	100	254	0.05	43	N.A.		43	0	2.07	2.1	0
g100d010	100	508	0.10	$\geqslant$ 31	N.A.		$\geqslant$ 31	0	4.94	N.A.	
g100d015	100	736	0.15	$\geqslant$ 24	N.A.		$\geqslant$ 24	4	9.81	N.A.	
g125d005	125	393	0.05	$\geqslant$ 44	N.A.		$\geqslant$ 49	1	4.92	N.A.	
g125d010	125	791	0.10	$\geqslant$ 30	N.A.		$\geqslant$ 33	6	12.58	N.A.	
g125d015	125	1160	0.15	$\geqslant$ 24	N.A.		$\geqslant$ 27	1	29.29	N.A.	
g150d005	150	545	0.05	$\geqslant$ 44	N.A.		$\geqslant$ 52	3	10.09	N.A.	
g150d010	150	1111	0.10	$\geqslant$ 32	N.A.		$\geqslant$ 38	4	27.48	N.A.	
g150d015	150	1566	0.14	$\geqslant$ 26	N.A.		$\geqslant$ 29	8	57.86	N.A.	
wg50d005	50	70	0.06	740	4.41	30528	740	0	0.29	0.3	0
wg50d010	50	126	0.10	636	3.12	19608	636	0	0.41	0.41	0
wg50d015	50	171	0.14	568	4.09	25533	568	0	0.59	4.93	13042
wg75d005	75	128	0.05	1761	744.29	4036453	1761	0	1.05	1.07	0
wg75d010	75	284	0.10	1198	325.92	1764478	1198	13	1.9	924.2	1974913
wg75d015	75	409	0.15	972	40.31	208146	972	0	3.62	51.08	87490
wg100d005	100	233	0.05	2302	N.A.		2302	0	2.59	2.62	0
wg100d010	100	488	0.10	$\geqslant$ 1778	N.A.		$\geqslant$ 1778	2	6.4	N.A.	
wg100d015	100	750	0.15	$\geqslant$ 1412	N.A.		$\geqslant$ 1412	2	15.21	N.A.	
wg125d005	125	372	0.05	$\geqslant$ 3390	N.A.		$\geqslant$ 3779	3	5.39	N.A.	
wg125d010	125	767	0.10	$\geqslant$ 2175	N.A.		$\geqslant$ 2796	0	18.5	N.A.	
wg125d015	125	1144	0.15	$\geqslant$ 1899	N.A.		$\geqslant$ 1991	4	38.24	N.A.	
wg150d005	150	588	0.05	$\geqslant$ 3759	N.A.		$\geqslant$ 4381	3	13.57	N.A.	
wg150d010	150	1167	0.10	$\geqslant$ 2533	N.A.		$\geqslant$ 3265	9	40.68	N.A.	
wg150d015	150	1630	0.15	$\geqslant$ 2518	N.A.		$\geqslant$ 2828	11	82.34	N.A.	

Table 3.1. Computational results on random graphs, with n vertices and m edges. All times are in seconds. N.A. means "not applicable" due to time limit (1000 s).

our method always finds better solutions than the standard constraint programming solver, in less time or within the time limit. This is not surprising, because the edge density of these instances are exactly in the region in which our method is supposed to behave best (with the exception of 1dc.64 and 1zc.128), as analyzed in Section 7.5.2. Again, our method finds the best solutions at a low discrepancy. Note that the instance 1et.64 shows the strength of the semidefinite relaxation with respect to standard constraint programming. The difference in computation time to prove optimality is huge.

instance					CP				SDP + CP					
			edge			total	back-		best	sdp	total	back-
name	n	m	density	α	α	time	tracks	α	discr	time	time	tracks
1dc.64	64	543	0.27	10	10	11.44	79519	10	0	5.08	5.09	0
1dc.128	128	1471	0.18	16	$\geq$ 16	N.A.		16	0	49.95	49.98	0
1dc.256	256	3839	0.12	30	$\geq$ 26	N.A.		30	0	882.21	882.33	0
1et.64	64	264	0.13	18	18	273.06	2312832	18	0	1.07	1.08	0
1et.128	128	672	0.08	28	$\geq$ 28	N.A.		$\geq$ 28	0	11.22	N.A.	
1et.256	256	1664	0.05	50	$\geq$ 46	N.A.		$\geq$ 50	0	107.58	N.A.	
1tc.64	64	192	0.10	20	$\geq$ 20	N.A.		20	0	0.78	0.79	0
1tc.128	128	512	0.06	38	$\geq$ 37	N.A.		38	0	8.14	8.18	0
1tc.256	256	1312	0.04	63	$\geq$ 58	N.A.		$\geq$ 63	4	72.75	N.A.	
1tc.512	512	3264	0.02	110	$\geq$ 100	N.A.		$\geq$ 110	2	719.56	N.A.	
lzc.128	128	2240	0.28	18	$\geq$ 18	N.A.		$\geq$ 18	4	129.86	N.A.	

Table 3.2. Computational results on graphs arising from coding theory, with n vertices and m edges. All times are in seconds. N.A. means "not applicable" due to time limit (1000 s).

