JEPPE NEJSUM MADSEN
nejsum@diku.dk

Methods for Interactive Constraint Satisfaction

MASTER'S THESIS
February 14, 2003

DEPARTMENT OF COMPUTER SCIENCE
University of Copenhagen

Abstract

A constraint satisfaction problem involves the assignment of values to variables subject to a set of constraints. A large variety of problems in artificial intelligence and other areas of computer science can be viewed as a special case of the constraint satisfaction problem.

In many applications, one example being product configuration, user interaction is required to find a solution. The topic of this thesis is algorithmic methods for solving constraint satisfaction problems interactively.

A number of fundamental operations, which form the core of an interactive constraint solver, are identified and described formally. The decision version of the constraint satisfaction problem is $\mathcal{NP}$-complete, so a method of offline compilation is proposed to circumvent this intractability and achieve short response times for these fundamental operations.

Based on existing methods for tree clustering and solution synthesis, a compilation method is devised. A new method, based on *uniform acyclic constraint networks*, is proposed which results in improved response time of the fundamental operations.

All methods and algorithms have been implemented and their performance evaluated on real-life problem instances arising from the area of product configuration. The performance study shows that the new methods presented can achieve response times suitable for interactive processing for most of the problem instances.

CONTENTS

Preface

This Master's Thesis (Danish: *speciale*) is submitted to the University of Copenhagen, Department of Computer Science, as part of the author's work towards the M.Sc. (Danish: *cand.scient.*) degree in computer science. The thesis was written under the supervision of Jyrki Katajainen.

A major part of the work carried out during this project was the implementation of the methods described in this thesis. In order to save some trees, I have chosen to include the source code of this implementation on a supplemental CD-ROM. The source code can also be downloaded from:

`www.diku.dk/forskning/performance-engineering/jeppe/`

Audience

The intended audience of this report is computer scientists and computer science students with a background in algorithmics. I have done my best not to assume any prior knowledge of constraint satisfaction, but readers are required to have a basic knowledge of mathematics (especially set theory) and graph theory.

The notation used relies on relational algebra, and while the notation is defined in the report, some prior exposition to relational algebra may be useful.

Acknowledgments

I want to thank my supervisor Jyrki Katajainen for exposing me to the world of algorithmic research, for careful proofreading, and for pushing me in the right direction when needed.

Thanks must also go to my fellow students Jacob Blom Andersen and Kristian Hedeboe Hebert for proofreading and constructive criticism that helped improve the presentation, and to Array Technology A/S for providing real-life benchmark instances.

Last but not least, a big thank you to my wife Tina for just being there, staying sane, and keeping our family together during the past few months.

CHAPTER 1

Introduction

A constraint satisfaction problem (CSP) involves the assignment of values to variables subject to a set of constraints. A large variety of problems in Artificial Intelligence (AI) and other areas of computer science can be viewed as a special case of the constraint satisfaction problem. Examples include machine vision [Montanari, 1974; Mackworth, 1977a], belief maintenance [Dechter and Dechter, 1988], scheduling [Sycara et al., 1991], circuit design [de Kleer and Sussman, 1980], and natural language processing [Menzel, 1998].

The formal study of constraint satisfaction problems was initiated by Montanari [1974] who used *constraint networks* to describe combinatorial problems arising in image processing. A great deal of research in constraint satisfaction has focused on algorithms which, given a constraint network as input, automatically find a solution. This is useful in applications where, once the problem has been formulated as a constraint network, no user interaction is required. One such example is planning and scheduling (e.g., find a plan/schedule which satisfies the constraints). However, in many applications user interaction is required to find a solution. An example is when the user guides the assignment of values to variables. This interactivity restricts the amount of time that can be used for calculations between the user's selections. Constraint satisfaction problems are generally hard to solve (we will prove that the decision version of a CSP is $\mathcal{NP}$-complete), so all known solution methods have worst-case exponential time performance.

While CSPs are difficult to solve, one could hope for a situation akin to the results from Linear Programming where the widely used simplex algorithm [Dantzig, 1963] is shown to have exponential worst-case running time [Klee and Minty, 1972] but the intractable instances are hard to pro-

duce and almost never show up in real life problems. Unfortunately this does not seem to be the case. It is usually easy to come up with a constraint network which takes exponential time to solve. To solve a CSP interactively, we therefore have to circumvent this intractability in some way.

The topic of this thesis is algorithmic methods for solving constraint satisfaction problems interactively. Focus is on methods which are useful for solving practical problems arising in real life. I restrict the treatment to cover a subset of CSPs where the problem can be divided into two parts:

1. A static part in which the constraint network does not change between user interactions.

2. A dynamic part in which the user can influence the solution by adding additional constraints to the problem.

For problems in which the static part can be reused many times, this formulation allows considerable time to be spent preprocessing the static part of the network to speed up the dynamic operations where interactivity is required. As we will see, many problems of practical interest can be formulated using this restricted subset.

1.1 A Tour of Constraint Satisfaction Problems

Any constraint satisfaction problem involves *variables*. Each variable can be given a value chosen from a set of possible values called its *domain*. The *constraints* impose limitations on the values a variable, or a combination of variables, may be assigned. Together, variables, domains, and constraints form a *constraint network*. Other terms often used for a constraint network are (an instance of) a *constraint satisfaction problem* or a *constraint store*. A formal definition of constraint networks is given in Chapter 3 on page 17.

To give an idea of the wide variety of problems that can easily be formulated as a CSP, a number of well-known problems are listed and a possible constraint network representation is (informally) described. It is important to note that there is usually more than one way to model a problem. The choice between different formulations is sometimes arbitrary but can have great impact on the effort needed to solve the problem. The task of formulating problems using some well defined formalism is know as *modeling*.

1.1.1 Combinatorial Problems

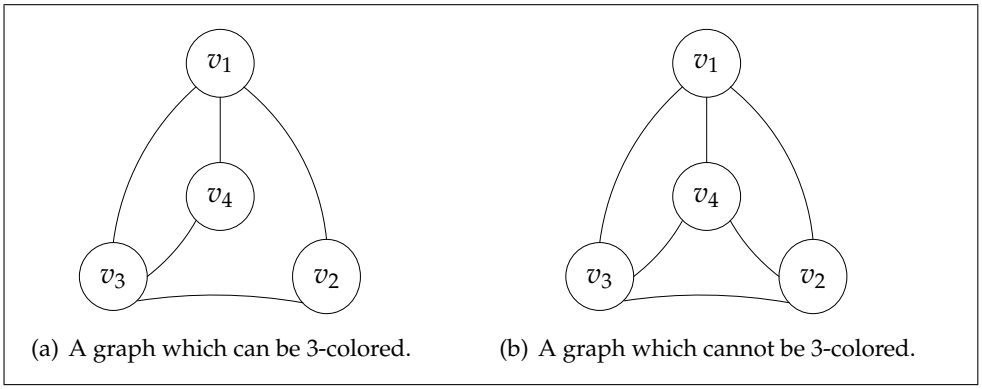
Many combinatorial problems are $\mathcal{NP}$-complete. In Section 3.2 we prove that this also holds for the decision version of the constraint satisfaction problem. It follows from the theory of $\mathcal{NP}$-completeness that (the decision version of) these problems can be converted (in polynomial time even) to a

CSP. However, as the following examples show, it is often quite natural to state combinatorial problems directly as a CSP.

Graph 3-Colorability

We are given an undirected graph $G = (V, E)$ with n vertices. The question is whether each vertex can be colored either red, green, or blue in such a way that two vertices connected with an edge have a different color. An example of different graphs can be seen in Figure 1.1.

To model this as a constraint network, we represent each node $v_i \in V$ with a variable x_i having the domain $\{R, G, B\}$. For each edge $\{v_i, v_j\} \in E$ we add the constraint $x_i \neq x_j$. If we can find a solution to this network, we have solved the problem.



(a) A graph which can be 3-colored. (b) A graph which cannot be 3-colored.

Figure 1.1: Graph 3-colorability examples.

1.1.2 Logic Puzzles

Much of the research in constraint satisfaction has been carried out by the AI community, and logic puzzles are often used as examples of problems which can be formulated as constraint satisfaction problems.

The n-Queen Problem

This is a classical problem from artificial intelligence. The goal is to place n queens ($n \geq 4$) on a chess board of size $n \times n$ such that no queen can attack another.

There are several ways to formulate this as a constraint network. One is to use a Boolean variable for each of the $n \times n$ cells on the chess board. If a variable is true, it means there is a queen in that cell. Instead I represent the ith column with an integer variable q_i. The value of q_i indicates the row where the ith queen should be placed. Using this representation, we

implicitly model the constraint (by creating n variables) that there shall be exactly one queen in each column.

The remaining constraints are:

$$|q_i - q_j| \neq i - j, \qquad \text{for } 1 \leq i \leq n, j > i \text{ and} \tag{1.1}$$

$$q_i \neq q_j, \qquad \text{for } 1 \leq i \leq n, j > i. \tag{1.2}$$

Constraint (1.1) states that queen i and j cannot be on the same diagonal and Constraint (1.2) that queen i and j cannot be on the same row.

A solution to the n-queen problem is thus an assignment of values in the range 1 to n to the variables $\{q_1, q_2, \ldots, q_n\}$ such that the constraints in (1.1) and (1.2) are satisfied. For $n = 8$ there are 92 possible solutions to this network [Erbas et al., 1992]. One such solution is shown in Figure 1.2.

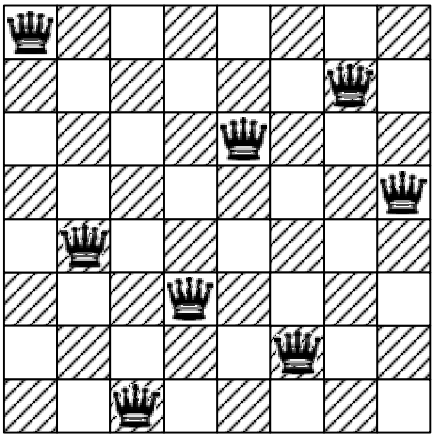


Figure 1.2: A solution to the 8-queen problem.

Who Owns the Zebra?

This problem is a classic example of a logic puzzle originally designed by the English logician Charles L. Dodgson (aka. Lewis Carroll). There are five houses, each of a different color and inhabited by people of different nationalities, with different pets (one is a zebra), drinks (one drink is water), and cigarettes. We are given the following clues:

1. In each house there lives only one person.
2. Each person has only one favorite drink, one pet, and smokes one brand of cigarettes.
3. The English person lives in the red house.
4. The Spaniard owns the dog.
5. Coffee is drunk in the green house.
6. The Ukrainian drinks tea.

7. The green house is immediately to the right (your right) of the ivory house.
8. The Old Gold smoker owns snails.
9. Kools are smoked in the yellow house.
10. Milk is drunk in the middle house.
11. The Norwegian lives in the first house on the left.
12. The person who smokes Chesterfields lives next to the house with the fox.
13. The person who smokes Kools lives next to the house with the horse.
14. The Lucky Strike smoker drinks orange juice.
15. The Japanese smokes Parliaments.
16. The Norwegian lives next to the blue house.

Questions: Who owns the zebra? Who drinks water?

To model this as a constraint network, we can assign the houses numbers 1 to 5 and use 5 variables for house $i, 1 \leq i \leq 5$:

$$C_i \in \{\text{red}, \text{green}, \text{ivory}, \text{yellow}, \text{blue}\}, \tag{1.3}$$
$$N_i \in \{\text{English}, \text{Spanish}, \text{Ukrainian}, \text{Norwegian}, \text{Japanese}\}, \tag{1.4}$$
$$P_i \in \{\text{zebra}, \text{dog}, \text{snails}, \text{fox}, \text{horse}\}, \tag{1.5}$$
$$D_i \in \{\text{water}, \text{coffee}, \text{tea}, \text{milk}, \text{juice}\}, \text{and} \tag{1.6}$$
$$S_i \in \{\text{Old Gold}, \text{Kools}, \text{Chesterfield}, \text{Lucky Strike}, \text{Parliaments}\}. \tag{1.7}$$

We then have that C_i is the color of the ith house, N_i the nationality of the person living in the ith house and so on. Because all houses have a different color, all persons a different nationality, etc., we get the following constraints:

$$C_i \neq C_j, N_i \neq N_j, P_i \neq P_j, D_i \neq D_j, S_i \neq S_j \quad 1 \leq i \leq 5, i < j \tag{1.8}$$

The remaining clues can also be specified as constraints. Take for example the third clue, which can be expressed as:

$$N_i = \text{English} \iff C_i = \text{red} \quad 1 \leq i \leq 5. \tag{1.9}$$

Constraint (1.9) states that if the nationality of the person living in the ith house is English then the ith house is red and vice versa. The remaining clues can be expressed in a similar way. The resulting network has a single solution in which $P_5 = \text{zebra}, N_5 = \text{Japanese}, D_1 = \text{water}$, and $N_1 = \text{Norwegian}$. So the answer is "The Japanese owns the zebra" and "The Norwegian drinks water".

1.2 Configuration

Configuration involves selecting combinations of predefined components subject to a number of problem constraints. The CSP paradigm offers an adequate framework for this task, and interactive configuration will be used as the working example throughout this thesis.

Configuration problems may involve sales (sales configuration), design, manufacturing, installation, or maintenance. The components involved need not be physical but can also be paragraphs of a legal document, financial services, actions in a plan, etc.

The possibility to automate the product configuration task using a configuration system[1] was recognized in the 1980's, and is now a rapidly growing industry. The following quote is from an article in Forbes Magazine about a large vendor of configuration systems. It illustrates the complexity of some of the problems solved by configuration systems.

> A [Boeing] 747 is made up of over 6 million parts, and a customer can choose among hundreds of options. How many seats? What kind of avionics? How many bathrooms? Do you want carbon composite landing gear or steel? Every option the customer chooses affects the availability of other options, and changes the plane's price. It takes the sales agent days or weeks working with company engineers to make sure all the chosen pieces fit together, re-negotiating the price at every step. [McHugh, 1996]

The preceding quote talks about the configuration of a large and complex product (a Boeing 747 airplane), probably sold to customers by a large team of salespersons and engineers. The ubiquitous use of the Internet has enabled individual customers to buy products manufactured to their liking directly from an online store. But even in a relatively simple product such as a Personal Computer (PC), there can be many dependencies among the available components. The picture in Figure 1.3 on the next page is from an online PC shop by one of the world's largest PC manufacturers (which shall remain anonymous). They appear not to be using a configuration system, but are instead relying on small textual notes on the screen advising the user about incompatible selections. The constraints that can be inferred from the notes are:

1. If you select more than three hard drives, you must select a hard drive bracket.

[1] Configuration systems are also known as product configuration systems (when used to configure products) or sales configuration systems (when used in a sales process). I use the more general term to emphasize that the usage is not restricted to the configuration of products nor the usage in a sales process

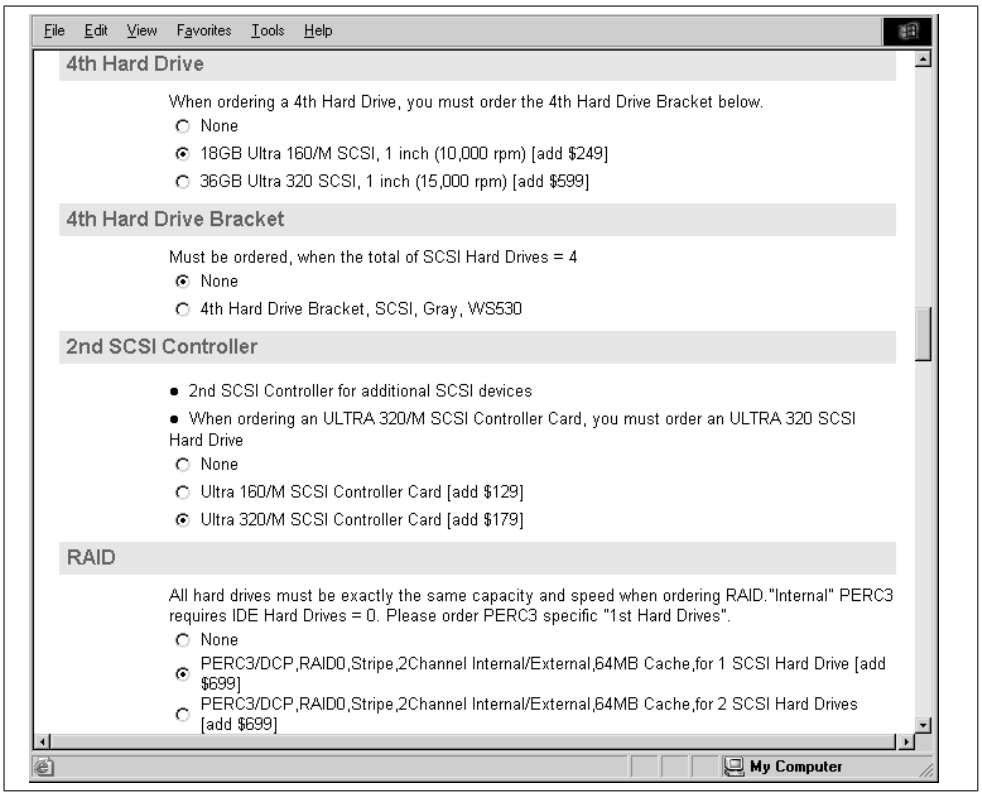


Figure 1.3: An example of an online store that could rely on a configuration system. Note how the incompatible selections are described using textual notes.

2. If you select an Ultra 320 SCSI Controller Card, you must select at least one Ultra 320 SCSI Hard Drive.

3. When ordering a RAID controller you must

   (a) select only drives which have the same size and speed,

   (b) select a 1st drive that is RAID compatible, and

   (c) not select any IDE drives.

Considering that the screen shot only shows a subset of the available options, it seems fairly easy to buy a product which will not function correctly. For both a customer and a manufacturer, it is very expensive to handle these types of configuration errors because manual intervention is required (call technical support, identify why the system does not work, perhaps return incorrect components, receive and install new necessary components, etc.). If instead the manufacturer had used a configuration system, these constraints would have been taken into account automatically and an invalid combination of components could not be shipped to the customer.

1.2.1 Elements of a Configuration System

At a high level, a constraint based configuration system is made up of two parts, as shown in Figure 1.4:

1. A modeling part where a *configuration model* is created. The model can be created manually by a user[2], automatically by extracting data from various enterprise systems or by a combination of the two approaches.

2. A runtime part where the end users of the configuration system interact with the configuration model to determine the configuration that suit their needs (whether it is a product, a service, or something completely different). The task of selecting the components is called the *configuration task*.

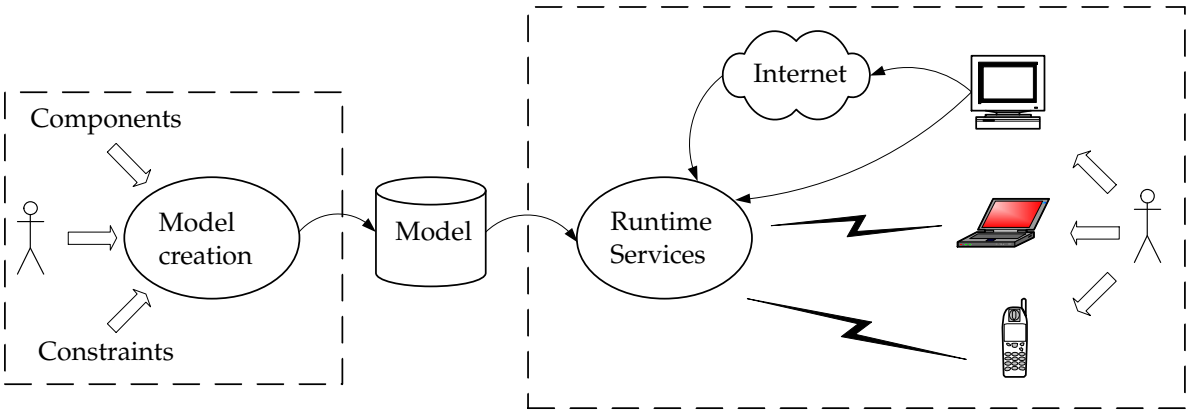


Figure 1.4: A high level view of a configuration system.

Creating a model involves specifying the components that are available and the constraints between these components. There are many possible ways to capture this knowledge. Constraint networks provide a convenient way to describe the components available as well as the constraints between them. In Section 4.4 on page 43, I will describe in more detail how a product model can be represented by a constraint network.

Changes to an already defined model involve changes in components (because some components are no longer available or new features become available to existing components) and/or changes to the constraints (because of new or changed components or because market requirements cause some combinations to be invalid). These changes occur infrequently relative to the number of configuration tasks carried out. This observation

[2]This is only a conceptual view. For large models many users may be involved in creating the model.

is important from an algorithmic point of view because it allows us to justify spending some time to preprocess a model to speed up the algorithms used during the configuration task.

The runtime part can come in many different incarnations depending on the target users of the system. If used by a company's customers, it may be a web application running in a browser and accessed using the Internet. If used by a company's salespersons, it may be an application running on the company's intranet or on the salesperson's laptop. If used by field engineers, it may be an application running on a wireless device such as a cellphone.

When the user has completed the configuration task, the resulting configuration is usually used in the next step of the business process. This may involve printing a quote, creating an order or maybe assembling a document based on the user's selections.

There are, of course, many other issues to an actual implementation of a configuration system (the actual process of capturing product knowledge, the language used for specifying this knowledge, how the model is distributed to the salesperson's laptop, what the model should look like when presented to the user, how the result of the configuration task should be stored, etc.). However, these issues are not essential to the contents of this thesis.

1.3 Contribution

I have identified a number of requirements that must be fulfilled by an interactive constraint solver, in order to provide a good user interface. Driven by these requirements I have identified three fundamental operations that form the basis of an interactive constraint solver. They are:

Add-Constraint: Add a new constraint to an existing constraint network.

Remove-Constraint: Remove a previously added constraint from a constraint network.

Restoration: Determine a set of constraints that restores satisfiability in a constraint network when incompatible constraints have been added.

I have proposed algorithms for the fundamental operations, which run in worst-case polynomial time when the constraint network is restricted to an *acyclic constraint network*. By combining existing methods for constraint network decomposition and solution synthesis, I have proposed a compilation scheme that compiles a general constraint network into an acyclic constraint network required by the fundamental operations.

While the algorithms for the fundamental operations are polynomial, experiments show that the acyclic networks constructed by the compilation scheme are, in many cases, too large to handle if short response times are required. To overcome this problem, I impose a further restriction on the structure of the constraint network and propose a transformation method that transforms an acyclic constraint network into a *uniform acyclic constraint network*. If a uniform acyclic network can be constructed, experiments show that the fundamental operations can be carried out in running-time suitable for interactive use.

All methods and algorithms presented in this thesis have been implemented and their performance evaluated on various constraint networks, many arising from real-life configuration problems. To my knowledge this is the first experimental study where performance of interactive constraint satisfaction methods have been evaluated on more than a single problem instance.

1.4 Related Work

Methods for solving static constraint satisfaction problems have been extensively studied. The book by Tsang [1993] covers many of the fundamental concepts and algorithms.

As noted in the preceding sections, most of the algorithms developed within the classical CSP framework cannot be used to solve interactive decision support systems. The classical (static) CSP framework have been extended with concepts that allow constraints to be added dynamically [Dechter and Dechter, 1988; Mittal and Falkenhainer, 1990]. The solution methods proposed, however, have not been directed towards interactive use.

The idea of compiling a constraint network into a form which allows more efficient processing is not new. Dechter and Pearl [1989] proposed a tree decomposition heuristic that transforms a constraint network into a tree structure. Vempaty [1992] introduced the idea of using a finite-state automaton that represents the set of all solutions to a constraint network. Amilhastre et al. [2002] continued this idea and described how this automaton could be used to solve constraint satisfaction problems interactively. Møller [1995] proposed a method for synthesizing all solutions to a constraint network and storing them compactly. Weigel and Faltings [1999] proposed a heuristic method, based on recursive spectral bisection, for compiling a CSP into a *minimal synthesis tree*.

A *Boolean Decision Diagram* [Bryant, 1986] is a canonical representation of a Boolean function using a directed acyclic graph. BDDs are widely used within the areas of circuit analysis and formal verification. A CSP can be viewed as a Boolean function that returns TRUE whenever a valid solution

is passed as input. Bouquet and Jégou [1997] proposed a method that used BDDs to solve dynamic constraint satisfaction systems.

Common for all these methods (as well as the methods proposed in this thesis) is that an initial constraint network is transformed into a form that allows efficient processing. Once the transformation is complete, efficient polynomial time algorithms are used subsequently. Unfortunately, the proposed transformations are all exponential in time (and some in space). This is to be expected since, as we will see, the decision version of the constraint satisfaction problem is $\mathcal{NP}$-complete. The applicability of the different methods thus depends on how well they are able to transform the constraint networks considered.

1.5 Outline of the Thesis

In Chapter 2 the definitions of general terms are given, the notation used throughout the thesis is fixed, and the basic definitions from graph theory and relational database theory are recalled. An experienced reader can skim this chapter.

Chapter 3 contains the formal definition of constraint networks, I prove that the decision version of a CSP is $\mathcal{NP}$-complete, and survey solutions methods for the classical CSP. The chapter ends with a discussion of conditions in which a CSP becomes tractable.

In Chapter 4, I identify a list of usability requirements for an interactive constraint satisfaction system and proceed with an extension of the classical CSP framework, which can be used to describe interactive constraint satisfaction problems. From these requirements and definitions, I identify operations that are fundamental for an interactive constraint satisfaction system and describe them formally. Polynomial-time algorithms, which operate on an *acyclic constraint network*, are then presented for the fundamental operations.

In Chapter 5, I describe an existing method for synthesizing solutions and an existing method for decomposing constraint networks, and show how they can be combined to transform any constraint network into an acyclic constraint network

Chapter 6 highlights the important parts of an implementation of the algorithms described in chapters 4 and 5. This chapter also contains the experimental results that have been obtained by running the implemented algorithms on several different constraint networks. The constraint networks used in the experiments consist of real life problems that have been encountered in various customer projects, problems mentioned in other research articles and logic puzzles.

In Chapter 7, I propose to further restrict the structure of an acyclic constraint network to obtain a *uniform acyclic constraint network* to improve

the response time of the fundamental operations. A tree transformation method is proposed which transforms an acyclic network into a uniform acyclic network. The chapter is concluded by an experimental evaluation of the proposed methods.

Chapter 8 concludes the thesis by highlighting the results and by proposing some further areas of research.

CHAPTER 2

Preliminaries

In this chapter the definitions of general terms are given, the notation used throughout the thesis is fixed and the basic definitions from graph theory and relational database theory are recalled. An experienced reader can skim this chapter.

2.1 Graph Theory

Definition 2.1. A *graph* $G = (V, E)$ is a structure where V is a finite set of *vertices* or *nodes* and E a finite set of *edges* or *arcs*[1]. For an *undirected graph* each edge is an unordered pair of vertices, and for a *directed graph* each edge is an ordered pair of vertices. □

Definition 2.2. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. □

Definition 2.3. A *clique* in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair which is connected by an edge in E. A clique is a *maximal clique* if it is not a proper subset of any other clique. □

Definition 2.4. A *hypergraph* $H = (V, S)$ is a structure where V is a finite set of vertices and S a finite set of *hyperedges*. Each hyperedge E is a subset of the vertices, i.e. $E \subseteq V$. A hypergraph is *reduced* if, and only if, no hyperedge is a proper subset of another. □

[1] I will frequently use the terms "node" and "arc" instead of "vertex" and "edge" respectively, to keep with the tradition in the AI community. This emphasizes the connection between the graph representation of constraint networks and some of the fundamental algorithms such as Node Consistency and Arc Consistency.

Definition 2.5. For a hypergraph $H = (V, S)$, the *primal graph* of H is an undirected multi-graph (V, E) where every two vertices joined by a hyperedge in S is joined by an edge in E. □

2.2 Relational Database Theory

As we will see in the following chapter, there is a close relationship between constraint networks and the relational data model used in database systems as initially defined by Codd in his seminal article [Codd, 1970]. Both use *relations* as the primary notation for representing data or knowledge. A *database* is a finite set of relations.

Definition 2.6. A *relation* consists of a *scheme* and an *instance*:

1. A *scheme* is a finite set of attributes. Each attribute is associated with a set of values, called its *domain*.
2. A *tuple* over a scheme is a mapping, that associates with each attribute of the scheme a value from its corresponding domain.
3. An *instance* over a scheme is a finite set of tuples over that scheme. □

Since relations are sets, the general set operations apply to relations with the restriction that both relations must have the same scheme. Given two relations R and S with the same scheme, the intersection of R and S, denoted $R \cap S$, is the relation containing tuples that are in both R and S; the union $R \cup S$ is the relation containing the tuples that are in either R or S or both, and the difference $R - S$ is the relation containing the tuples that are in R but not in S.

Many additional operations have been defined on relations. These operations are part of the *relational algebra* [Codd, 1970]. Of these, we will make use of projection and join which will be defined next.

Definition 2.7. Let R be a relation with scheme Y and $Z \subseteq Y$ a set of attributes. Let r be an instance over Y. The *projection* of r onto Z, denoted $\pi_Z(r)$, is a relation with scheme Z and instance $\{t_{|Z} \mid t \in r\}$ where $t_{|Z}$ denotes the tuple formed from t by keeping only those components associated with the attributes in Z. □

Definition 2.8. Let R be a relation with scheme Y and instance r. Let S be a relation with scheme Z and instance s. The *join*[2] of R and S, denoted $R \bowtie S$,

[2]The operation defined here is sometimes referred to as *natural join* to distinguish from the more general *theta join*. Here we use the short form since it does not give rise to confusion.

is defined to be the relation with scheme $Y \cup Z$ and an instance containing the following set of tuples:

$$\{t \mid t \text{ is a tuple over } Y \cup Z\,, t_{|Y} \in r, t_{|Z} \in s\}.$$

Again, $t_{|Z}$ denotes the tuple t restricted to Z. □

The projection operator is used to remove certain components of the tuples in a relation, and the join operator is used to combine two relations on all their common variables. If there are no common variables, the join operator behaves as a Cartesian product. Since the results of the operators are again relations, the operators can be combined.

2.2.1 Representing Relations

In Definition 2.6 on the facing page, schemes and tuples are formally defined in terms of sets. This allows specifying tuples without fixing the attribute order. Consider a relation with scheme $\{a, b, c\}$ and the domain of each attribute being the set of integers. A tuple over this scheme is $\{b \mapsto 2, c \mapsto 3, a \mapsto 1\}$, where $b \mapsto 2$ is used to denote that the attribute b maps to the value 2.

For notational convenience, however, I will express tuples as ordered sequences, with an implied ordering for the attributes of each relation. Thus with the attribute ordering (a, b, c) the same tuple is expressed as $(1, 2, 3)$. A relation is usually depicted using a table with the first row representing the attributes, a horizontal line and a row for each tuple in the instance:

a	b	c
1	2	3

Similarly, the relation instance is expressed as an ordered sequence of tuples and $R[i]$ denotes the ith tuple of relation instance R. The number of tuples in a relation instance is denoted by $|R|$.

Example 2.1. The result of applying the projection and join operators to two relations R and S is shown in Figure 2.1 on the next page. □

2.3 Model of Computation

The model of computation used is the unit-cost *word RAM*. For a positive integer parameter w, called the *word length*, the memory cells of a word RAM store w-bit words, viewed as integers in $\{0, \ldots, 2^{w-1}\}$ or as bit vectors in $\{0, 1\}^w$. Standard operations, including addition, bitwise Boolean

a	b
0	0
0	1
1	0
1	1

(a) Relation R

b	c
0	4
1	5
2	6
3	7

(b) Relation S

a
0
1

(c) $\pi_a(R)$

a	b	c
0	0	4
1	0	4
0	1	5
1	1	5

(d) $R \bowtie S$

Figure 2.1: Applying projection and join operations on relations.

operations, shifts and multiplication, can be carried out on words in constant time. The space requirements of a word-RAM algorithm is measured in units of w-bit words.

The word RAM is a natural and realistic model of computation and has been the object of much recent research (see, e.g., [Hagerup, 1998]). A detailed definition of the model can also be found in [Hagerup, 1998].

CHAPTER 3

Constraint Satisfaction Problems

In the present chapter, the terminology, concepts and definitions relating to constraint satisfaction problems are introduced and some of the fundamental algorithms for solving them are surveyed.

Due to the diversity of research, much of the terminology related to constraint satisfaction is unfortunately ambiguous. I have chosen the definitions to ease the presentation of the subsequent material while keeping in agreement with most of the existing literature.

3.1 Constraint Networks

An instance of a constraint satisfaction problem can be described by a *constraint network*[1].

Definition 3.1 (Constraint Network). A *constraint network* is a triple $\mathcal{R} = (X, D, C)$ where

1. X is a finite set of variables.
2. D is a function that maps each variable x in X to a finite set of values, written $D(x)$, which it is allowed to take. The set $D(x)$, called the *domain* of x, is also denoted D_x.
3. C is a finite set of constraints. Let $S = \{x_k, \ldots, x_\ell\} \subseteq X$. Each constraint $C_S \in C$ is a relation with scheme S and instance C_S. The set

[1]The term "network" is used to reflect the historical perspective when focus was restricted to constraints whose dependencies can naturally be captured by simple graphs as well as to emphasize the importance of the constraint dependency structure to the solution algorithms.

S is the *scope* of the constraint, and $|S|$ denotes the *arity* of the constraint. Each tuple in the instance $C_S \subseteq D_{x_k} \times \cdots \times D_{x_\ell}$ specifies a combination of values which the constraint allows. □

In general, domains can be infinite which in turn implies that the constraint relations can be infinite, but we only consider the case where variables have finite domains (sometimes called finite domain constraint satisfaction problems). The constraint representation chosen in Definition 3.1 on the preceding page is *extensional*, i.e., an explicit enumeration of the tuples that are allowed by the constraint. Other representations are possible, e.g., an *intensional* representation where a constraint is represented in symbolic form such as predicate logic.

Without loss of generality I assume that all constraints have a unique scope, i.e., for all $C_S, C_R \in C, C_S \neq C_R$ we have $S \neq R$. To simplify the notation, I usually denote the constraint $C_{\{x,y,z\}}$ as $C_{x,y,z}$ or even C_{xyz}, when no confusion can arise from this simplified notation. A constraint with arity one is called a *unary* or *domain constraint* (since it restricts the possible values of a variable). A constraint with arity two is called *binary*. A network with only binary constraints is called a *binary constraint network*, otherwise it is called a general constraint network.

Definition 3.2. Let $\mathcal{R} = (X, D, C)$ be a constraint network. An *assignment* of the value $a \in D_x$ to the variable $x \in X$ is denoted $\langle x, a \rangle$. An *instantiation* of a set of variables $\{x_k, \ldots, x_\ell\} \subseteq X$ is a simultaneous assignment of values to the variables $\{x_k, \ldots, x_\ell\}$ and is denoted $\{\langle x_k, a_k \rangle, \ldots, \langle x_\ell, a_\ell \rangle\}$. □

For simplicity, we often perceive an instantiation as a tuple $(a_k, \ldots, a_\ell)$ over the scope $\{x_k, \ldots, x_\ell\}$ and denote the instantiation briefly as $\bar{a}_{\{x_k,\ldots,x_\ell\}}$ or simply $\bar{a}$ if the scope is well defined or unimportant.

Definition 3.3. An instantiation $\bar{a}$ *satisfies* a constraint C_S if $\bar{a}_{|S} \in C_S$. Let $\mathcal{R} = (X, D, C)$ be a constraint network. An instantiation $\bar{a}_T$, where $T \subseteq X$, is *consistent* relative to $\mathcal{R}$ if, and only if, $\bar{a}_T$ satisfies all constraints $C_S \in C$ such that $S \subseteq T$. □

Definition 3.4. A *solution* of the constraint network $\mathcal{R} = (X, D, C)$ is an instantiation of all variables in X which is consistent relative to $\mathcal{R}$. The set of all solutions to a constraint network $\mathcal{R}$ is denoted $Sol(\mathcal{R})$. □

A network $\mathcal{R}$ is called *satisfiable* if $Sol(\mathcal{R}) \neq \emptyset$ and *unsatisfiable* if $Sol(\mathcal{R}) = \emptyset$. Two networks defined on the same set of variables are considered *equivalent* if they have the same set of solutions. A constraint is *redundant* if removal of the constraint does not change the set of all solutions. For a constraint network $\mathcal{R} = (X, D, C)$, any subset of variables $I \subseteq X$ induces a *subnetwork of* $\mathcal{R}$ which has variables I, domains D and the set of constraints is a subset of C:

$$\{C_S \mid C_S \in C, S \cap I \neq \emptyset\}. \tag{3.1}$$

Given a constraint network $\mathcal{R}$, a CSP can be classified into the following categories:

1. Determine whether the network $\mathcal{R}$ is satisfiable.
2. Find a solution to the network $\mathcal{R}$, with no preference as to which one.
3. Find the set of all solutions $Sol(\mathcal{R})$.
4. Find an optimal solution to the network $\mathcal{R}$, where optimality is defined by a function on the variables in $\mathcal{R}$.

Optimal or approximatively optimal solutions are often required in planning and scheduling problems where the objective is to, e.g., minimize the time required to finish a schedule.

3.1.1 Constraint Network Properties

A constraint network can be characterized by various parameters.

n**:** the number of variables, $|X|$,

d**:** the size of the largest domain, $\max_{x \in X} |D_x|$,

e**:** the number of constraints, $|C|$,

r**:** the size of the largest scope, $\max_{C_S \in C} |S|$, and,

t**:** the largest number of tuples in any constraint relation, $\max_{C_S \in C} |C_S|$.

Definition 3.5. The *tightness of a constraint* C_S is the ratio between the number of tuples in the constraint relation and the number of possible instantiations of the variables in S

$$\frac{|C_S|}{\prod_{x \in S} |D_x|}. \tag{3.2}$$

□

The *universal constraint* over the variables S, denoted $\mathcal{U}_S$, allows every instantiation of the variables in the scope S. It is the most relaxed constraint possible and has tightness 1.

In general, the number of tuples in a constraint with scope S is bounded by $\prod_{x \in S} |D_x|$, which is the number of tuples in the universal constraint with scope S. Frequently, constraints are quite tight. For binary constraints we have, e.g., the universal constraint with $t \leq d^2$, but functional constraints have $t \leq d$.

Definition 3.6. The *tightness of a constraint network* $\mathcal{R} = (X, D, C)$ is measured by the number of solutions over the number of possible instantiations for all variables

$$\frac{|Sol(\mathcal{R})|}{\prod_{x \in X} |D_x|}. \tag{3.3}$$

□

Definition 3.7. The *size* of a CSP described by a constraint network $\mathcal{R} = (X, D, C)$ is defined as

$$\sum_{C_S \in C} |S||C_S| = \mathcal{O}(ert). \tag{3.4}$$

□

Example 3.1. Let us construct a simple constraint network to illustrate the definitions. As an example we use the 4-queen problem introduced in Section 1.1.2 on page 3. As noted we have 4 integer variables, one for each queen. So we have $X = \{q_1, q_2, q_3, q_4\}$ and $D_x = \{1, 2, 3, 4\}$ for each $x \in X$. The two constraints in Equation (1.1) and Equation (1.2) can be combined so we get a total of six binary constraints. The constraints are shown in the following where the notation C_{12} is used to denote $C_{\{q_1, q_2\}}$:

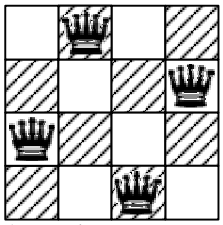
$$\begin{aligned}
C_{12} &= \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\} \\
C_{13} &= \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\} \\
C_{14} &= \{(1,2), (1,3), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,2), (4,3)\} \\
C_{23} &= \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\} \\
C_{24} &= \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\} \\
C_{34} &= \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}.
\end{aligned}$$

There are only two solutions to the 4-queen problem as shown in Figure 3.1(a) and Figure 3.1(b). Figure 3.1(c) on the facing page shows a consistent instantiation that cannot be extended to a full solution. □

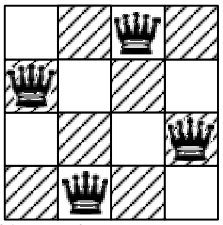
3.1.2 Links with Relational Database Theory

As noted in [Gyssens et al., 1994] the definitions given so far have close connections to relational database theory. For any constraint network $\mathcal{R} = (X, D, C)$, the following connections can be established:

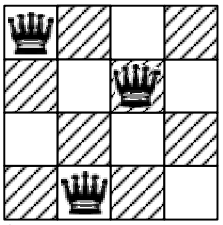
- The variables in X can be interpreted as attributes.
- The domain associated with an attribute is the domain of the corresponding variable in X.



(a) Solution corresponding to $\{3,1,4,2\}$.



(b) Solution corresponding to $\{2,4,1,3\}$.



(c) Consistent instantiation not part of a solution.

Figure 3.1: Some consistent instantiations of the 4-queen problem. Not all consistent instantiations can be extended to a full solution.

- The valid combinations of values with scope $S \subseteq X$ is a tuple over the scheme with the set of attributes S.

This gives rise to two alternative views of a constraint network $\mathcal{R} = (X, D, C)$.

1. The set of all solutions can be represented by the database consisting of a single relation with scheme X and instance $Sol(\mathcal{R})$.
2. The constraint network can be represented by the database $\{R_{C_S} \mid C_S \in C\}$ where R_{C_S} is a relation with scheme S and instance C_S.

It follows from these definitions that for a constraint network $\mathcal{R} = (X, D, C)$, $Sol(\mathcal{R})$ is equal to the relation $\bowtie_{C_S \in C} C_S$ (the join of all relations in C). Table 3.1 summarizes the terminology.

Constraint Terminology		Database Terminology
constraint network	$\equiv$	database
variable	$\equiv$	attribute
domain	$\equiv$	attribute domain
constraint	$\equiv$	relation
constraint scope	$\equiv$	scheme
constraint relation	$\equiv$	instance
set of solutions	$\equiv$	join of all tables

Table 3.1: Constraint and database terminology. Adapted from [Pearson and Jeavons, 1997].

3.1.3 Structure of Constraint Networks

The structure, or topology, of a constraint network can be described using various objects as the following definitions show.

Definition 3.8 (Primal constraint graph). The *constraint graph* of a constraint network $\mathcal{R} = (X, D, C)$ is an undirected graph in which each node represents a variable in X and there is an arc between any two variables that are related by a constraint. The constraint graph is also called a *primal constraint graph* [Dechter and Pearl, 1989]. □

The term *constraint graph* is sometimes used only for graphs representing binary constraint networks, but in this thesis I will use the general definition which is well defined for both binary and general constraint networks. For binary networks, however, there is a direct association between arcs and constraints. To maintain this association for general constraint networks, we need a hypergraph representation.

Definition 3.9 (Constraint hypergraph). The *constraint hypergraph* of a constraint network $\mathcal{R} = (X, D, C)$ is a hypergraph in which each node represents a variable in X and for each constraint $C_S \in C$ there is a hyperedge S that represents the constraint. □

Definitions 3.8 and 3.9 are of course closely related: In the case of a binary network, both the primal constraint graph and the constraint hypergraph reduce to an undirected graph with a node for each variable and an arc for each constraint. For some problems, such as the graph 3-colorability problem described in Section 1.1.1 on page 3, there is a direct correspondence between the problem instance and the constraint graph.

Definition 3.10 ([Dechter and Pearl, 1989]). The *dual constraint graph* of a constraint network $\mathcal{R} = (X, D, C)$ is an undirected graph in which each node represents a constraint. There is an arc between any two nodes sharing a common variable. The arcs are labeled by the shared variables. □

The dual constraint graph can be used to transform any non-binary constraint network into a special type of binary constraint network called the *dual network*.

Definition 3.11 (Dual Network). Given a constraint network $\mathcal{R} = (X, D, C)$, the *dual network* is a binary constraint network $\mathcal{R}^d = (X^d, D^d, C^d)$ where

1. X^d is a set of *dual variables*. There is a dual variable for each constraint $C_S \in C$ where each dual variable represents a subset S of variables from X
$$X^d = \{S \mid C_S \in C\}. \tag{3.5}$$
2. D^d is a function that maps each dual variable $S \in X^d$ to the *dual domain* D^d_S. The dual domain D^d_S is the set of tuples in the relation of the original constraint $C_S \in C$
$$D^d_S = \{t \mid t \in C_S\}. \tag{3.6}$$

3. Any two dual variables sharing an original variable must obey the restriction that their shared original variables must have the same values. The constraints in C^d are thus

$$C^d = \{C^d_{S,T} \mid S, T \in X^d, S \cap T \neq \emptyset\} \tag{3.7}$$

where $C^d_{S,T}$ is the binary constraint

$$C^d_{S,T} = \{(u, v) \mid u \in C_S, v \in C_T, u_{|S \cap T} = v_{|S \cap T}\}. \tag{3.8}$$

□

Thus all the constraints in the dual network are in some sense equality constraints. It should be clear that if we found a solution to the dual network we would also, by mapping the solution back to the original variables, find a solution to the original problem. In this way all general constraint networks can be transformed into binary networks and solved using binary network techniques. This is useful, since many CSP algorithms are defined only for binary networks. Other encodings exist for transforming a general network to a binary network. Bacchus et al. [2002] compare some of the encodings when used in conjunction with search algorithms.

Example 3.2. Consider the following constraint network with five variables and three constraints:

$$\begin{aligned} X &= \{model, case, ide, scsi, cpu\} \\ D_{model} &= \{\text{home}, \text{office}\} \\ D_{case} &= \{\text{desktop}, \text{tower}\} \\ D_{ide} &= \{\text{none}, \text{40gb}, \text{80gb}\} \\ D_{scsi} &= \{\text{none}, \text{18gb}, \text{36gb}\} \\ D_{cpu} &= \{\text{PIII}, \text{PIV}, \text{AMD}\} \\ C &= \{C_{\{model,cpu,case\}}, C_{\{ide,scsi\}}, C_{\{case,scsi\}}\} \end{aligned}$$

model	*cpu*	*case*
home	PIII	desktop
home	AMD	desktop
office	PIII	desktop
office	PIII	tower
office	PIV	desktop
office	PIV	tower
office	AMD	desktop
office	AMD	tower

ide	*scsi*
none	18gb
none	36gb
40gb	none
80gb	none

case	*scsi*
desktop	none
tower	none
tower	18gb
tower	36gb

The different graph representations for this constraint network are shown in Figure 3.2 on page 25. The dual network is a constraint network with

three variables: one for each constraint in the original problem, and two binary constraints: one for each edge in the dual constraint graph. The domains of the dual variables are the tuples of the corresponding original constraints. Note that if we depicted the primal graph of the dual problem, the topology would be equivalent to the dual graph of the original problem.

$$
\begin{aligned}
X^d &= \{\{model, cpu, case\}, \{ide, scsi\}, \{case, scsi\}\} \\
D^d_{\{model,cpu,case\}} &= C_{\{model,cpu,case\}} \\
D^d_{\{ide,scsi\}} &= C_{\{ide,scsi\}} \\
D^d_{\{case,scsi\}} &= C_{\{case,scsi\}} \\
C^d &= \{C^d_{(\{model,cpu,case\},\{case,scsi\})}, C^d_{(\{case,scsi\},\{case,ide\})}\}
\end{aligned}
$$

$\{model, cpu, case\}$	$\{case, scsi\}$
(home, PIII, desktop)	(desktop, none)
(home, AMD, desktop)	(desktop, none)
(office, PIII, desktop)	(desktop, none)
(office, PIV, desktop)	(desktop, none)
(office, AMD, desktop)	(desktop, none)
(office, PIII, tower)	(tower, none)
(office, PIV, tower)	(tower, none)
(office, AMD, tower)	(tower, none)
(office, PIII, tower)	(tower, 18gb)
(office, PIV, tower)	(tower, 18gb)
(office, AMD, tower)	(tower, 18gb)
(office, PIII, tower)	(tower, 36gb)
(office, PIV, tower)	(tower, 36gb)
(office, AMD, tower)	(tower, 36gb)

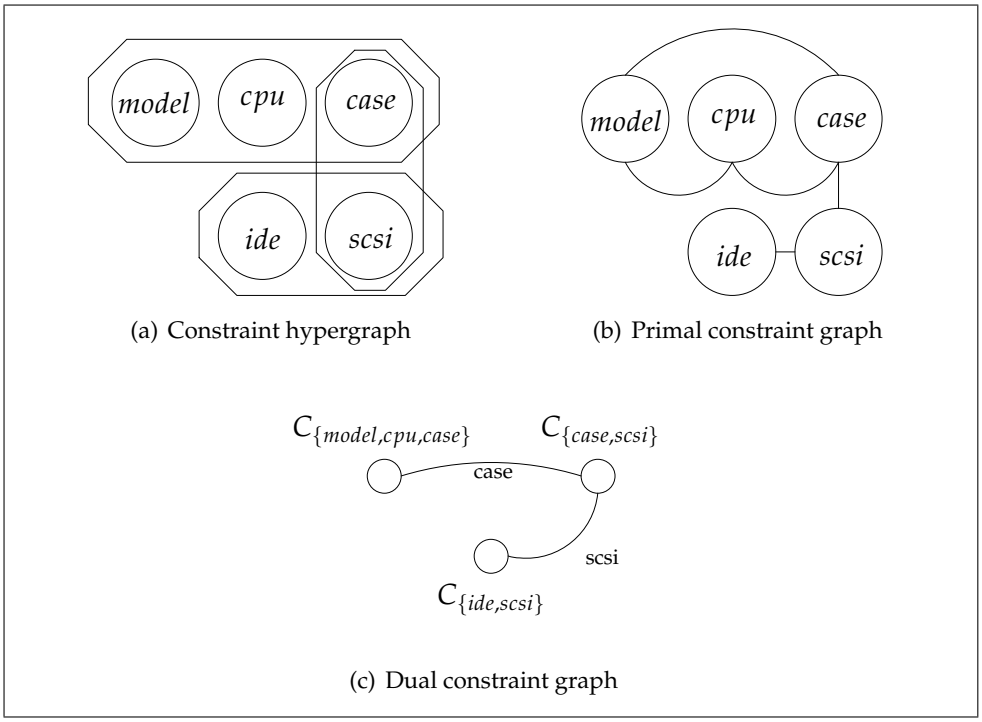
$\{case, scsi\}$	$\{ide, scsi\}$
(desktop, none)	(40gb, none)
(desktop, none)	(80gb, none)
(tower, none)	(40gb, none)
(tower, none)	(80gb, none)
(tower, 18gb)	(none, 18gb)
(tower, 36gb)	(none, 36gb)

□

3.2 Complexity of Constraint Satisfaction

Given a CSP, we can derive a corresponding decision problem:

Definition 3.12 (CONSTRAINT SATISFIABILITY). The decision version of a constraint satisfaction problem has a triple (X, D, C) as an input, where X,



(a) Constraint hypergraph

(b) Primal constraint graph

(c) Dual constraint graph

Figure 3.2: Different graph representations of the constraint network in Example 3.2.