7.5.5 Graphs from the DIMACS Benchmarks Set

Our final experiments are performed on a subset of the DIMACS benchmark set for the maximum clique problem[2]. Although our method is not intended to be competitive with the best heuristics and exact methods for maximum clique problems, it is still interesting to see its performance on this standard benchmark set. As pointed out in Section 7.4, we have transformed these maximum clique problems to stable set problems on the complement graph.

The results are reported in Table 3.3, which again follows the same format as Table 3.1. The choice for this particular subset of instances is made by the solvability of an instance by a semidefinite programming solver in reasonable time (again, reasonable means 1000 seconds). For all instances with edge density smaller than 0.24, our method outperforms the standard constraint

[2] See `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique`.

instance					CP			SDP + CP			
			edge			total	back-	best		sdp	total	back-
name	n	m	density	α	α	time	tracks	α	discr	time	time	tracks
hamming6-2	64	192	0.095	32	32	20.22	140172	32	0	0.68	0.69	0
hamming6-4	64	1312	0.651	4	4	0.28	804	4	0	27.29	28.10	706
hamming8-2	256	1024	0.031	128	≥ 128	N.A.		128	0	45.16	45.55	0
johnson8-2-4	28	168	0.444	4	4	0.05	255	4	0	0.35	0.35	0
johnson8-4-4	70	560	0.232	14	14	15.05	100156	14	0	4.82	4.83	0
johnson16-2-4	120	1680	0.235	8	≥ 8	N.A.		8	0	43.29	43.32	0
MANN_a9	45	72	0.072	16	16	162.81	1738506	16	1	0.17	82.46	411104
MANN_a27	378	702	0.010	126	≥ 103	N.A.		≥ 125	3	70.29	N.A.	
MANN_a45	1035	1980	0.004	345	≥ 156	N.A.		≥ 338	1	1047.06	N.A.	
san200_0.9_1	200	1990	0.100	70	≥ 45	N.A.		70	0	170.01	170.19	0
san200_0.9_2	200	1990	0.100	60	≥ 36	N.A.		60	0	169.35	169.51	0
san200_0.9_3	200	1990	0.100	44	≥ 26	N.A.		44	0	157.90	157.99	0
sanr200_0.9	200	2037	0.102	42	≥ 34	N.A.		≥ 41	4	131.57	N.A.	

Table 3.3. Computational results on graphs from the DIMACS benchmark set for maximum clique problems, with n vertices and m edges. All times are in seconds. N.A. means "not applicable" due to time limit (1000 s).

programming approach. For higher densities however, the opposite holds. This is exactly what could be expected from the analysis of Section 7.5.2. A special treatment has been given to instance MANN_a45. We stopped the semidefinite programming solver at the time limit of 1000 seconds, and used its intermediate feasible solution as if it were the optimal fractional solution. We then proceeded our algorithm for a couple of seconds more, to search for a solution up to discrepancy 1.

In Table 3.4 we compare our method with two methods that are specialized for maximum clique problems. The first method was presented by Östergård [2002], and follows a branch-and-bound approach. The second method is a constraint programming approach, using a special constraint for the maximum

instance		Östergård		Régin			CP + SDP		
	edge		total		total	back-		total	back-
name	density	α	time	α	time	tracks	α	time	tracks
hamming6-2	0.095	32	0.01	32	0.00	17	32	0.69	0
hamming6-4	0.651	4	0.01	4	0.00	42	4	28.10	706
hamming8-2	0.031	128	0.04	128	0.00	65	128	45.55	0
johnson8-2-4	0.444	4	0.01	4	0.00	14	4	0.35	0
johnson8-4-4	0.232	14	0.01	14	0.00	140	14	4.83	0
johnson16-2-4	0.235	8	0.27	8	11.40	250505	8	43.32	0
MANN_a9	0.073	16	0.01	16	0.00	50	16	82.46	411104
MANN_a27	0.010	126	> 10000	126	55.44	1258768	≥ 125	> 1000	
MANN_a45	0.004	345	> 10000	≥ 345	> 43200		≥ 338	> 1000	
san200_0.9_1	0.100	70	0.27	70	3.12	1040	70	170.19	0
san200_0.9_2	0.100	60	4.28	60	7.86	6638	60	169.51	0
san200_0.9_3	0.100	44	> 10000	44	548.10	758545	44	157.99	0
san200_0.9	0.102	42	> 10000	42	450.24	541496	≥ 41	> 1000	

Table 3.4. A comparison of different methods on graphs from the DIMACS benchmark set for maximum clique problems, with n vertices and m edges. All times are in seconds.

clique problem, with a corresponding propagation algorithm. This idea was introduced by Fahle [2002] and extended and improved by Régin [2003]. Since all methods are performed on different machines, we identify a time ratio between them. A machine comparison from SPEC[3] shows that our times are comparable with the times of Östergård. We have multiplied the times of Régin with 3, following the time comparison made by Régin [2003]. In general, our method is outperformed by the other two methods, although there is one instance on which our method performs best (san200_0.9_3).

[3] `http://www.spec.org/`

7.6 Discussion and Conclusion

We have presented a method to use semidefinite relaxations within constraint programming. The fractional solution values of the relaxation have been used to define variable and value ordering heuristics. Further, the solution value of the relaxation has been used as a bound for the corresponding constraint programming objective function.