D and C are as in Definition 3.1, and yields "yes" or "no" as an output. The output is "yes" if the constraint network (X, D, C) has a solution and "no" otherwise. This decision problem is called the CONSTRAINT SATISFIABILITY (CS) problem. □

Clearly, finding any solution to a constraint network, can be used to solve the corresponding decision problem, i.e., finding a solution, all solutions or an optimal solution is as difficult as CS. In the following, we will prove that CS is $\mathcal{NP}$-complete. First we need to introduce a well-known $\mathcal{NP}$-complete decision problem (see, e.g., Cormen et al. [2001] for more details):

Definition 3.13 (3-CNF-SAT). An input of 3-CNF-SAT is a Boolean formula composed of

1. Boolean variables: $x_1, x_2, \ldots$;
2. Boolean connectives such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT); and
3. parentheses.

A *literal* in a Boolean formula is an occurrence of a variable or its negation. A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is expressed

as a logical AND of clauses, each of which is the logical OR of one or more literals. A Boolean formula is in *3-conjunctive normal form*, or *3-CNF*, if each clause has exactly three distinct literals.

In 3-CNF-SAT, we are asked whether a given Boolean formula in 3-CNF has a satisfying assignment, i.e., a set of values for the variables that causes the formula to evaluate to true. □

Lemma 3.1. *3-CNF-SAT is polynomial-time reducible to CS.*

Proof. Let $\theta = C_1 \wedge C_2 \wedge \cdots C_k$ be a Boolean formula in 3-CNF with k clauses. From θ we need to create the input (X, D, C) to the decision problem CS, where X is the variables, D the variable domains and C the constraints. It is created as follows.

1. For each distinct variable x_i in θ, we add x_i to X.
2. For each variable x_i in X, we let $D_{x_i} = \{0, 1\}$.
3. For each clause $C_r = (l_1 \vee l_2 \vee l_3)$ in θ, we create a constraint C_S. The scope S contains the (not necessarily distinct) variables appearing in the literals l_1, l_2 and l_3. The constraint relation C_S contains a 3-tuple for each satisfying assignment to C_r.

This reduction is linear in the number of clauses in θ since there are at most $3k$ variables, k constraints, and each constraint relation has at most $2^3 = 8$ tuples. It is clear that θ is satisfiable if, and only if, the constraint network (X, D, C) has a solution. □

Theorem 3.2. *CS is $\mathcal{NP}$-complete.*

Proof. Let $\mathcal{R} = (X, D, C)$ be the input of a CSP decision problem where X is the set of variables, D their domains and C the set of constraints. First we note that CS is in $\mathcal{NP}$: Given an instantiation $\bar{a}$ of all variables in X, verify $a_{|S} \in C_S$ for each $C_S \in C$. The verification can be done in time polynomial in the size of $\mathcal{R}$. By $\mathcal{NP}$-completeness of 3-CNF-SAT and Lemma 3.1, CS is $\mathcal{NP}$-complete. □

3.3 CSP Solution Methods

A CSP can be solved using the generate-and-test method (also know as "the British Museum Algorithm" according to Hoare [1989]), where each possible instantiation of the variables is systematically generated and then tested to see if it satisfies all the constraints. The first such instantiation found is a solution. The number of instantiations generated in the worst case (which occurs when the network is unsatisfiable) is $\mathcal{O}(d^n)$, i.e., exponential in the number of variables. Figure 3.3(a) on page 28 shows the search tree when a generate-and-test algorithm is applied to an unsatisfiable problem.

By using backtracking [Bitner and Reingold, 1975], a potentially more efficient method can be constructed. During search, variables are instantiated according to some ordering called the *instantiation order*, and when all variables in a constraint have been instantiated, the constraint is checked. Whenever a partial instantiation violates a constraint, backtracking is performed to the most recently instantiated variable with uninstantiated values left in the domain, thereby eliminating a subspace from the Cartesian product of the variable domains. A naive backtracking algorithm (usually called *chronological backtracking*) is shown in Algorithm 3.1.

```
BACKTRACK(ā, X, D, C)
 1  ▷ ā is the current instantiation
 2  if X = ∅
 3     return ā
 4  else
 5     x ← some variable from X
 6     repeat
 7        v ← some value from D_x
 8        D_x ← D_x − {v}
 9        if ā ∪ {⟨x, v⟩} is consistent
10           r̄ ←BACKTRACK(ā ∪ {⟨x, v⟩}, X − {x}, D, C)
11           if r̄ ≠ NIL
12              return r̄
13     until D_x = ∅
14     return NIL
```

Algorithm 3.1: A simple backtracking solver.

Given a network $\mathcal{R} = (X, D, C)$ the call BACKTRACK$(\emptyset, X, D, C)$ returns a solution if the network is satisfiable, otherwise NIL is returned. While BACKTRACK still generates $\mathcal{O}(d^n)$ instantiations in the worst case, the ability to eliminate subspaces of the search tree makes it more efficient in most cases. Figure 3.3(b) on the next page shows the search tree when BACKTRACK is applied to an unsatisfiable problem. The left subtree illustrates the search space which is eliminated because the first variable assignment is inconsistent. It should be clear from this figure that the ordering in which variables are instantiated is important. In this example, if the first assignment was instead last, all instantiations would have to be tested, i.e. BACKTRACK deteriorates to generate-and-test.

There are two drawbacks in the standard backtracking scheme presented here. One is *trashing* [Gaschnig, 1979] where search fails repeatedly for the same reason. If, for example, the constraint C_{x_i,x_k} specifies that a particular assignment to x_i disallows all potential values for x_k then BACKTRACK fails when instantiating x_k for all values in D_{x_k}, repeating this failure for all instantiations of variables x_j for $i < j < k$. The other drawback

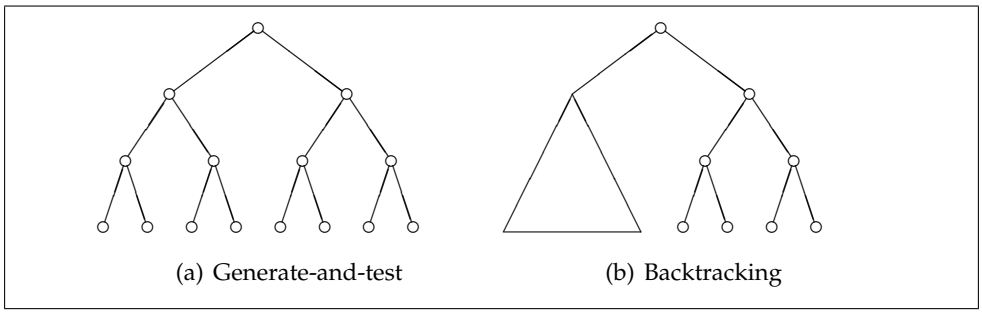


Figure 3.3: Search tree example for an unsatisfiable network. Each edge represents an assignment of a value to a variable, each level in the tree corresponds to a variable. In (a) all combinations are tested before unsatisfiability is detected. In (b) all combinations in the left branch are skipped since the first assignment to the first variable is found to be inconsistent.

is having to perform redundant work. If, for example during a constraint check, an instantiation of a subset of variables is found to be consistent and then deeper in the search tree, the instantiation has not changed; then there is no need to check these variables again.

Many variations on the naive backtracking scheme have been proposed to cope with these drawbacks as well as finding a good instantiation order. While theses methods differ in the number of constraint checks they must perform, they are all exponential in the worst case. For all but the smallest problems this makes them unsuitable for use in an interactive setting. I will therefore not go into more detail about search techniques, but refer interested readers to the papers by Miguel and Shen [2001a,b] which contain a survey on many of the known search strategies. Kondrak and van Beek [1997] present a theoretical evaluation of several backtracking algorithms.

3.3.1 Consistency in Binary Networks

Preprocessing a constraint network creates an equivalent network that is easier to solve, usually using some search algorithm. Algorithms that perform preprocessing are often called *consistency enforcing*, *inference* or *constraint propagation* algorithms.

The simplification involves removing domain values that can never be part of a solution, simplifying constraints by removing tuples that can never be part of a solution or introducing new tighter constraints that can be inferred by propagating information from existing constraints. The most widely used levels of consistency in binary networks are called *node*, *arc* and *path* consistency, and were defined by Mackworth [1977b].

Node Consistency

Definition 3.14. A variable is *node consistent* if all values in the domain satisfy the unary constraint on that variable[2]. A constraint network is node consistent if all variables are node consistent. □

Achieving node consistency amounts to checking all domain values to see if they satisfy the unary constraint on the corresponding variable. We can use relational algebra to specify node consistency, in which case we get

$$D_x \leftarrow D_x \cap C_x, \quad \text{for all } x \in X, C_x \in C, \tag{3.9}$$

where we have used the fact that the domain of a variable x can be viewed as a relation with scope x and instance D_x.

An algorithm for making a network node consistent is shown in Algorithm 3.2. The time complexity of NC is $\mathcal{O}(dn)$ if we assume the constraint check $a \notin C_x$ takes constant time.

```
NC(X, D, C)
1  for each x ∈ X
2     for each a ∈ D_x
3        if a ∉ C_x
4           D_x ← D_x − {a}
```

Algorithm 3.2: Make network (X, D, C) node consistent.

Arc Consistency

Definition 3.15. Let $\mathcal{R} = (X, D, C)$ be a node consistent constraint network with $C_{xy} \in C$. A variable x is *arc consistent* relative to y if, and only if, for every value $a \in D_x$ there exists a value $b \in D_y$ such that $(a, b) \in C_{xy}$. An arc $\{x, y\}$ in the constraint graph of $\mathcal{R}$ is arc consistent if and only if x is arc consistent relative to y and y is arc consistent relative to x. A constraint network is arc consistent if, and only if, all arcs are arc consistent. □

Consider an instantiation $\langle x, a \rangle$ of some variable x. If there is a constraint C_{xy} and there is no value $b \in D_y$ such that $(a, b) \in C_{xy}$, then we can remove the value a from D_x without affecting any solution.

Arc consistency algorithms have received a great deal of attention in the CSP literature. As a result, a number of algorithms for achieving arc consistency have been proposed. They have traditionally been named AC-n where n increases with each improvement or specialization of previous algorithms.

[2]Recall that we assumed that all constraint scopes are unique. This implies that there can be at most one unary constraint on any given variable.

Algorithm REVISE shown in Algorithm 3.3, makes a variable x arc consistent relative to a variable y, by removing values from the domain of x that cannot be part of a solution when considering the constraint C_{xy}. We can again use relational algebra to describe the result of REVISE:

$$D_x \leftarrow D_x \cap \pi_x(C_{xy} \bowtie D_y). \tag{3.10}$$

Algorithm REVISE can be used to create an algorithm for achieving arc consistency. A naive brute force arc consistency algorithm called AC-1 (first proposed by Mackworth [1977b]) is shown in Algorithm 3.4.

```
REVISE(x, y, X, D, C)
1  for each a ∈ D_x
2    for each b ∈ D_y
3      if (a, b) ∉ C_xy
4        D_x ← D_x − {a}
```

Algorithm 3.3: Make x arc consistent relative to y.

```
AC-1(X, D, C)
1  repeat
2    for each C_xy ∈ C
3      REVISE(x, y, X, D, C)
4      REVISE(y, x, X, D, C)
5  until no domain is changed
```

Algorithm 3.4: A naive algorithm for achieving arc consistency.

The time complexity of REVISE is $\mathcal{O}(d^2)$ if we assume the constraint check $(a, b) \notin C_{xy}$ takes constant time. The calls to REVISE in lines 2–4 thus take time $\mathcal{O}(ed^2)$ and may in the worst case delete only one domain value. There are $\mathcal{O}(nd)$ domain values, so the worst-case time complexity of AC-1 is $\mathcal{O}(ned^3)$.

Several improvements to the naive AC-1 algorithm have been proposed. When a value is removed from a domain, AC-1 checks all constraints again. AC-3 [Mackworth, 1977b] improves the running time to $\mathcal{O}(ed^3)$ by considering only the constraints which could be affected by a removal of values. A lower bound for achieving arc consistency is $\Omega(ed^2)$ since this is the time required to check arc consistency for all constraints. Mohr and Henderson [1986] describe an optimal algorithm AC-4 based on the notion of *support*; a domain value is supported if there exists a compatible value in the domain of every adjacent variable. When a value a is removed from the domain of x, it is not always necessary to examine all binary constraints C_{xy}. We can ignore those values in D_y which do not rely on a for support. The space complexity of AC-4 is $\mathcal{O}(ed^2)$. This is improved to $\mathcal{O}(ed)$ in AC-6 by Bessière [1994].

Path Consistency

Definition 3.16. Let $\mathcal{R} = (X, D, C)$ be a node consistent constraint network. A path $(x_0, \ldots, x_m)$ of length m in the constraint graph for $\mathcal{R}$ is *path consistent* if and only if for any consistent instantiation $(v_0, v_m) \in C_{x_0,x_m}$ there is a sequence of values of $v_j \in D_j, 1 \leq j < m$ such that $(v_0, v_1) \in C_{x_0,x_1}, \ldots, (v_{m-1}, v_m) \in C_{x_{m-1},x_m}$. A constraint network is path consistent if, and only if, every path in its constraint graph is path consistent. □

The following theorem provides a method for achieving path consistency.

Theorem 3.3 ([Montanari, 1974]). *A constraint network is path consistent if, and only if, every path of length* 2 *of a complete constraint graph is path consistent.*

Achieving path consistency thus involves tightening of constraints or, if no constraints exists between two variables, introducing new binary constraints. In analog to REVISE, which deals with two variables for achieving arc consistency, we define REVISE-3 shown in Algorithm 3.5 that takes a path (i, j, k) of length 2 and makes it path consistent by modifying the constraint C_{ik} (which may be the universal constraint if the original network does not contain a constraint between i and k) by deleting tuples that cannot be extended consistently by including values from j.

```
REVISE-3(i, j, k, X, D, C)
1  for each (v_i, v_k) ∈ C_ik
2    for each v_j ∈ D_j
3      if (v_i, v_j) ∉ C_ij or (v_j, v_k) ∉ C_jk
4        C_ik ← C_ik − {(v_i, v_k)}
```

Algorithm 3.5: Make (i, j, k) path consistent. Removes tuples from constraint C_{ik} which cannot be extended consistently to values of j.

We can again use relational algebra to describe the result of REVISE-3:

$$C_{ik} \leftarrow C_{ik} \cap \pi_{ik}(C_{ij} \bowtie D_j \bowtie C_{jk}). \tag{3.11}$$

From Theorem 3.3 it follows that path consistency can be achieved by calling REVISE-3 with all possible paths of length 2. Such a naive path consistency algorithm called PC-1 is shown in Algorithm 3.6 on the next page. Note that achieving path consistency, in general, changes the structure of the constraint graph of a network by introducing new constraints.

The time complexity of REVISE-3 is $\mathcal{O}(d^3)$ since there are at most d^2 tuples in each constraint and at most d values in each domain. We assume that both a constraint check and a set modification takes constant time. There are n variables so there can be at most n^2 binary constraints and each constraint can have at most d^2 tuples. In the worst case, the call to REVISE-3

removes only a single tuple so we get d^2n^2 iterations of the lines 3–6 in PC-1. There are n^3 different paths of length 2 so the time complexity of PC-1 is $\mathcal{O}(n^5d^5)$.

The naive path consistency algorithm can be improved in ways similar to AC-1. Mackworth [1977b] presented PC-2 with time complexity of $\mathcal{O}(n^3d^5)$, and Han and Lee [1988] presented an algorithm called PC-4 with time complexity $\mathcal{O}(n^3d^3)$ and space complexity $\mathcal{O}(n^3d^3)$. Singh [1996] presents PC-5 which improves the space complexity to $\mathcal{O}(n^3d^2)$.

```
PC-1(X, D, C)
1  repeat
2    for each x_k ∈ X
3      for each x_i ∈ X
4        for each x_j ∈ X
5          REVISE-3(x_i, x_j, x_k, X, D, C)
6  until no constraint is changed
```

Algorithm 3.6: A naive algorithm for achieving path consistency.

3.3.2 A Note on Complexity

In the preceding complexity analysis, we made some assumptions that certain constant time operations are available. This was done primarily to get complexity results matching those found in the literature. It may not always be possible, or it requires some preprocessing and/or additional storage, to make these operations run in constant time. The following section will provide some additional details.

Constraint Check

We assumed that a constraint check of the form $(a, b) \in C_{xy}$ takes constant time. For binary networks the maximum constraint arity is a constant, namely 2, but for general networks, a constraint check must use at least time $\Omega(r)$.

If we assume a binary network with constraint represented extensionally as relations, a constraint check becomes equivalent to the dictionary problem: Given a universe U of size d^2 and a subset $S \subseteq U$ of size t, is $x \in U$ a member of S?

If $d \leq w/2$, we can use the results from [Hagerup et al., 2001] where a static dictionary with constant lookup time and size $\mathcal{O}(t)$ can be computed in $\mathcal{O}(t \log_2 t)$ worst-case time.

Otherwise, we can resort to standard hashing techniques [Cormen et al., 2001] where a dictionary, which enables lookup time that takes constant

time on average and has size $\mathcal{O}(t)$, can be computed in $\mathcal{O}(t)$ worst-case time.

Set Modification

All the sets that are modified are subsets of some finite universe, which is known before the algorithms are executed. By representing the sets as bit vectors, we can achieve constant time set update at the expense of requiring time linear in the size of the universe for initialization and additional storage, also linear in the size of the universe, for the bit vectors.

3.4 Tractable Problems

As Theorem 3.2 on page 26 shows, constraint satisfaction problems are generally hard. There are, however, classes of constraint satisfaction problems that are tractable. A problem is considered tractable if it can be solved in time polynomial in the size of the problem representation. For constraint satisfaction problems we deal mainly with backtracking algorithms, so a problem is considered tractable if it can be solved without backtracking.

Identification of tractable problem classes is based mainly on two properties:

- Tractability due to restrictions in the type of constraints allowed.
- Tractability due to the structure of the constraint graph.

For solving arbitrary CSPs limiting the type of constraints allowed (i.e., by allowing only linear constraints) is usually not an option since this reduces the ease of which problems can be modeled, thereby loosing one of the key strengths of CSPs. I will therefore not focus more on tractability due to properties of the constraint relations but refer interested readers to the survey by Pearson and Jeavons [1997]. For the same reason it is generally not an option to limit the structure of the constraint graph since this will limit the kind of problems that can be modeled. But as we will see, the structure of the constraint graph can be modified in various ways to yield a problem that is tractable.

First, some definitions that allow us to identify the tractable problems.

Definition 3.17 ([Freuder, 1978]). Let $\mathcal{R} = (X, D, C)$ be a constraint network. The network $\mathcal{R}$ is 1-*consistent* if, and only if, $D_x = C_x \neq \emptyset$ for all $x \in X$. Let $2 \leq k \leq n$ be an integer. The network is *k-consistent* if, and only if, for any consistent instantiation of $k-1$ variables, we can find a value for an arbitrary kth variable such that we have a consistent instantiation of k variables. □

Definition 3.18 ([Freuder, 1982]). A network is *strongly k-consistent* if it is j-consistent for all $j \leq k$. A network with n variables which is strongly n-consistent is called *globally consistent*. □

Note that for binary networks, node, arc and path consistency is equivalent to strong 1-, 2- and 3-consistency respectively.

There are algorithms that can be used to make a network k-consistent for any k. Unfortunately, while Cooper [1989] describes an optimal algorithm for achieving k-consistency, the running time is exponential in k. Achieving global consistency is therefore not very useful, but low-order (i.e., node, arc and path) consistency algorithms are often used as a preprocessing step to create an equivalent network which may be simpler to solve using a search algorithm.

Definition 3.19 ([Freuder, 1982]). Given a graph (V, E) and a total ordering $\prec$ on V, the *width* of vertex v is the size of the set

$$\{w \mid w \prec v, \{v, w\} \in E\}.$$

The *width of a graph under the ordering* $\prec$ is the maximum width of all vertices. The *width of the graph* (V, E) is the minimum width over all possible orderings of V. □

Example 3.3. An example of an undirected graph with three vertices (and thus six orderings) is shown in Figure 3.4. The graph has width one. □

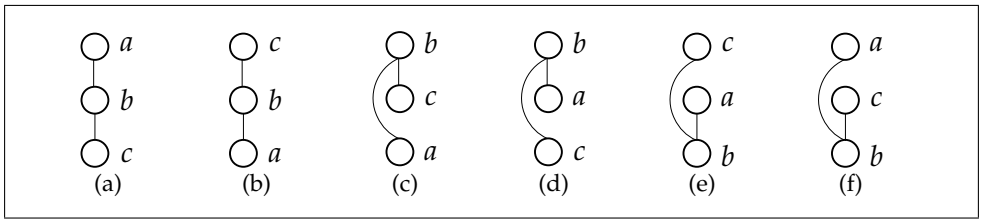


Figure 3.4: A graph ordering example. Nodes are ordered from top to bottom. Orderings (e) and (f) have width 2, the remaining orderings have width 1.

The following theorem establishes the relationship between the connective structure (width) and contextual structure (consistency).

Theorem 3.4 ([Freuder, 1982]). *Let $\mathcal{R}$ be a binary constraint network, let (V, E) be the associated constraint graph and let $\prec$ be a total ordering of V with width w. If $\mathcal{R}$ is strongly $(w+1)$-consistent, then a solution to $\mathcal{R}$ can be obtained by performing a backtrack-free search using the variable ordering $\prec$.*

It is easy to see that a graph has width 1 if, and only if, it is a tree. According to Theorem 3.4, a solution to a binary arc-consistent constraint

network can therefore be found without backtracking. A width-1 ordering of a tree can be constructed by a breadth first search from any root node. Consider a width-1 ordering of the variables $d = \{x_1, \ldots, x_n\}$ and assume a consistent instantiation $\bar{a}$ of variables $\{x_1, \ldots, x_i\}$ has been found. We now need to instantiate x_{i+1}. Since d has width 1, x_{i+1} has only one parent variable $x_j, 1 \leq j \leq i$ which can constrain x_{i+1}. Since x_j is arc consistent relative to x_{i+1}, there exists a value $b \in D_{x_{i+1}}$ such that $\bar{a} \cup \langle x_{i+1}, b \rangle$ is consistent.

It is tempting to believe, that finding the width w of the constraint graph and making the network strongly $(w+1)$-consistent is sufficient to ensure that a solution can be found without backtracking. However, for $w > 2$ enforcing w-consistency generally means that new constraints are added (as was the case in path-consistency), thereby changing the width.

There is some debate as to how much consistency is cost effective. Generally speaking, any search algorithm will benefit from a network having a high level of consistency. But achieving a high level of consistency comes at the expense of additional computation, so there is a tradeoff between the effort spent on preprocessing and the effort spent on search. It was thought not to be cost effective to apply consistency algorithms as part of a hybrid search algorithm [Kumar, 1992] however, especially for difficult problems, it has become clear that this is not the case [Sabin and Freuder, 1994].

CHAPTER 4

Interactive Constraint Satisfaction

The CSP definitions and algorithms presented in the previous chapter are for batch processing where the machine is intended to solve the problem autonomously. Many real world applications, however, require interactive decision support rather than automatic problem solving. This is the case where the initial constraint network admits many possible solutions and human guidance is needed to select a solution based on some additional criteria. These criteria cannot be modeled as constraints in the original network since they are not yet known — the user can only identify these criteria when consequences of the initial constraints are revealed.

In this chapter I first define a list of usability requirements for an interactive constraint satisfaction system, and proceed with an extension of the classical CSP framework (presented in Section 3.1) which can be used to describe interactive constraint satisfaction problems. From these requirements and definitions, I identify operations that are fundamental for an interactive constraint satisfaction system and describe them formally. Efficient algorithms for the fundamental operations are then presented. To make the algorithms efficient, they do not operate on a general constraint network, but on a restricted network called an acyclic network. In the next chapter, we will see how a general constraint network can be transformed into an acyclic network by a compilation procedure.

4.1 Usability Requirements

The setup is this: We are given an initial constraint network that models the problem at hand and this initial constraint network contains some degrees

of freedom, i.e., it allows more than one solution. This initial constraint network cannot be changed by the user but guided by the user's requirements, the number of solutions should be reduced until the number of solutions remaining is manageable or a single solution is found. Note that in the following, the term "user" denotes the person that is using the initial constraint network to find a set of solutions that match the user's criteria.

Example 4.1. The n-queen problem can be used as a simple example. The initial constraint network models the problem as described in Section 1.1.2 on page 3. This network cannot be changed by the user, as it would no longer model the n-queen problem. However, the initial constraint network does have some degrees of freedom that allow the user to influence the solution to be found. As an example, for $n = 4$, the queen in the first column can be placed in either row 2 or 3 as illustrated in Figure 3.1 on page 21. □

The number of solutions to a constraint satisfaction problem can be reduced by repeatedly adding constraints to an initial constraint network (since the addition of a constraint never extends the set of solutions to the network). During this process, some users will likely make choices that are wrong, either due to outright errors but more likely because they gain new insights as they proceed. These wrong choices result either in a network with no solutions or with a set of solutions which does not fulfill the criteria established by the user. An interactive constraint satisfaction system must thus allow for the retraction of already stated constraints.

An initial approach to interactive constraint satisfaction may be to extend a search algorithm with a method for letting the user decide which variable to instantiate next and what value to use in the instantiation. This, however, is of little value to the user since the machine is merely used to recording the user's choice and checking if it is consistent with the initial network and previous choices. If backtracking is needed because a solution does not exists with the current choices, the user has to change one or more of the choices made earlier.

A more reasonable approach would be to let the user specify a constraint and reveal the consequences of adding this constraint by updating the domains to include only values that, based on the initial network and user constraints, can be extended to a full solution.

A number of usability requirements for interactive constraint satisfaction systems can be listed. Frayman [2001] describes many of the requirements that one can have to such a system. Many of these requirements pertain the design of the user interface (i.e., easy navigation, intuitive graphical layout etc.) or the system architecture (i.e., client vs. server processing, security etc.). We will not focus on these requirements here. Instead I list the requirements that need support from the underlying algorithms and data structures.

1. When the decision support task is complete, the resulting instantiation must be consistent. This requirement is the motivation for implementing a constraint-based decision-support system.

2. The user should be able to make a "deselection" stating "this variable should not have these values". Often the specific value of a variable is not interesting, but it is important that it is not assigned a few specific values.

3. If the user wishes to make a selection, which is inconsistent with previous selections, the system should provide the user with a list of previous selections that need to be retracted in order to make the new selection consistent. In this way, the user is not locked by previous selections since they can be changed if a specific value of another variable is deemed more important.

4. The user should be able to make a selection and later retract the selection. It is a fundamental aspect of a good user interface that the user is able to undo actions since this allows experimenting with actions without any risk.

5. The user must be able to make selections in arbitrary order. Again this is a fundamental aspect of a good user interface that the sequence in which things are selected is not fixed. Different users may prefer to do things in different orders.

6. The user should not be able to make selections that lead to a dead-end, i.e., a situation where a solution cannot be found because of the selections made earlier in the process. A dead-end in the 4-queen problem is depicted in Figure 3.1(c) on page 21. Assume the user has manually selected the position of the three first queens. These selections are all mutually consistent, however, it is not possible to make a consistent selection for the last queen, and thereby completing the decision task, without changing some of the previous selections.

7. The response time for all operations should be short. An operation should take less than 1 second for the user's flow of thought to stay uninterrupted [Nielsen, 1994, Ch. 5]. The limit for keeping the user's attention focused on the dialogue is about 10 seconds.

The requirement to the response time should of course be seen in perspective to the size of the problem one is attempting to solve. If the user is trying to solve a problem that would have required weeks of manual work, it may be acceptable to wait a few minutes for the operations to be completed.

4.2 Extension of the Classical CSP Framework

The preceding concepts are formalized in the following definitions where I describe an extension of the classical CSP framework, which can be used to describe interactive constraint satisfaction problems. The notation is inspired by [Amilhastre et al., 2002], and separates a set of dynamic constraints from the initial constraint network.

Definition 4.1 (Dynamic Constraint Network). A *dynamic constraint network* (DCN) is a 4-tuple $\Delta = (X, D, C, H)$ where (X, D, C) is a satisfiable constraint network with a static set of variables X, domains D and a static set of constraints C. The set H is a dynamic set of constraints on the variables X. □

As with the constraints in C, we assume that constraints in H have unique scopes.

Definition 4.2. An instantiation $\bar{a}$ is a solution to a DCN $\Delta = (X, D, C, H)$ if, and only if, $\bar{a}$ is a solution to the constraint network $(X, D, C \cup H)$. The set of all solutions to Δ is denoted $Sol(\Delta)$. The DCN Δ is satisfiable (respectively unsatisfiable) if, and only if, the constraint network $(X, D, C \cup H)$ is satisfiable (respectively unsatisfiable). □

Definition 4.3. Let $\Delta = (X, D, C, H)$ be a DCN, $S \subseteq X$ a set of variables and C_S a constraint on S such that $(X, D, C \cup \{C_S\})$ is a satisfiable constraint network. A *restoration* of C_S on Δ is a subset $E \subseteq H$ such that $(X, D, C \cup E \cup \{C_S\})$ is a satisfiable constraint network. A restoration E of C_S on Δ is maximal if, and only if, there is no restoration E' of C_S on Δ such that $E \subset E'$. □

When combining sets of constraints in Definitions 4.2 and 4.3 we take this to mean that for any two constraints with the same scope, only a single constraint, which is the intersection between the two constraints, is present in the combined set. A constraint in H is called a user constraint, user selection, or just selection.

For the sake of generality, these definitions do not restrict the type of dynamic constraints in H. I will assume, however, that these constraints are always unary. As we will see, this restriction enables efficient algorithms for the fundamental operations while being sufficient to fulfill the requirements stated in Section 4.1 on page 37. Assigning a specific value to a variable can be expressed by adding a unary constraint, which allows only that specific value, on the variable.

The goal of interactive constraint satisfaction is, given a dynamic constraint network $\Delta = (X, D, C, H)$ where H is initially empty, to repeatedly add constraints to H until $|Sol(\Delta)| \leq \gamma$ for some constant γ which is the number of possible solutions the user can cope with. In many applications $\gamma = 1$ meaning that a single unique solution must be found.

4.2.1 Implications of Usability Requirements

By using the preceding definitions, we can describe the implications of the usability requirements (described in Section 4.1), on the fundamental operations required in an interactive constraint satisfaction system. We will use a DCN $\Delta = (X, D, C, H)$ to capture the initial constraint network (as specified by X, D and C) as well as the user selections (stored in H).

- Requirement 1 implies that we only return consistent instantiations, i.e., solutions to Δ.
- According to Requirement 2, the values that should not be selected for a variable x are in a set $N_x \subset D_x$. We can then fulfill the requirement by adding the constraint $C_x = D_x - N_x$ to H, which allows all domain values except the values that are not wanted.
- When the user wishes to add an inconsistent constraint C_x to H, Requirement 3 implies that a restoration E of C_x on Δ is computed. The selections that need to be retracted are thus $H - E$. Note that E is not required to be maximal.
- Selection and retraction in Requirement 4 can be achieved by adding/removing a constraint to/from H.
- Requirement 5 is implicit in our choice of representing selections as a set of constraints. It does imply, however, that we cannot rely on any fixed variable ordering in our algorithms.
- Requirement 6 implies that global consistency (as defined in Definition 3.18 on page 34) should be maintained at all times. Having a globally consistent network means that whenever we have instantiated $k - 1$ variables, we can always find a consistent value for any of the remaining variables and this holds for all $k < n$.

Restoration Computation

As noted, the restoration computed as a result of adding an inconsistent constraint need not be maximal. Indeed, the trivial restoration $\emptyset$ will suffice, but in this case all user selections are retracted which is probably not the best solution. By changing how the restoration is computed, a number of different behaviors can be achieved.

- From a usability perspective we can argue that the user probably makes selections in decreasing order of importance. By storing the dynamic constraints in H as an ordered list, we can thus compute the restoration by first removing all constraints from H and then add as many constraints as possible, starting from the beginning of the list.

- By defining a valuation function $\theta : H \rightarrow \mathbb{N}$, a degree of importance can be attached to each dynamic constraint. If we let $\theta(E) = \sum_{C_x \in H-E} \theta(C_x)$, the restoration E computed should be maximal in the sense that there is no restoration E' satisfying $\theta(E') < \theta(E)$.

Note that the behavior of the first method can be achieved by using the second method and defining θ appropriately. It is likely, however, that the complexity of the two methods is different

4.3 Fundamental Operations

We are now in a position where we can describe the fundamental operations needed in an interactive constraint satisfaction system. In the following, $\tilde{D}$ is a function constructed from the function D such that

$$\tilde{D}(x) \subseteq D(x), \text{ for all } x \in X, \tag{4.1}$$

and $\tilde{D} = \emptyset$ is used to denote the function $\tilde{D}$ where

$$\tilde{D}(x) = \emptyset, \text{ for all } x \in X. \tag{4.2}$$

The basic idea is as follows. The initial constraint network is stored in (X, D, C). Modifications to the domains, due to addition/removal of constraints to/from H, are recorded in $\tilde{D}$. By comparing values of $\tilde{D}$ before and after one of the operations, it is possible to compute the changes caused by the addition/removal of a constraint.

ADD-CONSTRAINT$(\Delta, C_x) \rightarrow (\tilde{D}, \tilde{H})$

Input: $\Delta = (X, D, C, H)$ is a DCN and C_x a unary constraint.
Output: A function $\tilde{D}$, and a set of constraints $\tilde{H}$.
Precondition: There is no $C_y \in H$ such that $y = x$.
Postcondition: If $(X, D, C, H \cup \{C_x\})$ is unsatisfiable then $\tilde{H} = H$ and $\tilde{D} = \emptyset$. Otherwise, $\tilde{H} = H \cup \{C_x\}$, and $\tilde{D}$ is a function where

$$\tilde{D}(y) = \pi_y(Sol(X, D, C, \tilde{H})), \text{ for all } y \in X. \tag{4.3}$$

REMOVE-CONSTRAINT$(\Delta, x) \rightarrow (\tilde{D}, \tilde{H})$

Input: $\Delta = (X, D, C, H)$ is a DCN and $x \in X$ a variable.
Output: A function $\tilde{D}$, and a set of constraints $\tilde{H}$.
Precondition: There exists $C_y \in H$ such that $y = x$.
Postcondition: $\tilde{H} = H - \{C_x\}$. $\tilde{D}$ is a function where

$$\tilde{D}(y) = \pi_y(Sol(X, D, C, \tilde{H})), \text{ for all } y \in X. \tag{4.4}$$

RESTORATION$(\Delta, C_x) \rightarrow E$

Input: $\Delta = (X, D, C, H)$ is a DCN and C_x a unary constraint.
Output: E a restoration of C_x on Δ or $\emptyset$.
Precondition: $(X, D, C, H \cup \{C_x\})$ is unsatisfiable.
Postcondition: If $(X, D, C, \{C_x\})$ is satisfiable, E is a restoration of C_x on Δ, otherwise E is $\emptyset$.

Equations (4.3) and (4.4) imply that upon return from ADD-CONSTRAINT and REMOVE-CONSTRAINT, the domain values in $\tilde{D}$ can be used to extend the current instantiation to a full solution without having to change any previous selections.

By using these fundamental operations it is possible to create an interactive constraint solver. The skeleton of such a solver is shown in Algorithm 4.1 on the following page. Lines 12–18 deal with the selection of an inconsistent value by computing the restoration, removing the constraints that cause the inconsistency and finally applying the new selection.

If we assume that line 4 does not involve any complicated rendering of the variables the response time of the UI is clearly dominated by the running times of ADD-CONSTRAINT, REMOVE-CONSTRAINT, and RESTORATION. Of these three algorithms, ADD-CONSTRAINT is probably the most often used, since REMOVE-CONSTRAINT is only called when the user wishes to remove a previous selection and RESTORATION is called when the user makes a selection that is inconsistent with the previous selections.

4.4 Application to Constraint-Based Configuration

We have already seen configuration problems in Section 1.2 on page 6. Here I will briefly describe how an interactive constraint satisfaction system can be used as a basis for a configuration system.

As noted, a configuration model is used to describe the components that are available as well as the relations between them. It is therefore natural to define the configuration model in terms of a constraint network:

Definition 4.4 (Configuration Model). A *configuration model* is a constraint network $\mathcal{R} = (X, D, C)$. The variables in X list the components available for selection, the domains in D contain the possible choices for each component and the constraints in C contain the constraints between the components. □

The goal of the configuration task is to solve a configuration problem described by a configuration model. Starting with a product model with many possible solutions, the user gradually reduces the number of solutions by choosing consistent values from the variable domains. The configuration system should assist the user by resolving which domain values lead to consistent assignments.

```
SOLVER(X, D, C)
 1  (D̃, H) ← (D, ∅)
 2  Δ ← (X, D, C, H)
 3  repeat
 4     Print available variables and valid domain values from D̃
 5     Read user input
 6     if input = "quit"
 7        return
 8     else if input = select value a for variable x
 9        C_x ← {a}
10        (D̃, H) ← ADD-CONSTRAINT(Δ, C_x)
11        if D̃ = ∅
12           E ← RESTORATION(Δ, C_x)
13           if E = ∅
14              error "Invalid selection"
15           else if User accepts retraction of the constraints H − E
16              for each C_y ∈ H − E
17                 REMOVE-CONSTRAINT(Δ, y)
18              (D̃, H) ← ADD-CONSTRAINT(Δ, C_x)
19     else if input = remove selection on variable x
20        (D̃, H) ← REMOVE-CONSTRAINT(Δ, x)
21     else
22        error "unknown input"
23  until |Sol(X, D̃, C, H)| < γ
```

Algorithm 4.1: A skeleton interactive constraint solver using the fundamental operations.

An instance of a configuration problem can be represented by a DCN $\Delta = (X, D, C, H)$. The product model is a constraint network (X, D, C) and the user restrictions are the set of dynamic constraints H. With this definition each solution to (X, D, C) represents a feasible product[1]. At any time, $Sol(\Delta)$ corresponds to the set of feasible products that satisfy the user's requirements. The generic interactive constraint solver shown in Algorithm 4.1 can be used as basis for the runtime services in a configuration system, as illustrated in Figure 1.4 on page 8.

Example 4.2. Let the product model be defined by the constraint network in Example 3.2 on page 23. Initially, when $H = \{\}$, all domain values are possible. Now the user decides she wants a desktop model by adding the constraint $C_{case} = \{\text{desktop}\}$. We update the list of domain values according to Equation (4.3) on page 42, and keep the domain values that

[1] And we can generally assume that (X, D, C) is satisfiable with more than one possible solution. Otherwise we would not need a configuration system.

are still consistent with the user constraints in H. The changed domains are

$$D_{ide} = \{40\text{gb}, 80\text{gb}\}$$
$$D_{scsi} = \{\text{none}\}$$

As can be seen, there are no choices left for the variable *scsi*. The value has been completely determined by the user constraint. Now we add the constraint $C_{model} = \{\text{home}\}$. The updated domain is:

$$D_{cpu} = \{\text{PIII}, \text{AMD}\}$$

Now the user reconsiders, and decides that she really needs a Pentium IV CPU, i.e., the constraint $C_{cpu} = \{\text{PIV}\}$. This is an inconsistent selection since the value PIV is no longer valid for the variable cpu. We therefore calculate the restoration $\{C_{case}\}$. In other words, the user needs to relax the constraints $H - \{C_{case}\} = \{C_{model}\}$ in order to satisfy the constraint C_{cpu}. The new domains are:

$$D_{model} = \{\text{office}\}$$
$$D_{case} = \{\text{desktop}\}$$
$$D_{ide} = \{40\text{gb}, 80\text{gb}\}$$
$$D_{scsi} = \{\text{none}\}$$
$$D_{cpu} = \{\text{PIV}\}$$

The only remaining selection is *ide*. □

In this example it is relatively straightforward to determine which of the previous selections caused the desired selection to become unavailable. But for models containing thousands of variables it is very difficult to manually identify the source of an inconsistent selection.

4.4.1 Model Specification

Creating and specifying a product model is often the most complicated task in implementing a configuration system. Many practical issues arise when trying to model a company's products. Many companies have product families, which contain different products, but share many common components. To ease maintenance it is therefore essential that product models can be created in a modular fashion, with components shared between models. A lot of research in configuration involves constructing model description languages [Soininen et al., 2001; Soininen and Niemelä, 1999; Sabin and Freuder, 1996].

In this thesis, however, we focus on the configuration task and we therefore ignore any details of the modeling task, including how the model is created. We simply assume the product model is represented by a constraint network $\mathcal{R} = (X, D, C)$, and the constraints in C are represented extensionally as relations. In practice, the last assumption is not an issue since all constraints on finite domains, whether they are specified as propositional forms, relations or something completely different, can always be converted into the extensional form used here.

4.4.2 Pros and Cons of Constraint-Based Configuration

Constraint-based methods have many advantages compared to implementing a configuration system using a procedural approach:

- Relations between components are specified in a declarative way, i.e., the modeler states *what* the relations are, not *how* they should be enforced.
- It is usually easy to check if a constraint is correct since there are no side effects and correctness is not influenced by other constraints in the model[2].
- The configuration system can check if the constraints are in contradiction with each other (by ensuring satisfiability of the model). This is useful when changing a model to ensure that the changes are consistent with the existing constraints.
- Powerful facilities for the end user can be provided (i.e., restorations and global consistency). This would be very hard to provide if the configuration system was implemented using a traditional programming language.

Despite all the advantages, there are a number of issues with constraint-based configuration as presented in this section:

- All domains are finite. In real-world applications we may not know in advance how large the domains should be, so in order to model these problems using a finite domain CSP, we have to make the domains larger than the largest value we think we will need. Since the size of the domains influence the computation time needed to solve the problem, this may not be feasible.
- The constraints presented here are "hard" in the sense that a solution must always satisfy all the constraints. In some applications, *soft*

[2]This requires, of course, that the meaning of a single constraint can be easily understood, e.g., by representing it using symbolic logic.

constraints are needed. Soft constraints are constraints that are not required to be satisfied, but are satisfied if possible.

Several researchers have proposed extensions to the classical CSP framework in order to cope with these issues (see, e.g., [Mittal and Falkenhainer, 1990; Miguel and Shen, 1999] and [Schiex, 1992; Dechter, 1996]). Still, the formulation used in this section is applicable to a large number of practical cases and has been used as the foundation of at least one commercial configuration product [Yu and Skovgaard, 1998].

4.5 Efficient Fundamental Operations

In the previous sections we ignored the complexity of the fundamental operations. Both ADD-CONSTRAINT and RESTORATION must solve a CONSTRAINT SATISFIABILITY problem which was proved $\mathcal{NP}$-complete in Theorem 3.2 on page 26. This makes them unsuitable for interactive use when applied to general constraint networks.

We will, however, postpone the treatment of general networks to the next chapter. For now, we assume the initial network has a certain structure which allows efficient implementation of the fundamental operations. In the next chapter, we will see how any general constraint network can be transformed into a network having this structure called an acyclic network.

Definition 4.5 (Acyclic Network). Let $\mathcal{R} = (X, D, C)$ be a constraint network. An *acyclic network* for $\mathcal{R}$ is a constraint network $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ such that $X \subseteq \hat{X}, D \subseteq \hat{D}$ satisfying the following properties:

1. When restricted to the variables in X, the two networks have the same set of solutions, that is
$$Sol(\mathcal{R}) = \pi_X(Sol(\hat{\mathcal{R}}). \tag{4.5}$$
2. The constraint graph of the dual network of $\hat{\mathcal{R}}$ form a tree if redundant constraints are removed. □

This definition allows us to add variables to the network and change the constraints if we ensure the two properties hold. The first property ensures that a solution to $\hat{\mathcal{R}}$ is also a solution to the original network $\mathcal{R}$. The second property ensures that we can process the network efficiently if we use the dual network representation of $\hat{\mathcal{R}}$. The dual network is always binary and a tree has width 1; so according to Theorem 3.4 on page 34, we can solve the problem without backtracking by enforcing arc consistency (and using, e.g, the ordering obtained by performing a breadth-first search on the constraint graph of the dual network).

If we choose to add variables to the problem, they should, of course, not be visible to the end user who is only interested in the variables of the original network.

4.5.1 Fundamental Operations on an Acyclic Network

Having identified the structure which we are working with, we need to describe how the fundamental operations (which operate on the original network) map to the acyclic dual network.

In the following, we assume the initial network is $\mathcal{R} = (X, D, C)$. The variables and domains of the acyclic dual network are $\hat{X}^d$ and $\hat{D}^d$ respectively, both defined as in Definition 3.11 on page 22. Let us first make a few observations.

Lemma 4.1. *Let $\hat{\mathcal{R}}$ be an acyclic network and $\hat{\mathcal{R}}^d = (\hat{X}^d, \hat{D}^d, \hat{C}^d)$ the corresponding dual network. If $\hat{\mathcal{R}}^d$ is arc consistent, then $\hat{\mathcal{R}}$ is satisfiable if, and only if, all dual domains are non-empty. That is,*

$$|Sol(\hat{\mathcal{R}})| > 0 \iff \hat{D}^d_R \neq \emptyset, \text{ for all } R \in \hat{X}^d, \hat{D}^d_R \in \hat{D}^d. \tag{4.6}$$

Proof. First we note that, by definition, $\hat{\mathcal{R}}$ is satisfiable if and only if $\hat{\mathcal{R}}^d$ is satisfiable.

If a dual domain is empty then $\hat{\mathcal{R}}^d$ is clearly unsatisfiable. Assume therefore that all dual domains are non-empty. A solution to $\hat{\mathcal{R}}^d$ can be found as follows. Assign a valid value to an arbitrary dual variable R. Since $\hat{\mathcal{R}}^d$ is arc consistent, we can assign a consistent value for all dual variables adjacent to R. We can repeat this step, yielding a breadth-first search in the constraint graph of $\hat{\mathcal{R}}^d$, until we have assigned valid values for all dual variables. Since the constraint graph of $\hat{\mathcal{R}}^d$ is a tree, we never have to select a value which makes a previous assignment inconsistent. □

Note that the reverse implication holds even if $\hat{\mathcal{R}}^d$ is not arc consistent: If any dual domain becomes empty, $\hat{\mathcal{R}}$ is unsatisfiable.

Lemma 4.2. *Let $\hat{\mathcal{R}}$ be an acyclic network and $\hat{\mathcal{R}}^d = (\hat{X}^d, \hat{D}^d, \hat{C}^d)$ the corresponding dual network. If $\hat{\mathcal{R}}^d$ is arc consistent, then all dual variables $\hat{X}^d$ agree on the values of the original variables in X. That is,*

$$\pi_x(\hat{D}^d_R) = \pi_x(\hat{D}^d_S), \text{ for all } R, S \in \hat{X}^d \text{ and } x \in R \cap S. \tag{4.7}$$

Proof. Let R and S be two dual variables sharing at least one original variable. Furthermore, let R and S by connected by a constraint in the constraint graph of $\hat{\mathcal{R}}^d$. Since $\hat{\mathcal{R}}^d$ is assumed to be arc consistent, for each tuple $t \in \hat{D}^d_R$, we can find a tuple $u \in \hat{D}^d_S$ such that $t_{|R\cap S} = u_{|R\cap S}$ since this is the only constraint between R and S according to Definition 3.11 on page 22. We can similarly, for each tuple in $u \in \hat{D}^d_S$, find a tuple $t \in \hat{D}^d_R$ satisfying the same condition. In other words, we have

$$\pi_{R\cap S}(\hat{D}^d_R) = \pi_{R\cap S}(\hat{D}^d_S) \tag{4.8}$$

in particular R and S agree on every shared original variable $x \in R \cap S$.

By Definition 4.5, all dual variables $\{S_k, \ldots, S_\ell\} \subseteq \hat{X}^d$ sharing an original variable $x \in X$ are connected by a path of constraints. By applying the previous argument to each pair of dual variables we get

$$\pi_x(\hat{D}^d_{S_k}) = \cdots = \pi_x(\hat{D}^d_{S_\ell}), \tag{4.9}$$

which proves the lemma. □

Adding a constraint C_S to an acyclic network $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ will, in general, change the constraint graph of the dual network, in which case it may no longer have width 1. In the special case where all variables in S are covered by constraints, i.e., $S \subseteq \bigcup_{C_R \in \hat{C}} R$, it is possible to add the constraint without changing the constraint graph of the dual network, by removing from the domains in $\hat{\mathcal{R}}^d$ values that do not satisfy C_S. In constraint networks arising from practical problems, we can usually assume that all variables are covered by at least one constraint (since otherwise, why is the variable included in the first place). We will, however, restrict the dynamic constraints to be unary since this is sufficient to express assignment of values to variables and they have the following desirable properties:

- Adding a unary constraint C_x to $\hat{\mathcal{R}}$ implies that we should remove from the domain of all dual variables in $\hat{\mathcal{R}}^d$ the tuples that do not satisfy C_x, i.e., from all dual variables, which contain x, the tuples that contain the values $D_x - C_x$ for the variable x should be removed:

  $$\hat{D}^d_S \leftarrow \hat{D}^d_S \bowtie C_x, \text{ for all } \hat{D}^d_S \text{ such that } x \in S. \tag{4.10}$$

  If x is not covered by a constraint in $\hat{C}$, we can simply update D_x directly:

  $$D_x \leftarrow C_x. \tag{4.11}$$

- Adding a unary constraint to $\hat{\mathcal{R}}$ does not change the structure of the dual constraint graph. This is obvious since we do not introduce any new dual variables or dual constraints, but merely update the domains of the existing dual variables.