Computational results on stable set and maximum clique problems have indicated that a semidefinite relaxation can be of great value in constraint programming. Compared to a standard constraint programming approach, our method obtained far better results. Specialized algorithms for the maximum clique problem however, generally outperformed our method.

The results in this chapter have shown that semidefinite relaxations are indeed applicable in constraint programming. The obtained results give rise to various extensions. First of all, it is interesting to apply our method to problems that also contain side constraints, apart from the structure for which a semidefinite relaxation is known. Secondly, the efficiency of our method may be increased by strengthening the relaxation. For example, for the stable set problem so-called clique-constraints (among others) can be added to the semidefinite relaxation. Thirdly, the proof of optimality may be accelerated similar to the method presented in Chapter 6, by adding discrepancy constraints to the semidefinite relaxation.

Although the solution to a semidefinite relaxation often provides a promising search direction, in some cases the relaxation is less informative. It is interesting to analyze this behaviour. For example, Leahu and Gomes [2004] have analyzed the quality of linear relaxations for the Partial Latin Square Completion Problem. They found that the solution of a linear relaxation is less informative exactly in a certain phase transition of the problem instances. A similar analysis could be made for semidefinite relaxations.

Finally, it may be possible to exploit the semidefinite relaxation as a propagation mechanism, based on sub-optimality of certain variable assignments. Such propagation, also called "variable fixing", in case of linear relaxations has been studied by Focacci, Lodi, and Milano [1999a]. For semidefinite relaxations Helmberg [2000] introduced a variable fixing procedure, but it is yet unclear how to exploit this in constraint programming.

Perspectives

In this thesis we have investigated the application of operations research techniques in constraint programming. The main conclusion of our work is that a wide range of techniques from operations research is effectively applicable in different aspects of constraint programming. We have already indicated detailed possible future investigations at the end of each chapter, where appropriate. Next we present a more general perspective of the topic, reflecting the author's opinion.

In Chapter 3 and Chapter 4 we have seen that the efficiency of many propagation algorithms relies on results from graph theory. The essence of such combinations is the embedding of a graph-theoretical structure inside a global constraint. In order to be applicable, the graph-theoretical structure should allow an efficient solution algorithm. For example, the `alldifferent` constraint embeds the matching structure, while the soft global constraints considered in Chapter 4 embed the structure of a flow. We could similarly embed other efficient graph algorithms in new global constraints. In fact, this line of research already attracts increasing attention. Every year new global constraints appear that embed graph-theoretical structures.

The above embedding of efficient algorithms inside global constraints can be extended to other techniques of operations research. For instance, a system of linear constraints can be embedded inside a global constraint, which applies a linear programming solver; see for example Rodosek, Wallace, and Hajian [1999]. Hence, virtually every efficient technique from operations research could be embedded inside a global constraint. Doing so, we increase the modelling power and hopefully the solving power of constraint programming.

A precaution should be made, however. Many authors have argued that more constraint propagation is not always better. In many cases it is more efficient to spent less time on propagation while traversing more nodes in the search tree. To give an intuition, in some cases obtaining hyper-arc consistency for the `alldifferent` constraint does not pay off, even though the particular propagation algorithm is very efficient. One way to overcome this

problem is to develop propagation algorithms that achieve weaker local consistency notions, but that are much faster. Another direction is to investigate and improve the actual application and possible incrementality of relatively expensive propagation algorithms within the search process. For example, it is not uncommon to apply certain propagation algorithms only below a certain depth in the search tree.

In Chapter 6 and Chapter 7 we have proposed to use linear and semidefinite relaxations from operations research to guide the search process of constraint programming. A successful and promising direction is the use of constraint programming constructs to improve such relaxations. For example, we have seen that a linear relaxation based on decision postponement in combination with a discrepancy constraint can greatly improve the proof of optimality. More research in this direction could make constraint programming more suitable for optimization problems.

A drawback of the tight integration of different solution methods is the induced overhead, which may not always pay off. For example, sometimes a problem can be solved by integer programming, constraint programming, or a satisfiability solver. If we build and apply an all-purpose combination of such solvers, the result may be much slower than if we apply the fastest solver among them to each particular instance. This leads to the idea of applying a "portfolio" of algorithms. There exist many possible strategies of combining algorithms in a portfolio during search; see Gomes [2003, Section 7] for an overview. Another promising research direction along this line is to identify problem characteristics to automatically decide the most suitable algorithm, as for example done by Guerri and Milano [2004].

In conclusion, there are many possibilities to apply operations research techniques in constraint programming. However, one should keep in mind the practical consequences, to make such applications actual improvements. Then this exciting research area may become even more successful.

References

R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993. Cited on page 9.

F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5 (1):13–51, 1995. Cited on page 14.

E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, third edition, 1999. `http://www.netlib.org/lapack/`. Cited on page 126.

G. Appa, D. Magos, and I. Mourtos. LP Relaxations of Multiple all_different Predicates. In J.-C. Régin and M. Rueher, editors, *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, volume 3011 of *LNCS*, pages 364–369. Springer, 2004. Cited on page 51.

K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999. Cited on page 16.

K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. Cited on page 15, 16.

N. Ascheuer. *Hamiltonian Path Problems in the On-line Optimization of Flexible Manufacturing Problems*. PhD thesis, Technische Universität Berlin, 1995. Also available as technical report TR-96-3, Zuse Institute Berlin. Cited on page 104, 111, 113.

N. Ascheuer, M. Fischetti, and M. Grötschel. Solving the Asymmetric Travelling Salesman Problem with time windows by branch-and-cut. *Mathematical Programming, Series A*, 90(3):475–506, 2001. Cited on page 116.

E. Balas and N. Simonetti. Linear Time Dynamic-Programming Algorithms for New Classes of Restricted TSPs: A Computational Study. *INFORMS Journal on Computing*, 13(1):56–75, 2000. Cited on page 116.

N. Barnier and P. Brisset. Graph Coloring for Air Traffic Flow Management. In *Proceedings of the Fourth International Workshop on Integration of AI*

and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2002), pages 133–147, 2002. Cited on page 28.

R. Barták. Dynamic global constraints: A first view. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.
`http://www.arxiv.org/html/cs/0110012`. Cited on page 42.

J.C. Beck and L. Perron. Discrepancy-Bounded Depth First Search. In *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2000)*, 2000. Cited on page 21, 22.

N. Beldiceanu. Global Constraints as Graph Properties on Structured Network of Elementary Constraints of the Same Type. Technical Report T2000/01, SICS, 2000. Cited on page 44, 54, 78.

N. Beldiceanu, M. Carlsson, and T. Petit. Deriving Filtering Algorithms from Constraint Checkers. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 107–122. Springer, 2004a. Cited on page 76.

N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. Cited on page 28.

N. Beldiceanu, I. Katriel, and S. Thiel. Filtering Algorithms for the Same Constraint. In J.-C. Régin and M. Rueher, editors, *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, volume 3011 of *LNCS*, pages 65–79. Springer, 2004b. Cited on page 78, 79.

N. Beldiceanu and T. Petit. Cost Evaluation of Soft Global Constraints. In J.-C. Régin and M. Rueher, editors, *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, volume 3011 of *LNCS*, pages 80–95. Springer, 2004. Cited on page 55, 70.

S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Satisfaction and Optimization. *Journal of the ACM*, 44(2):201–236, 1997. Cited on page 55.

B. Borchers. A C Library for Semidefinite Programming. *Optimization Methods and Software*, 11(1):613–623, 1999.
`http://www.nmt.edu/~borchers/csdp.html`. Cited on page 126.

G. Carpaneto, S. Martello, and P. Toth. Algorithms and codes for the Assignment Problem. In B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino, editors, *Fortran Codes for Network Optimization*, volume 13 of *Annals of Operations Research*, pages 193–223. J.C. Baltzer AG, 1988. Cited on page 110.

Y. Caseau and F. Laburthe. Solving Small TSPs with Constraints. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 316–330. The MIT Press, 1997a. Cited on page 107, 108.

Y. Caseau and F. Laburthe. Solving Various Weighted Matching Problems with Constraints. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330 of *LNCS*, pages 17–31. Springer, 1997b. Cited on page 47.

V. Chandru and J. Hooker. *Optimization Methods for Logical Inference*. Wiley, 1999. Cited on page 4.

V. Chvátal. *Linear programming*. Freeman, 1983. Cited on page 12.

G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation – Proceedings of a conference*. Wiley, 1951. Cited on page 13.

R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173:283–308, 1997. Cited on page 35.

M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702. Springer, 1988. Cited on page 27.

D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the Second IEEE International Conference on Fuzzy Systems*, volume 2, pages 1131–1136, 1993. Cited on page 55.

T.E. Easterfield. A combinatorial algorithm. *Journal of the London Mathematical Society*, 21:219–226, 1946. Cited on page 31.

J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17: 449–467, 1965. Cited on page 9, 46.

T. Fahle. Simple and Fast: Improving a Branch-And-Bound Algorithm for Maximum Clique. In *10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *LNCS*, pages 485–498. Springer Verlag, 2002. Cited on page 132.

H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. In *Proceedings of the first European Congress on Fuzzy and Intelligent Technologies*, 1993. Cited on page 55.

M. Fischetti and P. Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37:319–328, 1989. Cited on page 49, 105, 106.

M. Fischetti and P. Toth. An additive bounding procedure for the asymmetric travelling salesman problem. *Mathematical Programming*, 53:173–197, 1992. Cited on page 105.

F. Focacci. *Solving Combinatorial Optimization Problems in Constraint Programming*. PhD thesis, University of Ferrara, 2001. Cited on page 104, 106.

F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, pages 189–203. Springer, 1999a. Cited on page 133.

F. Focacci, A. Lodi, and M. Milano. Integration of CP and OR methods for matching problems. In *Proceedings of the First International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'99)*, 1999b. Cited on page 47.

F. Focacci, A. Lodi, and M. Milano. A Hybrid Exact Algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002. Cited on page 100, 108, 111, 115, 116.