After a constraint has been added, the acyclic dual network may not be arc consistent so we need to apply an arc-consistency algorithm in order to restore arc consistency. Lemma 4.2 implies that, after arc consistency has been restored, the dual variables all agree on the value of the original variables. Therefore (4.10) can simply be reduced to

$$\hat{D}^d_S \leftarrow \hat{D}^d_S \bowtie C_x, \text{ for some } \hat{D}^d_S \text{ such that } x \in S. \tag{4.12}$$

The arc consistency algorithm will ensure that the changes will be propagated to the remaining domains. The same reasoning can be used to determine the valid domain of the original variable x based on the valid domains in the dual network:

$$D_x \leftarrow \pi_x(\hat{D}_S^d), \text{ for some } \hat{D}_S^d \text{ such that } x \in S. \tag{4.13}$$

In the preceding discussion we ignored the fact that we also need the ability to remove constraints, which generally implies that we need to reintroduce some valid domain values from the initial acyclic dual network. Therefore, actually removing invalid domain values does not seem like a good idea. Instead we will mark the values that are valid and remove this mark if a value becomes invalid. If we, at some later point, need to reintroduce a domain value due to removal of a user selection, we simply mark the value as valid. We never need to introduce more domain values than are defined in the original network.

4.5.2 Auxiliary Algorithms

The algorithms for the fundamental operations rely on a number of auxiliary algorithms, which will be described next. In what follows, the input network is denoted by $\mathcal{R} = (X, D, C)$ and it is assumed that the acyclic network $\hat{\mathcal{R}}$ and the acyclic dual network $\hat{\mathcal{R}}^d$ are available. Note that for the DCN $\Delta = (X, D, C, H)$, the constraints in H, which we have assumed are unary, are implicitly reflected in the valid domains of $\mathcal{R}$.

For the complexity analysis we use $\hat{n}$, $\hat{e}$, $\hat{d}$, $\hat{t}$, $\hat{n}^d$, $\hat{e}^d$, $\hat{d}^d$, and $\hat{t}^d$ to denote the number of variables, constraints, domain values in the largest domain, and tuples in the largest constraint respectively, for $\hat{\mathcal{R}}$ and $\hat{\mathcal{R}}^d$. In addition, s will be used to denote the largest number of original variables used by a dual constraint, i.e.,

$$s = \max_{\hat{C}_{ST}^d \in \hat{C}^d} |S \cap T|. \tag{4.14}$$

We can use the following observations to simplify the results:

$$n \leq \hat{n}, \tag{4.15}$$

$$d \leq \hat{d} \tag{4.16}$$

$$\hat{d}^d = \hat{t}, \tag{4.17}$$

$$\hat{n}^d = \hat{e} \tag{4.18}$$

$$\hat{e}^d \leq \hat{e}. \tag{4.19}$$

Equations (4.15) and (4.16) follow from Definition 4.5, Equations (4.17) and (4.18) from Definition 3.11, and Equation (4.19) follows from Equation (4.18) and the fact that the constraint graph of the acyclic dual network is a tree.

From a complexity perspective we can assume the following:

- For any variable x the initial domain D_x is a non-empty finite set, so we can view the domain as integers in the range $\{1, \ldots, |D_x|\}$. Each domain value thus requires $\lceil \log_2 |D_x| \rceil$ bits.
- We can augment each domain D_x with an extra data structure M_x, which is a sequence of $|D_x|$ bits, which is used to mark the valid/invalid domain values. Initially $M[1..|D_x|] = 1$. Access and update to/of a bit takes time $\mathcal{O}(1)$. The storage requirement for M_x is $|D_x|$ bits.
- Removing a domain value, i.e. $\tilde{D}_x \leftarrow \tilde{D}_x - \{j\}, j \in D_x$ is implemented as $M[j] = 0$ which takes time $\mathcal{O}(1)$.
- Reintroducing a domain value, i.e. $\tilde{D}_x \leftarrow \tilde{D}_x \cup \{j\}, j \in D_x$ is implemented as $M[j] = 1$ which takes time $\mathcal{O}(1)$.
- The size of a set can be returned in time $\mathcal{O}(1)$ by maintaining counters appropriately.
- The storage requirement for storing a domain D_x is $|D_x| + \lceil \log_2 |D_x| \rceil$ bits. We will ignore any additional space needed to map the integers in D_x back to the original values. The initialization time is $\mathcal{O}(|D_x|)$ to initialize M_x.
- The set of variables is finite and static so we can similarly view the variables as integers in the range $\{1, \ldots, |X|\}$. The function D from variables to domains can thus be viewed as an ordered sequence of $|X|$ sets. Updating the function for variable $x \in X$ is implemented as an update of the set stored at $D[x]$ which takes time $\mathcal{O}(1)$.

Revise-Dual

The algorithm REVISE-DUAL, which is shown in Algorithm 4.2 on the following page, takes an acyclic dual network $\hat{\mathcal{R}}^d$ and a unary constraint C_x and updates a domain in $\hat{\mathcal{R}}^d$ to reflect the unary constraint. The domain is updated according to Equation (4.12) by marking the tuples which do not agree with C_x on the variable x as invalid. The return value is FALSE if the domain becomes empty (which means that $(X, D, C, H \cup \{C_x\})$ is unsatisfiable), otherwise the return value is TRUE. The acyclic dual network is, in general, not arc consistent upon return from REVISE-DUAL.

Lemma 4.3. *Let $\hat{D}^d_S \in \hat{D}^d$ be the domain of a dual variable S such that $x \in S$. Then* REVISE-DUAL

1. *correctly computes $\hat{D}^d_S \leftarrow \hat{D}^d_S \bowtie C_x$,*
2. *returns* FALSE *if $\hat{D}^d_S$ is empty after the computation and* TRUE *otherwise, and*

```
REVISE-DUAL(R̂^d, C_x)
1  D̂^d_S ← some domain in R̂^d such that x ∈ S
2  r ← FALSE
3  for j ← 1 to |D̂^d_S|
4     if M̂^d_S[j] = 1
5        if D̂^d_S[j]_|x ∈ C_x
6           r ← TRUE
7        else
8           M̂^d_S[j] ← 0
9  return r
```

Algorithm 4.2: Revise domain of a dual variable in $\hat{\mathcal{R}}^d$ to reflect unary constraint C_x.

3. *has time complexity* $\mathcal{O}(d + \hat{t})$ *and space complexity* $\mathcal{O}(d)$.

Proof. The first part follows from the fact that all valid values in $\hat{D}^d_S$ are checked against C_x and marked invalid if not present. TRUE is returned if, and only if, there is an index value j such that $\hat{M}^d_S[j] = 1$ and $\hat{D}^d_S[j]_{|x} \in C_x$. This proves the second part. The membership test in line 5 takes time $\mathcal{O}(1)$ if we use $\mathcal{O}(d)$ time to create a lookup table for the allowed values in C_x. The time complexity is thus $\mathcal{O}(d + \hat{d}^{\hat{d}})$, and by using Equation (4.17) we have proven the third part. □

Valid-Domains

The algorithm VALID-DOMAINS, shown in Algorithm 4.3 on the next page, takes a DCN Δ and creates the set of valid domain values from the domain values in the acyclic dual network $\hat{\mathcal{R}}^d$ according to Equation (4.13).

Lemma 4.4. *Let* $\Delta = (X, D, C, H)$ *be a dynamic constraint network and* $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ *the acyclic network corresponding to* (X, D, C). *Furthermore, let* $\hat{\mathcal{R}}^d$ *be the dual acyclic network obtained from* $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ *by applying* REVISE-DUAL *for each unary constraint in* H. *If* $\hat{\mathcal{R}}^d$ *is satisfiable and arc consistent then* VALID-DOMAINS

1. *returns a function* $\tilde{D}$ *such that*

$$\tilde{D}(y) = \pi_y(Sol(\Delta)), \text{ for all } y \in X, \text{ and} \tag{4.20}$$

2. *has time complexity* $\mathcal{O}(n(d + \hat{t}))$ *and space complexity* $\mathcal{O}(d)$.

Proof. We first note that VALID-DOMAINS treats each variable independently so it is sufficient to prove the first part for some variable $y \in X$. Second if y is not covered by any constraint in $\hat{C}$, the first part is trivially true because of lines 14–15. Otherwise, let $\hat{D}^d_S$ be a domain in $\hat{D}^d$ such that

```
VALID-DOMAINS(Δ)
 1  D̃ ← ∅
 2  for each y ∈ X
 3     if there exists some domain D̂^d_S in R̂^d such that y ∈ S
 4        ▷ Compute M_y
 5        M_y[1..|D_y|] = 0
 6        D̃_y ← ∅
 7        for j ← 1 to |D̂^d_S|
 8           if M̂^d_S[j] = 1
 9              v ← D̂^d_S[j]|_y
10              if M_y[v] = 0
11                 M_y[v] ← 1
12                 D̃_y ← D̃_y ∪ {v}
13     else
14        if there exists a C_y ∈ H
15           D̃_y ← D̃_y ∩ C_y
16     D̃[y] ← D̃_y
17  return D̃
```

Algorithm 4.3: Compute valid domains in Δ from $\hat{\mathcal{R}}^d$.

$y \in S$. Lines 5–12 compute $\tilde{D}_y$ from $\hat{D}^d_S$. By Lemma 4.2 the dual variables which contain y all agree on the valid values. It is therefore sufficient to prove that $\pi_y(\hat{D}^d_S) = \pi_y(Sol(\Delta))$.

If $a \in \pi_y(Sol(\Delta))$ then clearly $a \in \pi_y(\hat{D}^d_S)$. Now let $b \in \pi_y(\hat{D}^d_S)$. Then $b \in \pi_y(Sol(\Delta))$, since we can find a solution containing $\langle y, b\rangle$ by performing a breadth-first search in the tree rooted at S. $\hat{\mathcal{R}}^d$ is arc-consistent and satisfiable so we can always find a consistent value for each dual variable encountered during the search. This proves the first part.

Line 5 takes $\mathcal{O}(d)$ to initialize M_y which has size $\mathcal{O}(d)$, all the remaining operations take $\mathcal{O}(1)$ so the time complexity is $\mathcal{O}(n(d + \hat{d}^d))$. Using Equation (4.17) we have proven the second part. □

Arc-Consistency

While the AC-4 arc consistency algorithm has an optimal worst-case time complexity and thus better than AC-3, AC-3 usually has better average case time complexity and is therefore often the algorithm of choice in actual implementations [Wallace, 1993]. Bessière and Régin [2001] described a new arc consistency algorithm called AC-2001, which is a refinement of AC-3 that has the optimal worst-case time complexity. AC-2001 preserves much of the simplicity in AC-3, which makes it much simpler to implement compared to the other optimal algorithms, AC-4 and AC-6. Bessière and Régin

[2001] also presented experimental results that showed improved performance of AC-2001, for both number of constraint checks and CPU time, when compared with AC-3 and AC-4. They note however, that for problem instances which require a lot of constraint propagation[3] the much more complex algorithm AC-6, relying on sophisticated data structures, is more efficient.

The algorithm ARC-CONSISTENCY, which is shown in Algorithm 4.6 on the facing page, tries to enforce arc consistency in $\hat{\mathcal{R}}^d$ by using techniques similar to those in AC-2001 but is adapted here to work with the implicit equality constraints present in the dual network.

The algorithm needs an ordering on the domain values. The domain values in the dual network are tuples in a relation, which provides a natural ordering (cf., Section 2.2.1 on page 15). The basic idea is as follows. For each dual constraint $\hat{C}^d_{S,T}$ the structure $Last[S, i, T]$ contains the index of the last tuple in $\hat{D}^d_T$ which was found as support for the ith tuple in $\hat{D}^d_S$. Likewise, $Last[T, j, S]$ contains the index of the last tuple in $\hat{D}^d_S$ which was found as support for the jth tuple in $\hat{D}^d_T$. Initially the index is 0 indicating that no support has been found yet. Whenever we check an arc in the call REVISE-2001(S, T) we can simply, for each ith tuple in S that is still valid, check if the tuple at $Last[S, i, T]$ is still valid in T (lines 3–5). If it is valid, the ith tuple is still valid and we can continue with the next tuple. If it is not valid, we only need to check the tuples in T that follow $Last[S, i, T]$, since the tuples before have already been checked (lines 6–12). If we find a supporting tuple, we update *Last* and continue. Otherwise we need to mark the ith tuple of S as invalid since it no longer has support in T.

In ARC-CONSISTENCY, $\mathcal{Q}$ is a set of dual variables which have had their domains modified (and we thus need to check, for each dual variable $S \in \mathcal{Q}$, the domains of the dual variables connected to S with a dual constraint, cf., Definition 3.15 on page 29). Initially we check all dual constraints and add to $\mathcal{Q}$ the dual variables with modified domains. PROPAGATION is then called to propagate the changes to the remaining variables.

Lemma 4.5. *Let $\hat{\mathcal{R}}^d$ be an acyclic dual network.*

1. *If $\hat{\mathcal{R}}^d$ is unsatisfiable, the return value of* ARC-CONSISTENCY *is* FALSE.
2. *If $\hat{\mathcal{R}}^d$ is satisfiable, $\hat{\mathcal{R}}^d$ is made arc consistent and the return value of* ARC-CONSISTENCY *is* TRUE.
3. *The time complexity of* ARC-CONSISTENCY *is $\mathcal{O}(s\hat{e}\hat{t}^2)$ and space complexity is $\mathcal{O}(\hat{e}\hat{t})$.*

[3] These are usually randomly generated problem instances that fall in the phase transition of arc consistency. See [Gent et al., 1997] for more detail on the phase transition behaviour of arc consistency.

```
REVISE-2001(S, T)
 1  changed ← FALSE
 2  for i ← 1 to |D̂_S^d|
 3     if M̂_S^d[i] = 1
 4        j ← Last[S, i, T]
 5        if j = 0 ∨ M̂_T^d[j] = 0
 6           k ← j + 1
 7           while k ≤ |D̂_T^d| ∧ Last[S, i, T] = j
 8              if M̂_T^d[k] = 1 ∧ M̂_S^d[i]_{|S∩T} = M̂_T^d[k]_{|S∩T}
 9                 Last[S, i, T] ← k
10              k ← k + 1
11           if Last[S, i, T] = j
12              M̂_S^d[i] ← 0
13              changed ← TRUE
14  return changed
```

Algorithm 4.4: Remove values from D_S^d without support from D_T^d.

```
PROPAGATION(R̂^d, Q)
 1  while Q ≠ ∅
 2     T ← some dual variable from Q
 3     Q ← Q − {T}
 4     for each S ∈ X̂^d such that Ĉ_ST^d ∈ Ĉ^d
 5        if REVISE-2001(S, T)
 6           if D̂_S^d = ∅
 7              return FALSE
 8           Q ← Q ∪ {S}
 9  return TRUE
```

Algorithm 4.5: Propagate changes for dual variables in $\mathcal{Q}$.

```
ARC-CONSISTENCY(R̂^d)
 1  Q ← ∅
 2  Last[x, y, z] ← 0 for all x, y, z
 3  for each S ∈ X̂^d
 4     for each T ∈ X̂^d such that Ĉ_ST^d ∈ Ĉ^d
 5        if REVISE-2001(S, T)
 6           if D̂_S^d = ∅
 7              return FALSE
 8           Q ← Q ∪ {S}
 9  return PROPAGATE(R̂^d, Q)
```

Algorithm 4.6: Enforce arc-consistency in $\hat{\mathcal{R}}^d$.

Proof. We first note that ARC-CONSISTENCY only returns FALSE if a dual domain becomes empty (lines 6–7 of ARC-CONSISTENCY and PROPAGATION). To prove the two first parts, it is, according to Lemma 4.1 on page 48, sufficient to prove that $\hat{\mathcal{R}}^d$ is arc consistent when we reach line 9 of PROPAGATION.

We first claim that REVISE-2001 makes S arc consistent relative to T and returns TRUE if and only if the domain of S has changed. This follows from the fact that all valid domain values in $\hat{D}_S^d$ are checked (lines $2-3$). If the support for a valid domain value is still valid in $\hat{D}_T^d$ nothing happens (line 5). If this is the first call to REVISE-2001 for S and T or if the support has been marked invalid, a new support value is searched in the valid values of $\hat{D}_T^d$ (lines 6–10). If support is found (line 8) we store the index of the support value and nothing is changed in $\hat{D}_S^d$ for this particular value. If no support is found (line 11) we mark the domain value in $\hat{D}_S^d$ as invalid and change the return value to TRUE.

For ARC-CONSISTENCY we note that initially all arcs are checked twice (one check in each direction). During the loop (lines 3–8) a variable S is added to $\mathcal{Q}$ whenever its domain has changed (line 8). In a subsequent call to PROPAGATION, all variables which rely on S for support are then checked (lines 4–8). These variables are the only variables that can change due to changes in the domain of S. It follows from this that $\mathcal{Q}$ becomes empty and the loop terminates when no further domain values can be removed by the call to REVISE-2001, i.e., arc consistency has been achieved.

For the complexity analysis we first note that a variable is placed in $\mathcal{Q}$ whenever its domain has changed. In the worst case, only a single domain value is removed in the call to REVISE-2001. For any constraint $\hat{C}_{ST}^d$ REVISE-2001(S,T) is therefore called at most $\hat{d}^d$ times. For any tuple in $\hat{D}_S^d$, line 4 of REVISE-2001 is thus executed at most $\hat{d}^d$ times. The total time spent in lines 5–11 for a single tuple during the $\hat{d}^d$ calls is $\mathcal{O}(s\hat{d}^d)$ since we perform the constraint check in line 9 at most once for each tuple in $\hat{D}_T^d$, the number of tuples is bounded by $\hat{d}^d$ and a constraint check takes time $\mathcal{O}(s)$. Since the number of tuples in $\hat{D}_S^d$ is bounded by $\hat{d}^d$ and there are $\hat{e}^d$ constraints, the worst-case time complexity becomes $\mathcal{O}(s\hat{e}^d(\hat{d}^d)^2)$. The size of the set Q is bounded by $\hat{n}^d$ and we need $\mathcal{O}(\hat{d}^d)$ space to store the *Last* array for each dual constraint so the space complexity becomes $\mathcal{O}(\hat{e}^d\hat{d}^d)$. By using Equations (4.17) and (4.19) we have proven the third part. □

4.5.3 Algorithms for the Fundamental Operations

The following sections present algorithms for the fundamental operations.

Add-Constraint

The fundamental algorithm ADD-CONSTRAINT, which is shown in Algorithm 4.7, takes a DCN Δ and a unary constraint C_x and adds the unary constraint as described in Section 4.3 on page 42.

Theorem 4.6. *Let $\Delta = (X, D, C, H)$ be a DCN and C_x and unary constraint on some variable $x \in X$. Then* ADD-CONSTRAINT

1. *correctly computes the sets $\tilde{H}$ and $\tilde{D}$ as described in Section 4.3, and*
2. *has a worst-case time complexity of $\mathcal{O}(n(d + \hat{t}) + s\hat{e}\hat{t}^2)$ and space complexity $\mathcal{O}(\hat{e}\hat{t})$.*

Proof. Both parts of the theorem follow from Lemma 4.3, Lemma 4.4 and Lemma 4.5. □

```
ADD-CONSTRAINT(Δ, C_x)
1  if REVISE-DUAL(R̂^d, C_x) ∧ ARC-CONSISTENCY(R̂^d)
2     D̃ ← VALID-DOMAINS(Δ)
3     return (D̃, H ∪ {C_x})
4  else
5     return (∅, H)
```

Algorithm 4.7: Adding a unary constraint.

Remove-Constraint

The fundamental algorithm REMOVE-CONSTRAINT, which is shown in Algorithm 4.8 on the following page, takes a DCN Δ and a variable x and removes the constraint on C_x as described in Section 4.3 on page 42. It naively removes all constraints in H and then adds the constraints $H - \{C_x\}$. Lines 2–4 remove the constraints in H by marking all domain values in $\hat{\mathcal{R}}^d$ as being valid. Lines 6–10 add the constraints from $H - \{C_x\}$.

Theorem 4.7. *Let $\Delta = (X, D, C, H)$ be a DCN and x some variable in X such that there exists a constraint $C_x \in H$. Then* REMOVE-CONSTRAINT

1. *correctly computes the sets $\tilde{H}$ and $\tilde{D}$ as described in Section 4.3, and*
2. *has a worst case time complexity of $\mathcal{O}(n(d + \hat{t}) + s\hat{e}\hat{t}^2)$ and space complexity $\mathcal{O}(\hat{e}\hat{t})$.*

Proof. The fact that all constraints in H are removed in lines 2–7 combined with Lemma 4.3, Lemma 4.4 and Lemma 4.5 proves both part of the theorem. □

```
REMOVE-CONSTRAINT(Δ, x)
 1  ▷ Remove all constraints in H
 2  for each S ∈ X̂^d
 3     for j ← 1 to |D̂^d_S|
 4        M̂^d_S[j] ← 1
 5  for each x ∈ X̂
 6     for j ← 1 to |D̂_x|
 7        M̂_y[j] ← 1
 8  ▷ Add all constraints from H − C_x
 9  for each C_y ∈ H
10     if y = x
11        H ← H − {C_y}
12     else
13        REVISE-DUAL(R̂^d, C_y)
14  ARC-CONSISTENCY(R̂^d)
15  D̃ ← VALID-DOMAINS(Δ)
16  return (D̃, H)
```

Algorithm 4.8: Removing a unary constraint.

Restoration

The fundamental algorithm RESTORATION, shown in Algorithm 4.9 on the facing page, takes a DCN Δ and a constraint C_x and computes a restoration of C_x on Δ as described in Section 4.3 on page 42. It works by naively removing all constraints in H and thereafter adding C_x. If the resulting network is satisfiable, the constraints from H, which do not make the network unsatisfiable, are added. Lines 2–4 remove the constraint in H by marking all domain values in $\hat{\mathcal{R}}^d$ as being valid. Line 6 adds the constraint C_x and lines 10–15 add the constraints from H which do not make the network unsatisfiable.

Theorem 4.8. *Let $\Delta = (X, D, C, H)$ be a DCN and C_x a unary constraint on some variable $x \in X$. Then* RESTORATION

1. *correctly computes a restoration as described in Section 4.3, and*
2. *has worst-case time complexity $\mathcal{O}(|H|(s\hat{e}\hat{t}^2 + n(d + \hat{t})))$ and space complexity $\mathcal{O}(\hat{e}\hat{t})$.*

Proof. It follows from lines 2–7 that, if $(X, D, C \cup \{C_x\})$ is unsatisfiable, $\emptyset$ is returned. Otherwise E, which is initially empty, is a restoration of C_x on Δ since we only add a constraint $C_y \in H$ to E, if $(X, D, C, E \cup \{C_x\} \cup \{C_y\})$ is satisfiable (lines 11-15). Both parts of the theorem follow from Lemma 4.3, This proves the first part.

The last part follows from Lemma 4.4 and Lemma 4.5. □

```
RESTORATION(Δ, C_x)
 1  ▷ Remove all constraints in H
 2  for each S ∈ X̂^d
 3     for j ← 1 to |D̂^d_S|
 4        M̂^d_S[j] ← 1
 5  ▷ Add C_x
 6  if ¬REVISE-DUAL-(R̂^d, C_x) ∨ ¬ARC-CONSISTENCY(R̂^d)
 7     return ∅
 8  else
 9     ▷ Add consistent constraints from H
10     E ← ∅
11     for each C_y ∈ H
12        if REVISE-DUAL(R̂^d, C_y) ∧ ARC-CONSISTENCY(R̂^d)
13           E ← E ∪ {C_y}
14        else
15           REMOVE-CONSTRAINT(Δ, C_y)
16     return E
```

Algorithm 4.9: Calculate a restoration.

4.5.4 Summary of Results

Table 4.1 summarizes the complexity results for the fundamental operations. As can be seen, all the fundamental operations can be performed in time polynomial in the size of the acyclic constraint network. This means that, given a general constraint network as input, the performance of these operations can be assessed when the input network has been transformed into an acyclic constraint network. Note, however, that the time spent constructing the acyclic constraint network is, in the worst case, unlikely to be polynomial in the size of the input network (since otherwise we could solve CONSTRAINT SATISFIABILITY in polynomial time, which is unlikely unless $\mathcal{P} = \mathcal{NP}$). The best we can hope for is that the networks that arise in practical applications can be efficiently transformed into acyclic networks, which have a tractable size.

Function	Time Complexity	Space Complexity
ADD-CONSTRAINT	$\mathcal{O}(s\hat{e}\hat{t}^2 + n(d + \hat{t}))$	$\mathcal{O}(\hat{e}\hat{t})$
REMOVE-CONSTRAINT	$\mathcal{O}(s\hat{e}\hat{t}^2 + n(d + \hat{t}))$	$\mathcal{O}(\hat{e}\hat{t})$
RESTORATION	$\mathcal{O}(\lvert H\rvert(s\hat{e}\hat{t}^2 + n(d + \hat{t})))$	$\mathcal{O}(\hat{e}\hat{t})$

Table 4.1: Complexity results for the fundamental operations.

CHAPTER 5

Acyclic Network Construction

In the previous chapter, I presented polynomial algorithms for the fundamental operations in an interactive constraint satisfaction system. The algorithms rely on a restricted constraint network called an acyclic constraint network. Requiring that problems must be stated using acyclic networks is too restrictive, so we need a method which can transform a general constraint network into an acyclic constraint network. This transformation can be viewed as a compilation of the general network into a form that enables efficient processing.

Let us first ensure that an acyclic network can be found for any general constraint network.