F. Focacci, A. Lodi, and M. Milano. Exploiting relaxations in CP. In M. Milano, editor, *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces*, chapter 5. Kluwer Academic Publishers, 2003. Cited on page 120.

E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992. Cited on page 55.

M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, 1979. Cited on page 11, 12.

I. Gent, K. Stergiou, and T. Walsh. Decomposable Constraints. *Artificial Intelligence*, 123(1–2):133–156, 2000. Cited on page 36.

A.M.H. Gerards. Matching. In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors, *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science*, pages 135–224. Elsevier Science, 1995. Cited on page 8.

M.L. Ginsberg and W.D. Harvey. Iterative broadening. *Artificial Intelligence*, 55(2):367–383, 1992. Cited on page 98.

M. Goemans and F. Rendl. Combinatorial Optimization. In H. Wolkowicz, R. Saigal, and L. Vandenberghe, editors, *Handbook of Semidefinite Programming*, pages 343–360. Kluwer, 2000. Cited on page 14.

M.X. Goemans and D.P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *Journal of the ACM*, 42(6):1115–1145, 1995. Cited on page 120.

C.P. Gomes. Randomized Backtrack Search. In M. Milano, editor, *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces*, chapter 8. Kluwer Academic Publishers, 2003. Cited on page 88, 136.

C.P. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Ar-*

tificial Intelligence Conference (AAAI / IAAI), pages 431–437. AAAI Press / The MIT Press, 1998. Cited on page 87, 88.

C.P. Gomes and D. Shmoys. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, 2002. Cited on page 28, 97.

M. Grönkvist. A Constraint Programming Model for Tail Assignment. In J.-C. Régin and M. Rueher, editors, *Proceedings of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, volume 3011 of *LNCS*, pages 142–156. Springer, 2004. Cited on page 28.

M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Wiley, 1988. Cited on page 14, 125.

G. Gruber and F. Rendl. Computational experience with stable set relaxations. *SIAM Journal on Optimization*, 13(4):1014–1028, 2003. Cited on page 125.

A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 475–479. IOS Press, 2004. Cited on page 136.

P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935. Cited on page 30.

W.D. Harvey and M.L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 607–615, 1995. Cited on page 20, 90, 91.

C. Helmberg. Fixing variables in semidefinite relaxations. *SIAM J. Matrix Anal. Appl.*, 21(3):952–969, 2000. Cited on page 133.

M. Henz, T. Müller, and S. Thiel. Global constraints for round robin tournament scheduling. *European Journal of Operational Research*, 153(1):92–101, 2004. Cited on page 43.

J. Hooker. *Logic-Based Methods for Optimization - Combining Optimization and Constraint Satisfaction*. Wiley, 2000. Cited on page 4, 50.

J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. Cited on page 9, 41, 63.

J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979. Cited on page 72.

S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *INFORMS Journal on Computing*, 12(3): 177–191, 2000. Cited on page 120.

N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 302–311. ACM, 1984a. Cited on page 14.

N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984b. Cited on page 14.

L.G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979. Cited on page 14.

R.E. Korf. Improved Limited Discrepancy Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, volume 1, pages 286–291. AAAI Press / The MIT Press, 1996. Cited on page 20.

J. Larrosa. Node and Arc Consistency in Weighted CSP. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI / IAAI)*, pages 48–53. AAAI Press / The MIT Press, 2002. Cited on page 55.

J. Larrosa and T. Schiex. In the quest of the best form of local consistency for Weighted CSP. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 239–244. Morgan Kaufmann, 2003. Cited on page 55.

M. Laurent, S. Poljak, and F. Rendl. Connections between semidefinite relaxations of the max-cut and stable set problems. *Mathematical Programming*, 77:225–246, 1997. Cited on page 121.

M. Laurent and F. Rendl. Semidefinite Programming and Integer Programming. In K. Aardal, G. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, Handbooks in Operations Research and Management Science. Elsevier, 2004. Also available as Technical Report PNA-R0210, CWI, Amsterdam. Cited on page 14, 121, 122.

J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence. An International Journal*, 10(1): 29–127, 1978. Cited on page 27.

L. Leahu and C.P. Gomes. Quality of LP-based Approximations for Highly Combinatorial Problems. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 377–392. Springer, 2004. Cited on page 133.

M. Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Second International Workshop on Constraint-based Reasoning (Constraint-96), Key West, Florida*, 1996. Cited on page 36, 38, 39, 42.

J. Lee. All-Different Polytopes. *Journal of Combinatorial Optimization*, 6(3): 335–352, 2002. Cited on page 51.

A. Lodi. Personal communication, 2002. Cited on page 106.

A. Lodi and M. Milano. Discrepancy-based additive bounding. In *Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'03)*, pages 17–23, 2003. Cited on page 105, 106, 107.

A. Lodi, M. Milano, and L.-M. Rousseau. Discrepancy-Based Additive Bounding for the Alldifferent Constraint. In F. Rossi, editor, *Proceedings of the Ninth International Conference on Principles and Practice of Constraint*

Programming (CP 2003), volume 2833 of *LNCS*, pages 510–524. Springer, 2003. Cited on page 49, 107.

A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 245–250. Morgan Kaufmann, 2003. Cited on page 37, 38, 42, 43.