Theorem 5.1. *For any general constraint network $\mathcal{R} = (X, D, C)$ there exists a corresponding acyclic network.*

Proof. Let $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ be a constraint network such that $\hat{X} = X$, $\hat{D} = D$ and $\hat{C} = \{C_X\}$ where C_X is the constraint

$$C_X = Sol(\mathcal{R}). \tag{5.1}$$

Then $\hat{\mathcal{R}}$ is an acyclic network for $\mathcal{R}$. It is clear that they have the same set of solutions and since the constraint graph of $\hat{\mathcal{R}}^d$ is a single node, it is a tree. □

Theorem 5.1 also gives us a method to construct an acyclic network since, as noted in Section 3.1.2 on page 20, the set of all solutions can be found by joining all the constraints:

$$C_X = \bowtie_{C_S \in C} C_S. \tag{5.2}$$

For networks with a small number of solutions, this method may be acceptable provided that the size of the intermediate results are manageable.

This chapter presents an existing method for synthesizing solutions to a constraint network and an existing method for decomposing a constraint network. We end the chapter by proposing a method for acyclic network construction which combines the two existing methods.

5.1 Array-based Logic

Møller [1995] presented a method for constraint satisfaction based on the theory of *arrays* [More, Jr., 1973]. In what follows, the essential parts of the method, called *array-based logic*, are presented using the terminology already established in the preceding chapters.

Array-based logic works by basically building the set of all solutions by joining all the constraints of the network as described in Equation (5.2) on the page before. As the following example shows, the order in which the joins of Equation (5.2) are evaluated is important

Example 5.1. In Figure 5.1 on the facing page the relations R, S and T are shown. The join operator is commutative so there are three possible ways in which we can compute the join of these three relations:

$$(R \bowtie S) \bowtie T, \tag{5.3}$$
$$(S \bowtie T) \bowtie R, \text{and} \tag{5.4}$$
$$(T \bowtie R) \bowtie S. \tag{5.5}$$

While all three evaluation orders produce the same result, the size of the intermediate result differs as shown in Figure 5.1. The evaluation order according to Equation (5.5) produces a larger intermediate result than the two others, thus it is likely to perform worse. □

Møller [1995] used a simple heuristic to select the next pair of constraints to be joined. For all pairs of constraints $C_R, C_S \in C$ we define the *connectivity factor* as

$$F(C_R, C_S) = |C_R| \times |C_S| \times |R - S| \times |S - R|. \tag{5.6}$$

The connectivity factor should provide an approximation of the size of the joined result. The intuition behind Equation (5.6) is as follows. The larger the two constraints, the larger the joined result. This accounts for the first two factors. The more variables shared by the two constraints, the smaller the result. This accounts for the two last factors. The heuristic simply selects the pair of constraints with the smallest connectivity factor as the next pair to be joined.

a	b
0	0
0	1

(a) Relation R

b	c
0	3
0	4
1	5

(b) Relation S

c	d
4	6
5	7

(c) Relation T

a	b	c
0	0	3
0	0	4
0	1	5

(d) $R \bowtie S$

b	c	d
0	4	6
1	5	7

(e) $S \bowtie T$

a	b	c	d
0	0	4	6
0	1	4	6
0	0	5	7
0	1	5	7

(f) $T \bowtie R$

Figure 5.1: Intermediate results when joining relations.

If we return to Example 5.1, we see that $F(R,S) = 6$, $F(S,T) = 6$, and $F(T,R) = 8$. The join order selected by the heuristic thus, in this case, avoids the largest intermediate result, but does not necessarily select the smallest intermediate result.

As mentioned previously, it is not always feasible to join all the constraints. This was recognized in [Møller, 1995, pp. 138] where a pragmatic solution was presented, that, for a constraint network $\mathcal{R} = (X, D, C)$, can be formulated as follows.

1. If $|C| = 1$ we are done.
2. Select constraints $C_R, C_S \in C$ such that
$$F(C_R, C_S) = \min\{F(C_T, C_U) \mid C_T, C_U \in C\}. \tag{5.7}$$
3. Compute $C_{R\cap S} = C_R \bowtie C_S$.
4. If $|C_{R\cap S}| < k$ for some constant k, then set $C \leftarrow C - \{C_R, C_S\} \cup \{C_{R\cap S}\}$ and continue with step 1.

If we end up with a single constraint, we are done as we have an acyclic constraint network. Otherwise, we have a number of constraints whose dual constraint graph does not in general form a tree. In this case Møller augments each of the remaining constraints C_S with the set of solutions to the subproblem defined on $X - S$. The details of this final step are specific to Møller's inference algorithms and are not applicable to our case, so we will not pursue this idea any further.

5.2 Cartesian Product Representation

Møller [1995] used a compact representation of constraints, based on Cartesian products, which was supposed to make the array-based techniques applicable to real world problems by reducing the storage requirements.

Let the set of valid tuples for a constraint be represented by

$$\{(0,0,0),(0,0,1),(0,1,1),(1,0,1),(1,1,1)\}. \tag{5.8}$$

An alternative representation is to use, when feasible, a Cartesian product to generate the set of valid tuples. Thus the same set can be represented as

$$\{(0,0,0)\} \cup (\{0,1\} \times \{0,1\} \times \{1\}). \tag{5.9}$$

The representation in Equation (5.9) is what we refer to as a *Cartesian product representation* (CPR). We will refer to the explicit enumeration of all tuples in Equation (5.8) as the *normal representation*. In Figure 5.2 a relation is shown both using the normal and the Cartesian product representation. When using the tabular notation, a Cartesian product is implied between the sets on the same row. In addition, we omit the set delimiters $\{\}$ when the set is a singleton.

a	b	c
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

(a) Relation in the normal representation.

a	b	c
0	0	0
$\{0,1\}$	$\{0,1\}$	1

(b) Relation in CPR.

Figure 5.2: A relation stored using the normal and the Cartesian product representation.

The intuition behind CPR is to group domain values together that are *interchangeable*. Two domain values are *interchangeable* in some (local or global) environment if they can be substituted for each other without any effects to the environment. Freuder [1991] introduced the notion of *interchangeability* and defined several levels of interchangeability. Domain values a and b for some variable x are *fully interchangeable* if every solution containing $\langle x,a\rangle$ remains a solution if $\langle x,b\rangle$ is substituted for $\langle x,a\rangle$ and vice versa. Values a and b are said to be *neighborhood interchangeable* if, for every constraint involving x and every tuple that admits $\langle x,a\rangle$, there is an otherwise identical tuple which admits $\langle x,b\rangle$.

In Møller's use of interchangeability, the environment is restricted to a single constraint so domain values a and b are interchangeable for a con-

straint C_S if, for every tuple that admits $\langle x, a \rangle$, there is an otherwise identical tuple which admits $\langle x, b \rangle$.

Møller used CPR as a way to handle the combinatorial explosion when joining constraints, however the use of CPR has also been successfully applied to traditional CSP backtracking algorithms [Hubbe and Freuder, 1992; Silaghi et al., 1999].

5.2.1 Operations on CPR Relations

The values in a tuple are sets when the constraint is in the CPR and scalars when the constraint is in the normal representation. The following definition provides a general way to measure the size of a constraint.

Definition 5.1. The *size* of a constraint C_S with scope S, denoted $\|C_S\|$, is the number of scalar values contained in the constraint,

$$\|C_S\| = \sum_{x \in S} \sum_{i=1}^{|C_S|} |C_S[i]_{|x}|. \tag{5.10}$$

□

It follows from Definition 5.1 that the size of a constraint C_S in the normal representation is equivalent to $|S||C_S|$ as would be expected. For a constraint network $\mathcal{R}$ we will use $\|\mathcal{R}\|$ to denote the total size of all the constraints in $\mathcal{R}$.

The main benefit of the CPR is that it potentially saves spaces while still allowing the usual relational operators such as join, project and select to be applied without having to generate all the tuples of the original constraint. The space savings can be seen in Figure 5.2 on the preceding page where the normal representation has size 15 while the CPR has size 8.

The definition of the join operation must be changed slightly to handle CPR constraints. The tuples of two relations should be combined whenever the intersection between the values of all the common attributes is nonempty. When two tuples are combined, we should only include the intersection of the values from the common attributes. All non-common attribute values are included as before. Figure 5.3 on the following page illustrates the result of joining two CPR relations.

This leads to a new definition of the join operator which will also work with CPR relations. For relations in the normal representation, Definition 5.2 generates the same result as Definition 2.8 on page 14 as would be expected.

Definition 5.2. Let R be a relation with scheme Y and instance r. Let S be a relation with scheme Z and instance s. The *join* of R and S, denoted $R \bowtie S$,

a	b	c
0	$\{0,1\}$	0
$\{1,2\}$	0	1
4	4	1

(a) Relation R.

b	c	d
$\{0,1\}$	1	1
2	1	3
$\{1,3\}$	0	2

(b) Relation S.

a	b	c	d
0	1	0	2
$\{1,2\}$	0	1	1

(c) $R \bowtie S$

Figure 5.3: Joining relations stored using the Cartesian product representation. The two tuples in the result are obtained by combining $R[1]$ with $S[3]$ and $R[2]$ with $S[1]$ respectively.

is defined to be the relation with scheme $Y \cup Z$ and an instance containing the following set of tuples:

$$\{t = u \sqcup v \mid t \text{ is a tuple over } Y \cup Z, u \in r, v \in s, |u \sqcap v| = |Y \cap Z|\}, \quad (5.11)$$

where

$$u \sqcup v = u_{|Y-Z} \cup v_{|Z-Y} \bigcup_{x \in Y \cap X} (u_{|x} \cap v_{|x}), \quad (5.12)$$

$$u \sqcap v = \{x \mid x \in Y \cap X, u_{|x} \cap v_{|x} \neq \emptyset\}. \quad (5.13)$$

□

Equation (5.12) specifies that the new tuple should be combined from all the components of the non-common attributes as well as the intersection of the values of the common attributes. Equation (5.13) specifies a set containing common attributes where the intersection is not empty. It is easily seen that Definition 5.2 on the preceding page is equivalent to Definition 2.8 on page 14 when joining relations in the normal representation: The last part of Equation (5.12) becomes $u_{|Y \cap Z}$ and Equation (5.13) becomes a set containing the common attributes which have the same value in the two tuples.

Madsen [2002] presented an algorithm for in-memory join of relations in the CPR. The worst-case running time for joining two relations R and S with the same scheme is $\mathcal{O}(|R||S|)$.

We will introduce a new relational operator to convert a constraint to the CPR.

Definition 5.3. Let C_S be a constraint. The *Cartesian product representation* of C_S is obtained by applying the κ operator on C_S. The standard representation is obtained by applying the κ^{-1} operator. In other words, we have

$$C_S = \kappa^{-1}(\kappa(C_S)), \quad (5.14)$$

$$C_S = \kappa^{-1}(C_S). \quad (5.15)$$

□

For a constraint network $\mathcal{R}$ we will use $\kappa(\mathcal{R})$ to denote the network where all constraints are in the CPR.

Katajainen and Madsen [2002] presented a heuristic utilizing randomization for the κ operator. For a constraint C_S, the running time of the heuristic is $\mathcal{O}(|S|^2|C_S| + |S||C_S| \log_2 \min\{d, |C_S|\})$ in the average case.

5.3 Tree Clustering

Based on results from the theory of relational databases, Dechter and Pearl [1989] presented a general and systematic method of decomposition called *tree clustering*. The main idea is to replace the initial constraints with a new set of constraints, that allow the same set of solutions, such that the corresponding dual graph forms a tree. This method can thus be used to construct an acyclic network.

We start out with some definitions which are adapted here to the terminology presented in the preceding chapters.

Definition 5.4. Let $G = (V, E)$ be an undirected graph. G is *chordal* if every cycle with at least four distinct nodes has an edge joining two non-adjacent nodes in the cycle. Such an edge is called a *chord*. A hypergraph is chordal if the corresponding primal graph is chordal. □

Definition 5.5 ([Berge, 1973]). A hypergraph $H = (V, S)$ is *conformal* if, for every clique I in the primal graph of H, there is a hyperedge in S that contains I. □

Definition 5.6 ([Beeri et al., 1983]). For a constraint network $\mathcal{R} = (X, D, C)$ a *join tree* for $\mathcal{R}$ is a tree with a set of nodes C such that

1. each edge $\{C_R, C_S\}$ is labeled by the set of variables $R \cap S$, and
2. for every pair $C_R, C_S \in C$ and for every $x \in R \cap S$ each edge along the unique path between C_R and C_S includes x. □

If a join tree exists. it is easy to identify as the following theorem shows.

Theorem 5.2 ([Maier, 1983]). *If a constraint network $\mathcal{R}$ has a join tree, it is a maximum spanning tree of the dual graph where the edges are weighted with the number of shared variables.*

Verifying if a constraint network has a join tree can thus be reduced to finding a maximum spanning tree on the dual graph and verify that this tree satisfies Definition 5.6. Constraint networks that have a join tree can also be characterized by properties of the constraint hypergraph. This is formalized in Theorem 5.3 on the next page.

Theorem 5.3 ([Beeri et al., 1983]). *Let $\mathcal{R} = (X, D, C)$ be a constraint network. A join tree exists for C if and only if the constraint hypergraph of $\mathcal{R}$ is chordal and conformal.*

Identifying conformal constraint hypergraphs can be achieved by finding the maximal cliques of the corresponding primal graph as the following theorem states.

Theorem 5.4 ([Beeri et al., 1983]). *A hypergraph H is reduced and conformal if and only if its hyperedges are precisely the maximal cliques of a graph. If there is such a graph, then the graph is the primal graph of H.*

The preceding theory give us a method for finding a join tree for a set of constraints. We can simply make the primal graph chordal and identify the maximal cliques. If we create a constraint for each of the maximal cliques, then the dual graph will have a join tree that is a maximum spanning tree. In the following, the conceptual steps of the tree clustering method for a constraint network $\mathcal{R} = (X, D, C)$ are listed in more detail.

1. Make the primal constraint graph (and thus the constraint hypergraph) chordal by adding redundant universal binary constraints to the primal constraint graph.

2. Identify all the maximal cliques $\{X_1, \ldots, X_\ell\}$ in the chordal primal graph.

3. Replace the set of constraints in C with a new set of constraints, one constraint for each maximal clique. Then we have $C = \{C_{X_1}, \ldots, C_{X_\ell}\}$ where each constraint C_{X_i} is a constraint with scope X_i such that

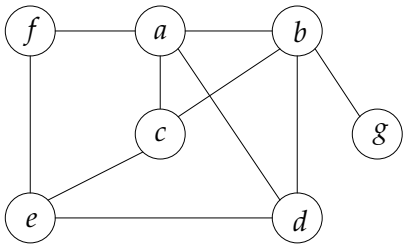
$$C_{X_i} = \pi_{X_i}(Sol(\mathcal{R})). \tag{5.16}$$

4. Find a join tree for $\mathcal{R}$ as a maximum spanning tree of the new dual graph. By definition, the new constraint hypergraph of $\mathcal{R}$ is chordal (because of step 1) and conformal (step 2 and 3 together with Theorem 5.4). Theorem 5.3 ensures that a join tree for $\mathcal{R}$ exists.

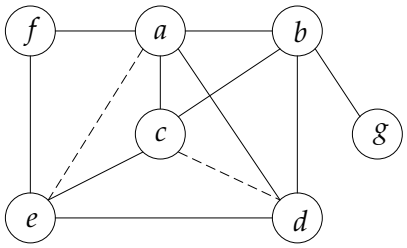
Example 5.2. The following example illustrates the steps of the tree clustering method. Let $\mathcal{R}$ be a constraint network with 7 variables $\{a, \ldots, g\}$ and 8 constraints $\{C_{af}, C_{bc}, C_{bd}, C_{bg}, C_{ce}, C_{de}, C_{ef}, C_{abc}\}$. The primal graph of $\mathcal{R}$ is depicted in Figure 5.4(a) on the facing page.

The first step is to make the primal graph chordal by adding redundant constraints. The resulting chordal primal graph is depicted in Figure 5.4(b) where the new constraints are drawn using dotted lines. The second step is to identify the maximal cliques of the chordal primal graph. The maximal cliques are depicted in Figure 5.4(c). For each clique we compute a new

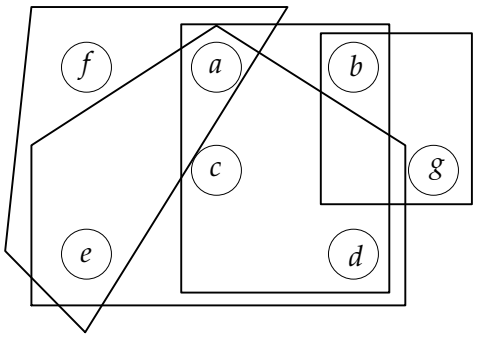
constraint with scope equal to the clique's variables. The dual graph of the new network is depicted in Figure 5.4(d). Finally, we remove redundant constraints from the dual network and we end up with a dual network whose constraint graph is a tree. This is depicted in Figure 5.4(e). □



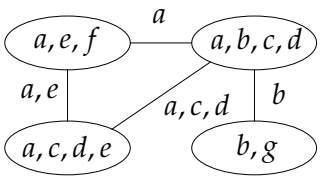
(a) Primal graph.



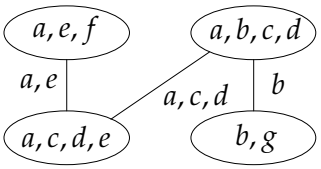
(b) Chordal primal graph. Dotted lines are added universal constraints.



(c) 4 maximal cliques are identified.



(d) Constraint graph of the new dual network. Each clique is turned into a constraint. The constraints are labeled with the shared variables.



(e) Constraint graph of the dual network forms a tree when redundant constraints are removed.

Figure 5.4: Constructing an acyclic network using the tree clustering method.

Complexity

A graph can be made chordal by triangulating it. Finding a minimal triangulation is $\mathcal{NP}$-complete [Yannakakis, 1981], but a heuristic algorithm which runs in $\mathcal{O}(n+e)$ using a *maximal cardinality search* was presented by Tarjan and Yannakakis [1984]. The maximal cliques can be identified in

time $\mathcal{O}(e')$ where e' is the number of edges in the chordal graph [Dechter and Pearl, 1989].

Each constraint C_{X_i} can be viewed as the set of solutions to the subnetwork defined on the variables in X_i and it thus takes time $\mathcal{O}(d^{|X_i|})$ to find each constraint. If we let w denote the size of the largest clique, the overall complexity is thus exponential in w.

5.3.1 Relation To Tree Decomposition

The tree clustering method described in the previous section fits the general graph-theoretic framework of *tree decomposition* introduced by Robertson and Seymour [1986].

Definition 5.7. A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair $(T, \mathcal{B})$, where $T = (J, F)$ is a tree and $\mathcal{B} = \{B_j \mid j \in J\}$ is a family of subsets of V such that

1. $\bigcup_{j \in J} B_j = V$,
2. for all $\{v, w\} \in E$, there exists a node $j \in J$ such that $\{v, w\} \in B_j$, and
3. for all $x, y, z \in J$, if y is on the path from x to z in T, then $B_x \cap B_z \subseteq B_y$.

The *width* of a decomposition $(T, \{B_j \mid j \in J\})$ is $\max_{j \in J} |B_j| - 1$, and the *treewidth* of an undirected graph G, denoted $tw(G)$, is the smallest width of any tree decomposition of G. □

Many $\mathcal{NP}$-complete graph problems can be solved in polynomial time on graphs which have bounded treewidth. The problem of computing the treewidth of a graph is in general $\mathcal{NP}$-hard[Arnborg et al., 1987], however for fixed k, linear-time algorithms exist that determine whether a graph has treewidth at most k, and if so, finds a tree decomposition with width at most k [Bodlaender, 1996]. Their practical use, however, is limited since the running time contains large constants with k in the exponent.

Tree Clustering as Tree Decomposition

The tree clustering method can be seen as a tree decomposition of the dual graph: Let $\{X_1, \ldots, X_\ell\}$ be the maximal cliques. Then $J = \{1, \ldots, \ell\}$ and each subset $B_j, j \in J$ corresponds to some maximal clique X_j. Condition 1 is clearly satisfied. Every clique is maximal so condition 2 is satisfied. The result of tree clustering is a join tree so condition 3 is satisfied. It follows from this that the size of the largest clique is then bounded by $tw(G) + 1$ where G denotes the primal graph of the constraint network.

We should note that while the treewidth of a primal constraint graph is an invariant, the size of the largest clique computed by the tree clustering method is determined by the heuristics used by the triangulation algorithm.

Other Decomposition Methods

Other decomposition methods have been proposed. Gyssens et al. [1994] present a method called *hinge decomposition* and they show that the cardinality of the largest vertex of any hinge decomposition is an invariant of the given network. A hinge tree for a constraint network can be computed in time $\mathcal{O}(ne^2)$.

Recently, a general hypertree decomposition method has been proposed in database research [Gottlob et al., 1999a]. It is similar to tree decomposition, but is now defined on hypergraphs instead of regular undirected graphs. As was the case with a tree decomposition, finding a hypertree decomposition with minimum hyperwidth is $\mathcal{NP}$-complete. A polynomial time algorithm exists for determining whether a hypertree decomposition of hyperwidth at most k exists for any fixed k, and if so, return a hypertree decomposition of hyperwidth at most k [Gottlob et al., 1999b]. The practical relevance of this remains to be seen, but as was the case with a tree decomposition, the worst-case running time of the hypertree decomposition algorithm is exponential in k.

In [Gottlob et al., 2000] various CSP decomposition methods (including tree clustering, hinge decomposition, and hypertree decomposition) were compared. A way of comparing decomposition methods was proposed, in which a method d_1 is more powerful than method d_2 in the sense that whenever d_2 guarantees the decomposed problem can be solved in polynomial time, the problem decomposed using d_1 is also solvable in polynomial time, but there are classes that can be solved in polynomial time using d_1 but not d_2. The conclusion of the paper was that the hypertree decomposition is the most powerful of the current known methods.

One of the authors of [Gottlob et al., 2000] provides a program[1] that can find a hypertree decomposition of width k if such a decomposition exists and returns failure otherwise. The program was briefly tested on some of the constraint network instances used in the performance study presented in Chapter 6, but due to the memory requirements ($k = 3$ required 2GB memory, for $k > 3$ the program crashed with an overflow error), I did not succeed in producing any hypertree decompositions.

5.4 A Combined Method for Acyclic Network Construction

Both array-based logic and tree clustering construct acyclic networks. However the two methods can also be combined: We can utilize array-based

[1] At the time of writing it was available at `http://si.deis.unical.it/~frank/Hypertrees/`.

logic to solve the subproblems induced by the cliques found by tree clustering. Another matter that we have not covered yet is how the CPR affect the algorithms for the fundamental operations described in Chapter 4 on page 37. Both of these issues are explained in the following.

5.4.1 Solutions to Subproblems

As noted in Section 5.3 on page 67, the first part of the tree clustering method identifies a number of cliques, each of which induces a subnetwork for which all solutions must be found. We find these constraints as follows.

1. Place each original constraint C_R in a set S_{X_i} such that $R \subseteq X_i$. Each original constraint C_R forms a clique in the primal graph. If this clique is not maximal, there exists another clique X_j such that $R \subseteq X_j$.

2. For each S_{X_i} construct a constraint C_{X_i}

$$C_{X_i} = \begin{cases} \mathcal{U}_{X_i} & \text{if } S_{X_i} = \emptyset, \\ \mathcal{U}_{X_i} \cap \bowtie_{C_S \in S_{X_i}} C_S & \text{if } S_{X_i} \neq \emptyset. \end{cases} \tag{5.17}$$

Note that the constraints in Equation (5.17) can be tightened further by considering all constraints of the original problem sharing a variable with the clique being processed:

$$C_{X_i} \leftarrow C_{X_i} \cap \pi_{X_i}(\bowtie_{C_S \in C, S \cap X_i \neq \emptyset} C_S). \tag{5.18}$$

5.4.2 Fundamental Operations for CPR Constraints

In a CPR constraint, a single tuple no longer represents a single valid assignment but instead a set of valid assignments. A tuple in a CPR constraint is thus invalid exactly when all the assignments represented by the tuple are invalid. It is possible to determine the set of tuples in a single CPR constraint that become invalid because of unary constraints, as the following lemma shows.

Lemma 5.5. *Let $S = \{x_1, \ldots, x_\ell\}$ be a set of variables and let C_S be a CPR constraint on S. Furthermore, let C_{x_i} be a (possibly universal) unary constraint for each $x_i \in S$. The set of tuples that are invalid because of the unary constraints C_{x_i} are exactly those identified by the set*

$$\{t \mid t \in C_S, t_{|x_i} \cap C_{x_i} = \emptyset \text{ for some } x_i \in S\}. \tag{5.19}$$

Proof. If $t_{|x} \cap C_{x_i} \neq \emptyset$ for all $x_i \in X$, then t clearly represents at least one valid assignment, namely $t_{|x_1} \cap C_{x_i} \times \cdots \times t_{|x_\ell} \cap C_{x_\ell}$. □

As the following lemma shows, it is also possible to determine the opposite: the set of valid domain values can be determined from the set of valid tuples.

Lemma 5.6. *Let $S = \{x_1, \ldots, x_\ell\}$ be a set of variables and let C_S be a CPR constraint on S where all invalid tuples have been removed. Furthermore, let C_{x_i} be a (possibly universal) unary constraint for each $x_i \in S$. Let $\mathcal{R}$ be the constraint network $(S, D, C_S \cup \{x_1, \ldots, x_\ell\})$. Then*

$$\pi_{x_i}(Sol(\mathcal{R})) = C_{x_i} \cap \bigcup_{t \in C_s} t_{|x_i}. \tag{5.20}$$

Proof. For each $a_i \in \pi_{x_i}(Sol(\mathcal{R}))$, clearly $a_i \in C_{x_i}$ and $a_i \in t_{|x_i}$ for some $t \in C_S, C_S \in C$. For each $b_i \in C_{x_i} \cap \bigcup_{t \in C_s} t_{|x_i}$ we have $b_i \in t_{|x_i}$ for some $t \in C_S$. The tuple t contains at least one solution, since otherwise it would have been removed according to Lemma 5.5 on the facing page. Therefore we also have $b_i \in \pi_{x_i}(Sol(\mathcal{R}))$. □

Note that Lemma 5.5 and 5.6 hold, even for constraints in the normal representation, if each scalar value in a tuple is viewed as a singleton set.

Lemma 5.5 and 5.6 show that it is possible to implement the fundamental operations in a CPR network if there is only a single constraint that is not unary. When a user constraint is added, the tuples of the single CPR constraint that become invalid are marked according to Lemma 5.5. The valid domain values of the variables can then be calculated according to Lemma 5.6. This is illustrated in the following example.

Example 5.3. Let the constraint network $\mathcal{R}$ be defined by the single constraint, shown in Figure 5.5(a) on the next page, and universal unary constraints $\{C_a, C_b, C_c\}$. If we replace the universal constraint on b with $C_b = \{0\}$, we see that the last tuple becomes invalid. The resulting constraint is shown in Figure 5.5(b) on the following page. Using Lemma 5.6 we can calculate the valid domain values:

$$D_a = \{0, 1, 2\} \tag{5.21}$$
$$D_b = \{0\} \tag{5.22}$$
$$D_c = \{0, 1\} \tag{5.23}$$

□

Evidently, if we can use the method of array-based logic, we end up with a network having a single constraint and the fundamental operations described in Example 5.3 are applicable. The *state* operation described by Møller [1995] basically uses this method to handle the external influences from the environment and determine the consequences on the system state.

The question is of course: Can we extend this method for networks containing more than one constraint? In the general case, the answer seems to be no as the following example illustrates.

a	b	c
0	$\{0,1\}$	0
$\{1,2\}$	0	1
4	4	1

(a) C_{abc}.

a	b	c
0	$\{0,1\}$	0
$\{1,2\}$	0	1

(b) Invalid tuple removed due to unary constraint $C_b = \{0\}$.

Figure 5.5: Fundamental operations in a CPR network with a single constraint.

Example 5.4. Let the constraint network $\mathcal{R}$ be defined by the two CPR constraints shown in Figure 5.6. None of the constraints contain invalid tuples according to Lemma 5.5, and according to Lemma 5.6, the valid domains for a and b are $\{1,2,3\}$ and $\{0,1,2\}$ respectively. But it is easy to see that the assignment $\langle b, 2\rangle$ can never be part of a solution. □

The problem is that we cannot maintain arc consistency in a CPR network. If we return to Example 5.4, we note that arc consistency in the normal representation would remove the tuples $(3,2,0)$ and $(2,0,1)$ from the constraint C_{abc} and the tuple $(2,2,1)$ from the constraint C_{abd}, and thus correctly ensure that the assignment $\langle b, 2\rangle$ is no longer valid.

In order to enforce arc consistency, we thus still need to verify that each tuple in a constraint, when viewed in the normal representation, has support in adjacent constraints.

This means we can store constraints in the CPR and create an acyclic network using the combined method. However when executing the fundamental operations, we must use time proportional to the number of tuples in the normal representation and not the number of tuples in the CPR.

a	b	c
3	$\{0,2\}$	0
$\{1,2\}$	$\{0,1\}$	1

(a) C_{abc}.

a	b	d
$\{1,3\}$	$\{0,1\}$	0
2	$\{1,2\}$	1

(b) C_{abd}.

Figure 5.6: CPR network with a two constraints.

CHAPTER 6

Performance Study

The algorithms for the fundamental operations described in Chapter 4 and the methods for acyclic network construction described in Chapter 5 have been implemented and the performance has been tested on various problem instances. This chapter contains a brief description of the implementation and the problem instances used, as well as the result of the performance study.

6.1 Implementation

The methods for acyclic network construction described in Sections 5.3 and 5.1, as well as algorithms for the fundamental operations described in Section 4.5 on page 47, have been implemented and this section contains a brief overview of the implementation. The complete source (comprising approximately 7000 lines of non-comment C++ code in 40 files) is available electronically. See Appendix A on page 109 for details on how to obtain and compile the code as well as a brief description of the source files.

Preprocessing

For many practical applications, preprocessing techniques can reduce the size of the combinatorial problem significantly. As an example, Weihe [1998] reported that preprocessing completely solved many instances of a problem, equivalent to the $\mathcal{NP}$-complete *hitting set problem* [Garey and Johnson, 1979], arising from the German and European train schedules. The problems not solved by preprocessing were easily solved by hand.

Two preprocessing rules were applied to the constraint networks used in this performance study:

1. Repeatedly replace two constraints that have the same scope with the single constraint obtained by joining the two constraints.

2. Repeatedly replace two constraints, where the scope of one constraint is a subset of the other, with the single constraint obtained by joining the two constraints.

If constraints are not in the CPR form, these rules will always decrease the total size of the constraint network since the size of the joined result will never be larger than any of the two relations. For constraints in the CPR form, however, this may not be the case since the joined result may be larger than any of the two relations.

For some networks these rules have no effect, but for many of the networks arising from practical applications the effect can be dramatic as the following example shows.

Example 6.1. The "car" instance (see Section 6.2 on page 81 for more details) contains 7883 constraints and has size 2071515. Applying the first reduction rule took 11 CPU seconds and resulted in a network with 849 constraints and size 593403. Applying the second rule took 54 CPU seconds and resulted in a network with 199 constraints and size 697216. Without preprocessing, none of the methods for acyclic network construction were able to complete the compilation. □

Array-Based Logic

The implementation of array-based logic is straightforward. All constraints are compressed and we select a pair of constraints to be joined based on the connectivity factor (cf., Equation (5.6) on page 62). The joined result is compressed and we select the next pair of constraints to be joined. When there is a single constraint left, we are finished.

Tree Clustering

The implementation of the tree clustering method is based on the algorithms described by Tsang [1993]. Triangulation of the primal graph is done using a naive algorithm having $\mathcal{O}(n^2)$ worst-case running time. In all the instances used in the performance study, this turned out not to be a problem since triangulation was performed in a few seconds.

Tsang [1993] notes that the triangulation heuristic can be used with an arbitrary node ordering, and suggests the use of the *maximum cardinality* ordering which has the nice property that no edges are added if the graph is already chordal. The nodes are ordered by first selecting an arbitrary node and then repeatedly selecting the unordered node adjacent to the maximum number of already ordered nodes. However, El Fattah and

Dechter [1996] observed a dramatic difference in the effects of various ordering heuristics on the resulting join tree obtained from the tree clustering method. In all the instances tested, they found that the *minimum degree* ordering produced the smallest cliques.

The minimum degree ordering is computed from last to first by repeatedly selecting the node having the least number of neighbors in the graph (hence minimum degree). When a node has been selected, the node and the incident edges are removed from the graph and the neighbors of the removed nodes are connected. The process is continued until the graph is empty.

George and Liu [1989] give a historical survey of the minimum degree ordering and the improvements made to the basic algorithms. We use the minimum degree ordering implementation provided by the *Boost Graph Library*[1] [Siek et al., 2001].

6.1.1 Correctness

It is difficult to determine if an implementation always computes the correct results. Basically, there are two ways to determine this:

1. Formally prove that the implementation correctly implements the algorithms required. If the algorithms are proved correct, this ensures the correctness of the implementation.

   This is usually not feasible for programs of any significant size and is even more difficult when the implementation language is C++ where side effects and pointer aliasing can occur.

2. Construct an exhaustive list of test instances and manually verify that the correct results are produced.

   Constructing the test instances such that all possible combinations of code paths are tested is very time consuming and difficult without automated tools for testing code coverage.

A pragmatic solution has been chosen to at least ensure some validity of the implementation. It is asserted, that the major steps in the methods compute the correct results (i.e., that the triangulation algorithm in fact computes a triangulation). All the acyclic networks constructed have been tested against a commercial CSP solver[2] on the belief that this solver has received thorough testing and use by various customers. It is verified that the addition of every possible unary constraint has the same effect in both implementations.

[1] The version used was 1.29, available at `www.boost.org`.

[2] Array Database v5.5 by Array Technology A/S, `www.arraytechnology.com`.

6.1.2 Implementation Details

The basic algorithm for compressing constraints is based on that presented in [Katajainen and Madsen, 2002]. The basic algorithm for joining constraints is based on the implementation presented in [Madsen, 2002] but modified according to the suggestions in [Madsen, 2002, p. 29] so that the amount of memory required is proportional to the size of the output.

Data structures

There are basically two ways to layout a relation in memory:

Horizontally decomposed: The values of each tuple form a consecutive byte sequence.

Vertically decomposed: The values of each column form a consecutive byte sequence.

When joining two relations, we need to compare values for the columns common to both relations. This is usually not all the values in a tuple, so the *stride*, i.e., the offset between two subsequently accessed memory addresses, will be larger than one. In [Boncz et al., 2000], experiments show that the stride size is an important parameter for optimizing memory access patterns in join algorithms. On a simple in-memory scan of a single byte from 200000 tuples, an increase of a factor of 8 was measured in execution times when the stride was increased from 1 to 150 bytes.

We therefore choose the vertical decomposition since values of a single column can then be accessed sequentially. This choice has another benefit: the tuple width need not be uniform. This means that values in a compressed column can vary in size.

A consequence of this choice is that all algorithms should preferably work column-wise instead of tuple-wise in order to maximize the locality of memory access.

Representing columns

As noted earlier, each column in a relation contains values from a finite domain D_x. Without loss of generality, we therefore encode the domain values as integers $\{0, 1, \ldots, |D_x| - 1\}$.

There are several ways to represent the sets in a compressed column of a CPR relation. We need to be able to efficiently determine

- whether the intersection between two sets is empty,
- the intersection between two sets, and
- whether a given element exists in a set.

For small domains, a bit vector seems like a natural choice: the intersection operation can utilize the inherent word-parallelism of the bitwise AND operator and the single element test is simply a bit test. But for large domains, the memory usage of the bit vector becomes prohibitive since we assume the sets are sparse. The methods for representing a column are therefore as follows:

Uncompressed columns: Stored as a sequence of integers. The number of bits needed for each integer is $\lceil \log_2 |D_x| \rceil$, $|D_x| \geq 1$. To minimize the memory usage, we represent an integer as the smallest native data type that can hold $\lceil \log_2 |D_x| \rceil$ bits. In the current implementation this is either 8, 16, or 32 bits.

Compressed columns: Stored as a sequence of either fixed-sized bit vectors or (pointers to) sorted integer vectors.

In the current implementation, bit vectors are used when the domain has no more than $2^8 = 256$ values. This enables encoding the set in at most 8 32-bit words. It is likely that it is beneficial to encode even larger domains as bit vectors, since we avoid memory allocation for set elements and we exhibit better cache behavior. The exact size where the encoding should switch to using a sorted vector, should be determined experimentally.

Using optimal representations

The columns are represented using different classes depending on whether a column is compressed and on the domain size of the corresponding variable. The creation of the optimal class for any given variable is implemented in a central factory function, which creates new columns. For compressed columns, it looks as follows.

```
CompressedColumn *CompressedColumn::createNew(Variable &var,
    size_t rowCount)
  {
  size_t max_dom_index = var.getDomainSize()-1;

  if (max_dom_index < 8)
    return new BitmapCompressedColumn<8>(var, rowCount);
  if (max_dom_index < 16)
    return new BitmapCompressedColumn<16>(var, rowCount);
    ...
  if (max_dom_index < 256)
    return new BitmapCompressedColumn<256>(var, rowCount);
  if (max_dom_index <= std::numeric_limits<unsigned char>::max())
    return new ConcreteCompressedColumn<unsigned char>(var,
        rowCount);
  if (max_dom_index <= std::numeric_limits<unsigned short>::max())
    return new ConcreteCompressedColumn<unsigned short>(var,
        rowCount);
```

```
if (max_dom_index ≤ std::numeric_limits<unsigned int>::max())
  return new ConcreteCompressedColumn<unsigned int>(var,
      rowCount);
return new ConcreteCompressedColumn<unsigned long>(var, rowCount
    );
}
```

First we check if we can use a bit vector to represent the column cells. If this is not possible, we select a sequence of sorted integers using the smallest possible data type to represent the integers.

Using Traits Classes to Simplify Implementation

The performance critical part of the code is the functions that implement the compress and join functionality. Since different C++ types are used to represent the column data (depending on the domain size and whether the column is compressed), the compress and join functions should potentially be implemented in several different versions, one for each type of column. This gives maximum performance since the type of the column we are operating on is known to the compiler which makes full optimization possible in the inner loops.

An alternative solution is to create a single version of each algorithm and reference the column data using virtual functions in the column classes. This has the drawback that many virtual function calls are performed in the inner loops, and, more importantly, it is not possible for the compiler to inline the virtual methods.

By using the concept of *traits* classes, originally introduced by Meyers [1995], it is possible to combine the two methods. Here we use the term traits class to refer to a class that aggregates the basic operations (copy, intersection, hash value etc.) on a single cell value in a column. The traits class for a cell containing scalar values has the following signature:

```
template <typename T>struct ScalarCellTraits
  {
  typedef T value_type;
  static std::string toString(T cellValue)
  static T cloneCell(T cellValue)
  static bool intersectionEmpty(T v1, T v2)
  static void expandCell(T cellValue, idvector_t &values)
  static bool equal(T cellValue1, T cellValue2)
  static void hashCell(size_t *hashValue, size_t index, T
      cellValue, unsigned char domainBits)
  };
```

By supplying different traits classes to the same algorithm (e.g., join), this algorithm can be applied to different column types. The compiler will generate an appropriate implementation of the algorithm (using templates), specialized for the types passed. Since the compiler knows the types at the time of compilation, full optimization and in-lining can be applied in the

critical inner loops. In this case, virtual functions are only used to call the correct (compiler generated) function. We have such created a bridge between the *generic programming* paradigm and the *object oriented programming* paradigm.

6.2 Problem Instances

The problem instances used in the performance study fall in various categories. The most interesting category is the configuration problems as these are the problems that require interactivity. The remaining categories are included to see how well the compilation methods perform with other types of networks.

In the following, a number of problem instances, which have been collected from various sources, are described. For all instances, n denotes the number of variables, e the number of constraints, r the largest arity of a constraint, d the size of the largest domain, $|D_x|$ the size of the domain for variable x, $\|R\|$ the size of the network, t the largest number of tuples in a constraint, $|\kappa(R)|$ the size of the network in compressed form, and finally t_κ the largest number of tuples in a compressed constraint.

The symbol † is used to represent a number larger than 1.7×10^{308}, which is the maximal value representable in an 8 byte floating-point number.

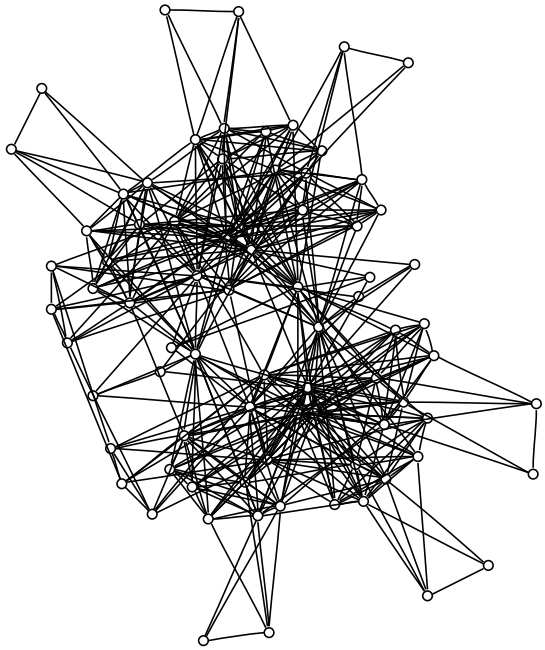
Configuration Problems

The instances presented in Table 6.1 on page 83 are networks which originate from real life configuration problems. The "renault" instance is the test problem described in [Amilhastre et al., 2002] and deals with configuration of a specific family of cars called Renault Megane[3]. The remaining instances have been provided by Array Technology A/S and have been collected during various customer projects. The constraint graphs for the instance "ns11" are depicted in Figure 6.1 on the following page.

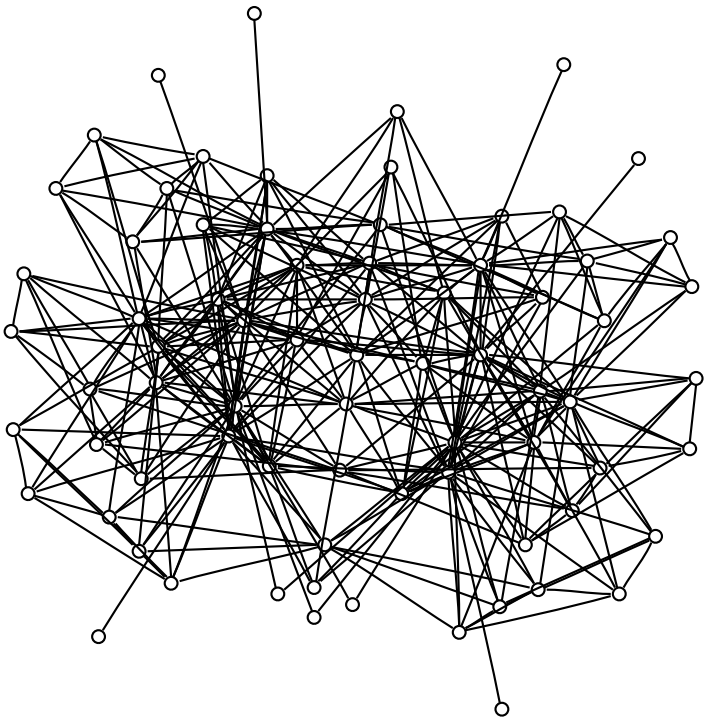
Circuit Verification

The ISCAS '85 benchmark suite consists of a logical description of 10 electronic circuits, provided to authors at the 1985 International Symposium on Circuits and Systems [Brglez and Fujiwara, 1985]. They have subsequently been used by many researchers as a basis for comparing results in the area

[3] At the time of writing, the problem instance was available at `ftp://ftp.irit.fr/pub/IRIT/RPDMP/Configuration/`.



(a) Primal constraint graph.



(b) Dual constraint graph.

Figure 6.1: Constraint graphs for instance ns11.

Name	n	e	r	d	$\Pi_{x \in X} \lvert D_x \rvert$	$\lVert R \rVert$	t	$\lVert \kappa(R) \rVert$	t_κ
renault	101	113	10	42	6.84×10^{49}	1.29×10^{6}	4.87×10^{4}	29279	104
plan-2-25	417	290	16	3	2.32×10^{147}	1.51×10^{5}	2.05×10^{3}	10787	192
plan-31	283	394	30	3	2.30×10^{88}	2.38×10^{12}	1.56×10^{10}	62803	118
car	184	7879	17	42	1.58×10^{82}	1.35×10^{16}	3.95×10^{12}	2071511	17
shelf	50	43	3	132	3.12×10^{76}	1.85×10^{5}	5.15×10^{3}	155850	5151
cf0-td	29	310	7	118	3.70×10^{20}	4.64×10^{7}	8.04×10^{5}	41123	6
heq	1157	586	6	4	†	5.74×10^{4}	9.00×10^{1}	23709	16
ns11	77	74	11	8	4.25×10^{53}	8.19×10^{8}	1.24×10^{7}	34403	296

Table 6.1: Characteristics of configuration problem instances.

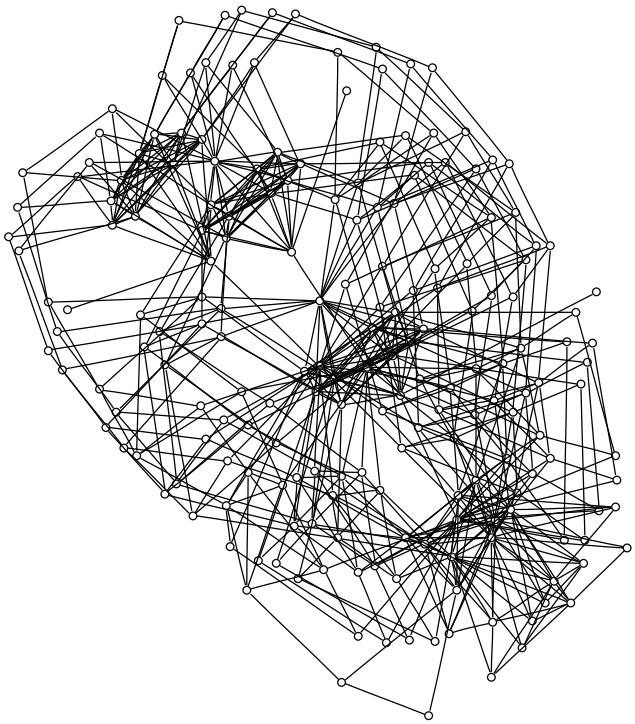
of test generation and circuit verification. Each circuit[4] is described in Table 6.2.

From each circuit a constraint network is created. There is a Boolean variable for each gate's output and for all the circuit inputs. For each connection in the circuit a corresponding constraint is created which maintains the relations between a gate's input and output. The characteristics of the resulting networks are presented in Table 6.3 on page 85. While these problems do not require interactivity, they are included here to see how the compilation methods behave on structured constraint networks. The constraint graphs for the instance "c432" are depicted in Figure 6.2 on the next page.

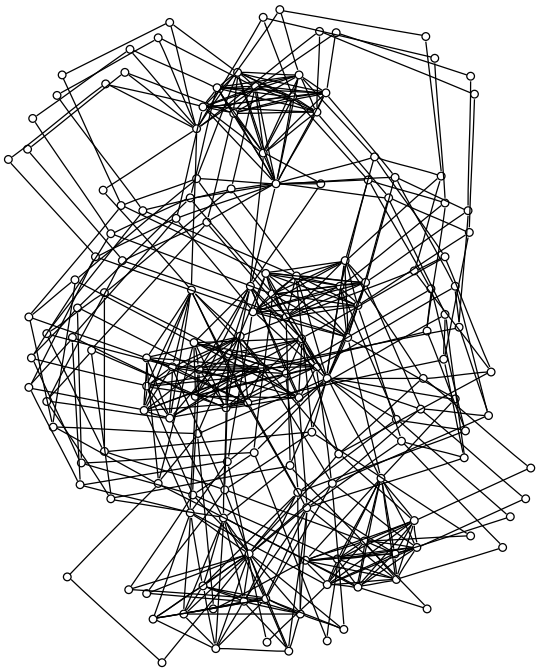
Circuit Name	Circuit Function	Total Gates	Input Lines	Output Lines
c432	Priority Decoder	160 (18 EXOR)	36	7
c499	ECAT	202 (104 EXOR)	41	32
c880	ALU and Control	383	60	26
c1355	ECAT	546	41	32
c1908	ECAT	880	33	25
c2670	ALU and Control	1193	233	140
c3540	ALU and Control	1669	50	22
c5315	ALU and Selector	2307	178	123
c6288	16-bit Multiplier	2406	32	32
c7552	ALU and Control	3512	207	108

Table 6.2: ISCAS '85 benchmark circuits.

[4] At the time of writing, the net-lists for the circuits were available at `http://www.cbl.ncsu.edu/www/CBL_Docs/iscas85.html`.



(a) Primal constraint graph.



(b) Dual constraint graph.

Figure 6.2: Constraint graphs for circuit c432.

Name	n	e	r	d	$\Pi_{x\in X}\lvert D_x\rvert$	$\lVert R\rVert$	t	$\lVert\kappa(R)\rVert$	t_κ
c432	196	160	10	2	1.00×10^{59}	2.02×10^{4}	5.12×10^{2}	2176	10
c499	243	202	6	2	1.41×10^{73}	4.22×10^{3}	3.20×10^{1}	2486	6
c880	443	383	5	2	2.27×10^{133}	5.29×10^{3}	1.60×10^{1}	3803	5
c1355	587	546	6	2	5.07×10^{176}	8.10×10^{3}	3.20×10^{1}	5526	6
c1908	913	880	9	2	6.92×10^{274}	2.85×10^{4}	2.56×10^{2}	8389	9
c2670	1426	1193	6	2	†	1.63×10^{4}	3.20×10^{1}	10865	6
c3540	1719	1669	9	2	†	5.61×10^{4}	2.56×10^{2}	16012	9
c5315	2485	2307	10	2	†	4.86×10^{4}	5.12×10^{2}	24267	10
c6288	2448	2416	3	2	†	2.87×10^{4}	4.00×10^{00}	23968	3
c7552	3719	3512	6	2	†	5.09×10^{4}	3.20×10^{1}	32228	6

Table 6.3: Characteristics of circuit verification problem instances.