L. Lovász. On the Shannon capacity of a graph. *IEEE Transactions on Information Theory*, 25:1–7, 1979. Cited on page 124.

L. Lovász and M. D. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986. Cited on page 8.

K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 306–319. Springer, 2000. Cited on page 37, 38, 42.

S. Micali and V.V. Vazirani. An $O(\sqrt{|v|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 17–27. IEEE, 1980. Cited on page 9, 46.

M. Milano, editor. *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces*. Kluwer Academic Publishers, 2003. Cited on page 4.

M. Milano and W.J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In *Proceedings of the Joint ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*, pages 7–22, 2002a. Cited on page 5.

M. Milano and W.J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In P. Van Hentenryck, editor, *Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *LNCS*, pages 1–16. Springer Verlag, 2002b. Cited on page 5.

R. Mohr and G. Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI)*, pages 651–656, 1988. Cited on page 42.

G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988. Cited on page 12, 13.

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 Third International Conference on the Practical Applications of Prolog (PAP'95)*. Alinmead Software Ltd, 1995. Cited on page 28.

P.R.J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002. Cited on page 131.

G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In M. Wallace, editor, *Proceedings of the Tenth International*

Conference on Principles and Practice of Constraint Programming (CP 2004), volume 3258 of *LNCS*, pages 482–495. Springer, 2004. Cited on page 72, 74, 76.

G. Pesant, M. Gendreau, J.Y. Potvin, and J.M. Rousseau. An exact constraint logic programming algorithm for the travelling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998. Cited on page 108.

J. Petersen. Die Theorie der regulären graphs. *Acta Mathematica*, 15:193–220, 1891. Cited on page 8, 40.

T. Petit, J.-C. Régin, and C. Bessière. Meta constraints on violations for over constrained problems. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 358–365, 2000. Cited on page 55, 81, 82.

T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. In T. Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *LNCS*, pages 451–463. Springer, 2001. Cited on page 55, 62.

J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, pages 359–366. AAAI Press / The MIT Press, 1998. Cited on page 36, 37, 42.

J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 362–367. AAAI Press, 1994. Cited on page 27, 39, 40, 42, 61.

J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI / IAAI)*, volume 1, pages 209–215. AAAI Press / The MIT Press, 1996. Cited on page 28, 65, 68.

J.-C. Régin. Arc Consistency for Global Cardinality Constraints with Costs. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, pages 390–404. Springer, 1999a. Cited on page 49, 60, 65.

J.-C. Régin. The symmetric alldiff constraint. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 420–425, 1999b. Cited on page 43, 44, 46.

J.-C. Régin. Cost-Based Arc Consistency for Global Cardinality Constraints. *Constraints*, 7:387–405, 2002. Cited on page 49, 60, 65.

J.-C. Régin. Using Constraint Programming to Solve the Maximum Clique Problem. In F. Rossi, editor, *Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 634–648. Springer Verlag, 2003. Cited on page 132.

J.-C. Régin, T. Petit, C. Bessière, and J.-F. Puget. An Original Constraint Based Approach for Solving over Constrained Problems. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 543–548. Springer, 2000. Cited on page 55, 56.

G. Reinelt. TSPLIB - a Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3:376–384, 1991. Cited on page 95, 111.

R. Rodosek, M.G. Wallace, and M.T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operations Research*, 86:63–87, 1999. Cited on page 135.

F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI)*, pages 550–556, 1990. Cited on page 34.

Z. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of the First IEEE Conference on Evolutionary Computing*, pages 542–547, 1994. Cited on page 55.

T. Schiex. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints ?". In *Proceedings of the 8th Annual Conference on Uncertainty in Artificial Intelligence*, pages 268–275. Morgan Kaufmann, 1992. Cited on page 55.

T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 631–639. Morgan Kaufmann, 1995. Cited on page 55.

A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986. Cited on page 12, 13.

A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003. Cited on page 7, 8, 9, 10, 30, 31, 40, 110.

C. Schulte and P.J. Stuckey. When Do Bounds and Domain Propagation Lead to the Same Search Space. In H. Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pages 115–126. ACM Press, 2001. Cited on page 43.

M. Sellmann. An Arc-Consistency Algorithm for the Minimum Weight All Different Constraint. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *LNCS*, pages 744–749. Springer, 2002. Cited on page 49.

K. Stergiou and T. Walsh. The difference all-difference makes. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 414–419, 1999. Cited on page 34, 36.

R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972. Cited on page 41, 46, 65.

E. Tsang, J. Ford, P. Mills, R. Bradwell, R. Williams, and P. Scott. ZDC-Rostering: A Personnel Scheduling System Based On Constraint Program-

ming. Technical Report 406, University of Essex, Colchester, UK, 2004. Cited on page 28.

P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. The MIT Press, Cambridge, MA, 1989. Cited on page 27, 42.

W.J. van Hoeve. The Alldifferent Constraint: A Survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.
`http://www.arxiv.org/html/cs/0110012`. Cited on page 4.

W.J. van Hoeve. A Hybrid Constraint Programming and Semidefinite Programming Approach for the Stable Set Problem. In F. Rossi, editor, *Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 407–421. Springer Verlag, 2003a. Cited on page 5.

W.J. van Hoeve. A hybrid constraint programming and semidefinite programming approach for the stable set problem. In *Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'03)*, pages 3–16, 2003b. Cited on page 5.

W.J. van Hoeve. A Hyper-Arc Consistency Algorithm for the Soft Alldifferent Constraint. In M. Wallace, editor, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 679–689. Springer, 2004. Cited on page 5, 55.

W.J. van Hoeve. Exploiting Semidefinite Relaxations in Constraint Programming. *Computers and Operations Research*, 2005. To appear. Cited on page 5.

W.J. van Hoeve and M. Milano. Decomposition Based Search - A theoretical and experimental evaluation. Technical Report LIA00203, University of Bologna, 2003. Cited on page 5.

W.J. van Hoeve and M. Milano. Postponing Branching Decisions. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 1105–1106. IOS Press, 2004. Cited on page 5.

W.J. van Hoeve, G. Pesant, and L.-M. Rousseau. On Global Warming (Softening Global Constraints). In *Proceedings of the 6th International Workshop on Preferences and Soft Constraints (held in conjunction with CP 2004)*, 2004. Cited on page 5.

M. Wallace, Y. Caseau, and J.-F. Puget. Open Perspectives. In M. Milano, editor, *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces*, chapter 10. Kluwer Academic Publishers, 2003. Cited on page 53.

M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, 1997. Cited on page 27.

T. Walsh. Depth-Bounded Discrepancy Search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1388–1393, 1997. Cited on page 21, 91.

R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2): 3–35, 2001.
`http://math-atlas.sourceforge.net/`. Cited on page 126.

H.P. Williams and Hong Yan. Representations of the all_different Predicate of Constraint Satisfaction in Integer Programming. *INFORMS Journal on Computing*, 13(2):96–103, 2001. Cited on page 50.

H. Wolkowicz, R. Saigal, and L. Vandenberghe, editors. *Handbook of Semidefinite Programming*, volume 27 of *International series in operations research and management science*. Kluwer, 2000. Cited on page 14.

J. Zhou. A permutation-based approach for solving the job-shop problem. *Constraints*, 2(2):185–213, 1997. Cited on page 28.

Index

Samenvatting

In dit proefschrift onderzoeken we de toepassing van efficiënte technieken uit de operationele research in constraint programmering voor het oplossen van NP-moeilijke combinatorische problemen. Onder dergelijke technieken uit de operationele research verstaan we bijvoorbeeld technieken uit de lineaire en semidefinite programmering en uit de grafentheorie. Deze technieken worden gekenmerkt door hun toepassing op specifieke beperkte problemen, en hun geschiktheid voor optimaliseringsproblemen.

Een combinatorisch probleem wordt gemodelleerd door middel van variabelen waarover eisen gesteld worden. Deze eisen worden "constraints" genoemd. In constraint programmering worden dergelijke problemen opgelost door systematisch *zoeken* en *propagatie*. Het zoeken gebeurt door alle mogelijke combinaties van waarden voor de variabelen te genereren. Dit proces vormt een zogenaamde zoekboom. Wanneer een combinatie aan alle constraints voldoet, is deze een oplossing van het probleem. Helaas is de zoekboom in het algemeen exponentieel groot. Om het aantal combinaties, en dus de zoekboom, te beperken, wordt er tijdens het zoekproces propagatie toegepast. Propagatie houdt in dat voor elke constraint in het model apart wordt nagegaan of deze nog vervuld kan worden. Bovendien worden bepaalde waarden geïdentificeerd die nooit tot een oplossing leiden. Door het verwijderen van deze waarden wordt het aantal combinaties beperkt.

Het eerste deel van het proefschrift behandelt propagatie-algoritmen. We beginnen met propagatie-algoritmen voor een van de bekendste constraints: de *alldifferent* constraint. In de loop der tijd zijn er verschillende propagatie-algoritmen voor deze constraint ontworpen. We presenteren deze algoritmen op een systematische wijze, veelal gebaseerd op technieken uit de operationele research, in het bijzonder uit de grafentheorie. Deze technieken zorgen ervoor dat de propagatie-algoritmen efficiënt, en dus praktisch toepasbaar zijn. In het nu volgende geval passen we deze technieken toe op vergelijkbare constraints.

Voor veel praktische problemen bestaat er geen oplossing, omdat er te veel eisen zijn gesteld. Om toch een enigszins bevredigende "oplossing" te

verkrijgen, worden deze problemen gemodelleerd met "zachte" constraints. Deze zachte constraints mogen worden overschreden. Echter, de totale mate van overschrijding van de zachte constraints moet hierbij worden geminimaliseerd. We presenteren een nieuwe, generieke, methode om dergelijke zachte constraints efficiënt te propageren. Hierbij maken we gebruik van grafentheoretische technieken voor stromen in netwerken. Onze methode is toegepast op een aantal bekende constraints. Naast het verkrijgen van efficiënte propagatie-algoritmen voor bekende "maten van overschrijding" hebben we tevens een aantal nieuwe maten van overschrijding gedefinieerd, met bijbehorende propagatie-algoritmen.

Het tweede deel van het proefschrift behandelt zoekmethoden. Hierbij richten we ons met name op strategieën om een zoekboom te doorlopen. Hiervoor worden heuristieken gebruikt. Op elk knooppunt in de boom wordt op grond van een heuristiek bepaald welke tak, corresponderend met een mogelijke waarde voor een variabele, als eerstvolgende wordt bezocht. Dergelijke heuristieken hebben grote invloed op de efficiëntie van constraint programmering.

In sommige gevallen kan een heuristiek tussen twee of meer takken geen keuze maken. We introduceren een nieuwe zoekmethode voor dergelijke gevallen: stel de keuze tussen deze takken uit. Middels een theoretische en experimentele analyse laten we de voordelen van deze methode zien. In specifieke gevallen biedt deze methode bovendien de mogelijkheid om op effectieve wijze technieken uit de lineaire programmering toe te passen op optimaliseringsproblemen, zoals uit het onderstaande blijkt.

Om constraint programmering beter te laten presteren in het geval van optimalisatie problemen, introduceren we de volgende methode. Voordat we de zoekboom doorlopen, lossen we een lineaire relaxatie van het probleem op. Hieruit verkrijgen we onder anderen zogenaamde "gereduceerde kosten", die elk corresponderen met een keuze in de zoekboom. Daarop passen we deze gereduceerde kosten toe als heuristiek. We laten experimenteel zien dat een op gereduceerde kosten gebaseerde heuristiek zeer effectief is. In het geval twee of meer takken corresponderen met dezelfde gereduceerde kostwaarde, passen we bovenstaande methode van keuze-uitstelling toe. Dit stelt ons in staat, in combinatie met "limited discrepancy search", eerder optimaliteit van een oplossing te bewijzen, door middel van technieken uit de lineaire programmering. Experimentele resultaten laten de toepasbaarheid en effectiviteit van onze methode zien.

Tot slot behandelen we de mogelijkheid om een semidefinite relaxatie toe te passen in constraint programmering. Vergelijkbaar met het voorgaande lossen we een dergelijke relaxatie op voordat we de zoekboom doorlopen. Vanwege de aard van de semidefinite relaxatie (er zijn geen gereduceerde kosten) gebruiken we in dit geval echter de oplossing van de relaxatie als heuristiek. We tonen experimenteel aan dat dergelijke relaxaties inderdaad effectief toegepast kunnen worden in constraint programmering.

Titles in the ILLC Dissertation Series:

ILLC DS-2001-01: **Maria Aloni**
Quantification under Conceptual Covers
ILLC DS-2001-02: **Alexander van den Bosch**
Rationality in Discovery - a study of Logic, Cognition, Computation and Neuropharmacology
ILLC DS-2001-03: **Erik de Haas**
Logics For OO Information Systems: a Semantic Study of Object Orientation from a Categorial Substructural Perspective
ILLC DS-2001-04: **Rosalie Iemhoff**
Provability Logic and Admissible Rules
ILLC DS-2001-05: **Eva Hoogland**
Definability and Interpolation: Model-theoretic investigations
ILLC DS-2001-06: **Ronald de Wolf**
Quantum Computing and Communication Complexity
ILLC DS-2001-07: **Katsumi Sasaki**
Logics and Provability
ILLC DS-2001-08: **Allard Tamminga**
Belief Dynamics. (Epistemo)logical Investigations
ILLC DS-2001-09: **Gwen Kerdiles**
Saying It with Pictures: a Logical Landscape of Conceptual Graphs
ILLC DS-2001-10: **Marc Pauly**
Logic for Social Software
ILLC DS-2002-01: **Nikos Massios**
Decision-Theoretic Robotic Surveillance
ILLC DS-2002-02: **Marco Aiello**
Spatial Reasoning: Theory and Practice
ILLC DS-2002-03: **Yuri Engelhardt**
The Language of Graphics
ILLC DS-2002-04: **Willem Klaas van Dam**
On Quantum Computation Theory
ILLC DS-2002-05: **Rosella Gennari**
Mapping Inferences: Constraint Propagation and Diamond Satisfaction
ILLC DS-2002-06: **Ivar Vermeulen**
A Logical Approach to Competition in Industries
ILLC DS-2003-01: **Barteld Kooi**
Knowledge, chance, and change
ILLC DS-2003-02: **Elisabeth Catherine Brouwer**
Imagining Metaphors: Cognitive Representation in Interpretation and Understanding
ILLC DS-2003-03: **Juan Heguiabehere**
Building Logic Toolboxes
ILLC DS-2003-04: **Christof Monz**
From Document Retrieval to Question Answering

ILLC DS-2004-01: **Hein Philipp Röhrig**
Quantum Query Complexity and Distributed Computing
ILLC DS-2004-02: **Sebastian Brand**
Rule-based Constraint Propagation: Theory and Applications
ILLC DS-2004-03: **Boudewijn de Bruin**
Explaining Games. On the Logic of Game Theoretic Explanations
ILLC DS-2005-01: **Balder David ten Cate**
Model theory for extended modal languages
ILLC DS-2005-02: **Willem-Jan van Hoeve**
Operations Research Techniques in Constraint Programming