Satisfiability Problems

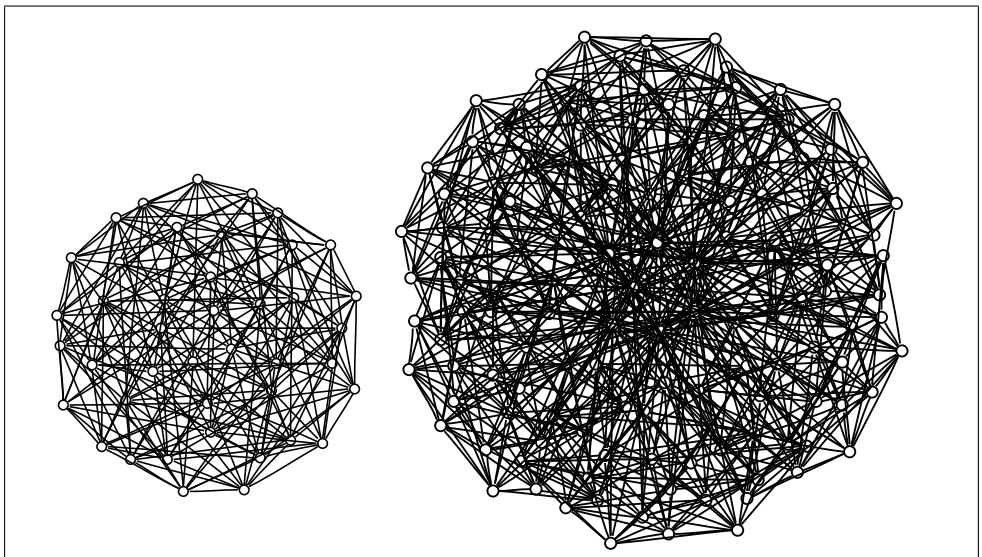
The instances presented in Table 6.4 are networks which originate from satisfiability benchmarks. All the networks presented are instances of the *pigeon hole problem*. We are asked whether it is possible to place $h+1$ pigeons in h holes such that no hole contains more than one pigeon. This is clearly an unsatisfiable problem.

The encoding of this problem is straightforward[5]. For each pigeon i we have a variable x_{ij} which means that pigeon i is placed in hole j. Then we have $h+1$ clauses which say that a pigeon has to be placed in some hole. For each hole we have a set of clauses ensuring that only a single pigeon is placed into that hole. This encoding leads to a total of $h(h+1)$ Boolean variables and $(h+1)+h(h(h+1)/2)$ constraints. As can be seen in Figure 6.3 on the next page, both the primal and dual constraint graphs are highly symmetric, which makes graph-based decomposition difficult.

Name	n	e	r	d	$\Pi_{x\in X}\lvert D_x\rvert$	$\lVert R\rVert$	t	$\lVert\kappa(R)\rVert$	t_κ
hole6	42	133	6	2	4.40×10^{12}	3.40×10^{3}	6.30×10^{1}	987	6
hole7	56	204	7	2	7.21×10^{16}	8.29×10^{3}	1.27×10^{2}	1540	7
hole8	72	297	8	2	4.72×10^{21}	2.01×10^{4}	2.55×10^{2}	2268	8
hole9	90	415	9	2	1.24×10^{27}	4.84×10^{4}	5.11×10^{2}	3195	9
hole10	110	561	10	2	1.30×10^{33}	1.16×10^{5}	1.02×10^{3}	4345	10

Table 6.4: Characteristics of satisfiability problem instances.

[5] At the time of writing, CNF instances were available at `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html`.



(a) Primal constraint graph. (b) Dual constraint graph.

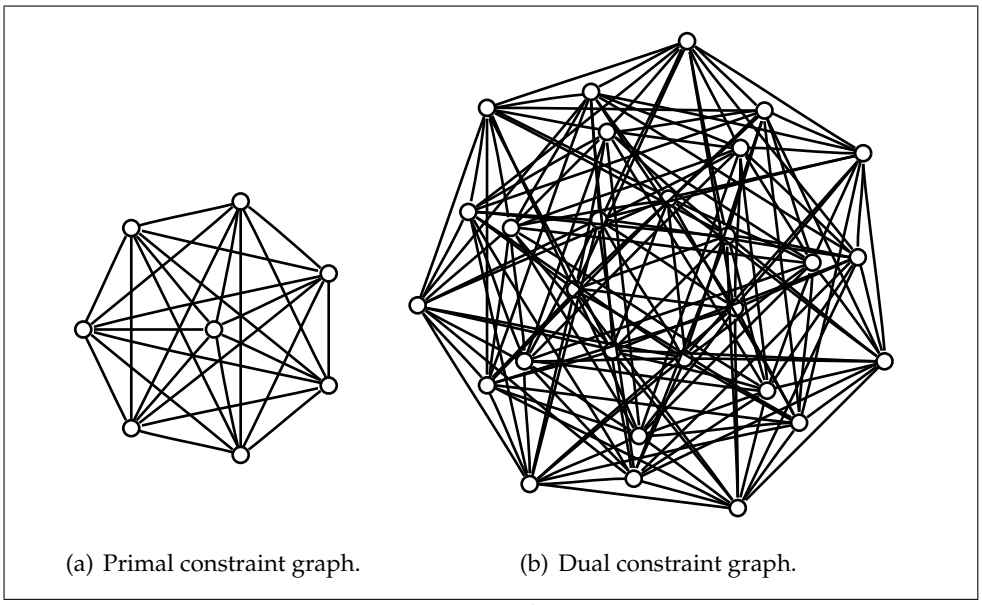
Figure 6.3: Constraint graphs for the pigeon hole problem for $h = 6$.

Puzzles

The instances presented in Table 6.5 are all instances of the n-queen problem for various values of n. The formulation used is similar to that of Example 3.1 on page 20. While the n-queens problem is a popular benchmark, it has very specific features unlikely for to be found in real-life problems. First, all constraints are binary. Second, every variable is constrained by every other variable. This makes graph based decomposition difficult as both the primal and dual constraint graph are completely symmetric. An example for $n = 8$ is shown in Figure 6.4 on the facing page.

Name	n	e	r	d	$\Pi_{x \in X} \lvert D_x \rvert$	$\lVert R \rVert$	t	$\lVert \kappa(R) \rVert$	t_κ
queens8	8	28	2	8	1.68×10^7	2.58×10^3	5.40×10^1	1302	8
queens9	9	36	2	9	3.87×10^8	4.37×10^3	7.00×10^1	2193	9
queens10	10	45	2	10	1.00×10^{10}	6.96×10^3	8.80×10^1	3394	10
queens11	11	55	2	11	2.85×10^{11}	1.06×10^4	1.08×10^2	5111	11
queens12	12	66	2	12	8.92×10^{12}	1.54×10^4	1.30×10^2	7372	12

Table 6.5: Characteristics of puzzle problem instances.



(a) Primal constraint graph. (b) Dual constraint graph.

Figure 6.4: Constraint graphs for the 8-queen problem.

6.3 Experimental Protocol

Two metrics are interesting when trying to asses the practical value of the methods described in the preceding sections:

1. The time it takes to construct an acyclic network for a given problem instance.

2. The worst-case and average-case time it takes to execute one of the three fundamental operations described in Section 4.3 on page 42.

The first metric is obtained by compiling the instances presented in Section 6.2. For each instance, the running time (measured in number of CPU seconds in user mode) and the memory usage (measured in megabytes) are reported. A limit of 2 CPU hours and 512MB memory has been set on the running time and memory usage respectively.

For the second metric, I use the method introduced in [Amilhastre et al., 2002], where we "simulate" the behavior of a user interacting with the constraint solver:

1. Process the variables according to some random ordering.

2. If the number of valid domain values for the variable being processed is 1, continue with the next variable.

3. Order the valid domain values for the variable being processed. Call ADD-CONSTRAINT with the selected variable and domain value.

4. Report the CPU time used in the call to ADD-CONSTRAINT.

The simulation is complete when all variables have a single valid domain value. For each instance, 20,000 simulations are performed and the cumulative distribution function of the measured time is reported as well as the average and worst-case time. We restrict the simulation to test ADD-CONSTRAINT, since the performance of the other fundamental operations should be proportional.

All experiments were carried out on a Dell Inspiron 8200 Laptop with a Pentium 4-M 2GHz CPU, 512MB memory, and running Windows XP. All code has been compiled with Microsoft Visual C++ 7.0 using the -O2 optimization flag.

6.4 Experimental Results

The compilation results obtained using array-based logic are shown in Table 6.6 on the facing page. The symbol $\diamond$ in one of the columns 'CPU' or 'Mem' indicates that the compilation process could not be completed within the given limits for that column. If the compilation succeeded, the characteristics of the resulting acyclic network is reported. Note that for array-based logic, the acyclic network always contains a single n-ary relation.

The compilation results obtained using the tree clustering method are shown in Table 6.7 on page 90. For the tree clustering method, the overall complexity is dependent on the size of the largest clique found in the decomposition phase. This is reported as r, the arity of the largest scope, in the acyclic network. Similarly, the number of cliques is reported as e, the number of constraints in the acyclic network. Note that the size of the cliques found for the circuit verification instances differ from the numbers reported by El Fattah and Dechter [1996]. This is in all likelihood attributed to the use of a different implementation of the minimum degree ordering heuristic.

Simulation Results

The running time of ADD-CONSTRAINT depends quadratically on the number of uncompressed tuples in the acyclic network. The simulations have therefore been restricted to the configuration problem instances where the maximum number of uncompressed tuples is small. The remaining instances all required running times in excess of 1 hour, clearly not feasible for interactive use. The results of the simulations are shown in Table 6.8 on page 90. The percentages shown in the table are the cumulative distribution function of the running times, i.e., for the "car" instance 8% of the calls completed within 0.01 seconds and 87% completed within 0.05 seconds.

Name	CPU	Mem	$\|R\|$	t	$\|\kappa(R)\|$	t_κ
renault		◇				
plan-2-25		◇				
plan-31	◇					
car	◇					
shelf		◇				
cf0-td	44.6	229	5.98×10^{14}	2.14×10^{13}	19625700	135792
heq		◇				
ns11		◇				
c432		◇				
c499		◇				
c880		◇				
c1355		◇				
c1908		◇				
c2670		◇				
c3540		◇				
c5315		◇				
c6288		◇				
c7552		◇				
hole6	2.1	44				
hole7	394.4	70				
hole8		◇				
hole9		◇				
hole10		◇				
queens8	0.23	14	7.36×10^2	9.20×10^1	736	92
queens9	1.9	46	3.17×10^3	3.52×10^2	3168	352
queens10	178.2	75	7.24×10^3	7.24×10^2	7240	724
queens11	5546	248	2.95×10^4	2.68×10^3	29480	2680
queens12		◇				

Table 6.6: Compilation results using array-based logic.

The table shows that most calls complete within a few seconds but for the "renault" instance some calls take more than 6 seconds, barely tolerable for interactive use. This pattern is caused by the fact that , initially, many combinations are valid but as soon as a constraint is added, the number of valid combinations is reduced and the subsequent calls thus require less time to execute.

6.5 Discussion of Results

If we look at the characteristics of the different problem instances, we see that using the CPR for constraints is beneficial, especially for the configuration problems where the domain sizes are large. The extreme case is the

Name	CPU	Mem	e	r	$\|\|R\|\|$	t	$\|\|\kappa(R)\|\|$	t_κ
renault	12.4	10	84	10	1.30×10^6	2.83×10^4	57222	328
plan-2-25	6.3	8	289	23	6.30×10^8	1.03×10^7	128718	1560
plan-31	265.3	130	98	44	7.80×10^{14}	8.80×10^{12}	251879	478
car	180.4	82	166	16	3.99×10^5	1.46×10^4	22458	65
shelf	1.8	5	39	6	8.15×10^9	2.55×10^8	147243	710
cf0-td	9.2	22	21	9	1.31×10^9	1.01×10^8	239909	897
heq	4.1	7	643	10	3.88×10^5	2.59×10^3	44157	40
ns11	16.7	13	40	14	4.56×10^8	1.92×10^7	442034	9265
c432	2.2	5	157	28	1.88×10^9	3.41×10^7	10949	36
c499	2.6	5	195	25	2.00×10^9	3.77×10^7	11150	144
c880	3.6	6	357	26	1.86×10^9	3.36×10^7	11547	18
c1355	4.7	7	435	25	2.01×10^9	3.77×10^7	15078	144
c1908	58.7	33	719	56	1.96×10^{17}	2.53×10^{15}	965458	5340
c2670	17.8	16	1168	38	2.66×10^{12}	6.87×10^{10}	37000	36
c3540	166.6	69	1442	122	5.36×10^{36}	4.15×10^{34}	196139	384
c5315	105.5	48	2050	74	6.12×10^{21}	7.38×10^{19}	173791	256
c6288	133.4	58	1980	59	1.47×10^{19}	1.37×10^{17}	347392	1600
c7552	160.8	45	3036	47	1.40×10^{15}	1.95×10^{13}	203911	256
hole6	33	12	11	28	2.55×10^6	5.15×10^4	400179	10000
hole7	35.6	133	13	36	2.48×10^8	4.67×10^6	5114711	82944
hole8		◇	15	46				
hole9		◇	17	58				
hole10		◇	19	72				
queens8	0.51	45	1	8	7.36×10^2	9.20×10^1	736	92
queens9	2.3	46	1	9	3.17×10^3	3.52×10^2	3168	352
queens10	18.5	49	1	10	7.24×10^3	7.24×10^2	7240	724
queens11		◇	1	11				
queens12		◇	1	12				

Table 6.7: Compilation results using the tree clustering method.

	%(CPU time $\leq$ X)								
Name	0.001	0.005	0.01	0.05	0.1	5.0	10.0	Avg	Max
renault	0%	54%	73%	85%	88%	97%	100%	0.24	6.31
car	0%	0%	8%	87%	89%	100%	100%	0.06	1.09
heq	0%	0%	0%	0%	0%	100%	100%	0.11	0.19

Table 6.8: Results of running simulations of ADD-CONSTRAINT.

"car" instance where the maximum number of tuples is decreased from 3.95×10^{12} to just 17.

The compilation results for array-based logic show, that even if the CPR is used, this method is only feasible for the smallest problems.

The compilation results for the tree clustering method show that it is feasible to use this method to construct an acyclic constraint network. All the configuration problems were compiled in less than 4 minutes, and most problems compiled in only a few seconds.

We also note that for the n-queen instances, tree clustering does not improve the results. This is not surprising as the primal constraint graph is complete in thus only a single clique is found, i.e., the method degenerates to array-based logic. For most real-life problems it is expected, however, that the constraint graph has a structure that enables smaller cliques to be found, as was the case with all the real-life instances used in this performance study.

Tree clustering alone is not sufficient to detect that the pigeon hole problems are unsatisfiable. We need to enforce arc consistency on the acyclic network to detect this.

While the CPR is applicable when constructing the acyclic network, the usefulness is not obvious when it comes to execution of the fundamental operations since the running time depends quadratically on the number of tuples in the uncompressed form. This in turn means that, if we require response times within a few seconds, the methods described in the preceding sections are only applicable when the number of tuples in the uncompressed acyclic network is small.

CHAPTER 7

Uniform Acyclic Networks

The results from Chapter 6 showed that, while the algorithms for the fundamental operations are polynomial in the size of the acyclic network, the acyclic networks constructed from most of the real-life problem instances were too large to solve interactively. The main reason is that the running time of ARC-CONSISTENCY is quadratic in the number of tuples in the uncompressed acyclic network. If we could somehow avoid working with the uncompressed acyclic network, we would stand a much better chance of achieving the desired response time.

In this chapter, I show how we can maintain arc consistency for constraints in the compressed form by imposing an additional requirement on the structure of the acyclic network to get a *uniform acyclic network*. A heuristic is then presented which transforms an acyclic network obtained from the tree clustering method into a uniform acyclic network. The chapter is concluded with experimental results obtained by executing the tree transformation and running the simulation on the resulting uniform acyclic network.

In the following sections, I assume that all constraint networks are using the CPR. All results generalize to networks in the normal representation if each scalar value is treated as a singleton set.

7.1 Fundamental Operations in Uniform Acyclic Networks

It turns out that, if we restrict the acyclic network so that two constraints share at most a single variable, we can maintain arc consistency in the dual network of an acyclic CPR network. I call this restricted acyclic network a

uniform acyclic network.

Definition 7.1. A *uniform acyclic network* is an acyclic network $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ where the constraints share at most a single variable:

$$|C_S \cap C_T| \leq 1, \text{ for all } C_S, C_T \in \hat{C}. \tag{7.1}$$

□

In the following I will show how we can modify the algorithms for the fundamental operations, described in Section 4.5.3 on page 56, to work with a uniform acyclic CPR network. We first replace REVISE-DUAL with REVISE-DUAL-CPR shown in Algorithm 7.1. Whenever we add a unary constraint C_x, we simply update the domain D_x (by maintaining the marks M_x).

```
REVISE-DUAL-CPR(R̂^d, C_x)
 1  empty ← TRUE
 2  for each v ∈ D_x
 3     if v ∈ C_x ∧ M_x[v] = 1
 4        ▷ We have at least one valid domain value
 5        empty ← FALSE
 6     else
 7        M_x[v] ← 0
 8  return empty
```

Algorithm 7.1: Revise domain of a variable in $\hat{\mathcal{R}}$ to reflect unary constraint C_x.

Arc consistency is defined on individual domain values (cf., Definition 3.15 on page 29), so we must define how we identify the individual domain values in the dual network when using the CPR.

Definition 7.2. Let $S = \{x_1, \ldots, x_\ell\}$ be a dual variable and let D_S^d be the corresponding dual domain. The *set of valid dual domain values represented by* D_S^d is

$$\bigcup_{t \in D_S^d} t_{|x_1} \cap D_{x_i} \times \cdots \times t_{|x_\ell} \cap D_{x_\ell}. \tag{7.2}$$

□

Note in particular that if any $t_{|x_1} \cap D_{x_i} = \emptyset$ for some tuple t then t does not represent any valid dual domain values. The main idea is as follows: First we remove all tuples that do not represent any valid dual domain values. From the remaining tuples we can now find the valid domain values for each original variable x_i as

$$D_{x_i} \leftarrow D_{x_i} \cap \bigcup_{t \in D_S^d} t_{|x_i}. \tag{7.3}$$

This is formalized in Algorithm 7.2 and we can now prove the following theorems.

```
REVISE-CPR(S, T)
 1  ▷ Note T is not used, but included to maintain signature compatibility
    with REVISE-2001
 2  changed ← FALSE
 3  ▷ Mark invalid rows
 4  for i ← 1 to |D̂_S^d|
 5     for each x ∈ S
 6        if D̂_S^d[i]_|x ∩ D̂_x = ∅
 7           M̂_S^d[i] ← 0
 8  ▷ Update valid domains
 9  for each x ∈ S
10     M[1..|D̂_x|] ← 0
11     for i ← 1 to |D̂_S^d|
12        if M̂_S^d[i] = 1
13           for each v ∈ D̂_S^d[i]_|x
14              M[v] = 1
15     for j ← 1 to|D_x|
16        if M[j] = 0
17           if D̂_x[j] = 1
18              changed ← TRUE
19              D̂_x[j] ← 0
20  return changed
```

Algorithm 7.2: Mark invalid tuples and update valid domain values of original variables.

Theorem 7.1. *Let S and T be two dual variables such that $S \cap T = \{x\}$. After the calls* REVISE-CPR(S, T), *and* REVISE-CPR(T, S) *S is arc consistent relative to T and T is arc consistent relative to S, when we consider the set of valid domain values, as defined in Definition 7.2 on the preceding page, represented by D_S^d and D_T^d.*

Proof. Lines 4–7 mark the tuples that do not represent any valid domain values. Lines 10–14 compute the union $\bigcup_{t \in D_S^d} t_{|x_i}$ and lines 15–19 compute the set D_{x_i} according to Equation (7.3) on the facing page.

To prove S arc consistent with T, we must prove that for all $u \in D_S^d$ that represents a valid domain value and all $a \in u_{|x} \cap D_x$, there exists a tuple $v \in D_T^d$ that represents a valid domain value such that $a \in v_{|x} \cap D_x$. We know that $a \in D_x$. In the call to REVISE-CPR(T, S) we computed D_x

according to Equation (7.3) on page 94 so we have

$$a \in D_x \subseteq \bigcup_{u \in D_T^d} u_{|x}. \tag{7.4}$$

That T is arc consistent with S is proved similarly. □

Theorem 7.2. *Let $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ be a uniform acyclic CPR network. Let $\hat{\mathcal{R}}^d$ be the corresponding dual network. If we replace* REVISE-2001 *with* REVISE-CPR *shown in Algorithm 7.2 on the preceding page,* ARC-CONSISTENCY *(Algorithm 4.6 on page 55) maintains arc consistency in $\hat{\mathcal{R}}^d$ in worst-case time $\mathcal{O}(\hat{r}\hat{e}\hat{d}\hat{t}_\kappa^2)$.*

Proof. We first note that for each arc $\{S, T\}$ in the dual network, we call both REVISE-CPR(S, T) and REVISE-CPR(T, S) in the initial loop of ARC-CONSISTENCY. Subsequently, according to Theorem 7.1, the only way S can become arc inconsistent relative to T is if a tuple in D_S^d becomes invalid. This can only happen if any of the domains of the original variables in S change, however in this case S is added to Q and REVISE-CPR(T, S) will be called and arc consistency restored. The same argument applies when T becomes arc inconsistent relative to S.

For the complexity result, we note that in PROPAGATION, the call to REVISE-CPR only returns FALSE when a new invalid tuple is found. There are at most $\hat{t}_\kappa$ tuples in each dual variable and a total of $\hat{e}$ dual variables. The time complexity of REVISE-CPR is $\mathcal{O}(\hat{r}\hat{d}\hat{t}_\kappa)$ since we can compute the intersections in time at most $\mathcal{O}(\hat{d})$. This proves the result. □

Replacing REVISE-2001 with REVISE-CPR essentially creates an algorithm resembling AC-3, though now working with CPR constraints in the dual network. Note, that, in order not to change the algorithms from Chapter 4, Theorem 7.2 did not utilize the fact that the dual network is a tree. In the case where we call ADD-CONSTRAINT on a original variable x, we can apply REVISE-CPR from root nodes to leaf nodes and back, in the tree rooted at some dual variable containing x. In this case we can achieve arc-consistency in worst-case time $\mathcal{O}(\hat{r}\hat{e}\hat{d}\hat{t}_\kappa)$.

7.1.1 Summary of Results

REVISE-CPR updates the domains of the original variables, so we no longer need VALID-DOMAINS. By making the changes proposed in the preceding section, we can thus execute the fundamental operations in a uniform acyclic network containing CPR constraints. The complexity results are summarized in Table 7.1 on the next page.

Function	Time Complexity	Space Complexity
ADD-CONSTRAINT	$\mathcal{O}(\hat{r}\hat{e}\hat{d}\hat{t}_\kappa^2)$	$\mathcal{O}(\hat{e}\hat{t}_\kappa)$
REMOVE-CONSTRAINT	$\mathcal{O}(\hat{r}\hat{e}\hat{d}\hat{t}_\kappa^2)$	$\mathcal{O}(\hat{e}\hat{t}_\kappa)$
RESTORATION	$\mathcal{O}(\lvert H\rvert\hat{r}\hat{e}\hat{d}\hat{t}_\kappa^2)$	$\mathcal{O}(\hat{e}\hat{t}_\kappa)$

Table 7.1: Complexity results for the fundamental operations in a uniform acyclic network.

7.2 Uniform Acyclic Network Construction

A uniform acyclic network can be constructed by a tree transformation that transforms the tree generated by the tree clustering method into a uniform acyclic network.

A new relational operation called *split* forms the basis of the transformation. When a constraint is split, we augment the constraint with a new meta variable which, for each tuple, has the same valid value as the tuple's index. The split relation is then the projection of this meta variable on a given set of variables.

Definition 7.3. Let S be a relation with scheme Y, $Z \subset Y$ be a set of attributes, and let $\lambda \notin Y$ be a *meta attribute*. The *split* of S on (Z, λ), denoted $\tau_{Z,\lambda}(S)$ is a relation with scheme $Z \cup \{\lambda\}$ and instance

$$\{t[i]_{|Z} \times \{i\} \mid t[i] \in S, 1 \leq i \leq |S|\}. \tag{7.5}$$

□

The split operation can be carried out in time $\mathcal{O}(|S|)$. Note that for a relation S with scheme Y, a set of attributes $Z \subset Y$, we have by definition

$$S = \pi_Y(\tau_{Z,\lambda}(S) \bowtie \tau_{Y-Z,\lambda}(S)). \tag{7.6}$$

The effect of the split operator is illustrated in Figure 7.1 on the following page. It is easy to see that if we join the two split relations, we will get the original relation from Figure 7.1(a).

The following lemma shows that we can replace a constraint by two new constraints obtained by splitting the original constraint.

Lemma 7.3. *Let $\mathcal{R} = (X, D, C)$ be a constraint network and $C_S \in C$ a constraint. Let $T \subset S$ be a set of variables, and let λ be a* meta variable. *Furthermore, let $\mathcal{R}_\lambda = (X_\lambda, D_\lambda, C_\lambda)$ where*

$$X_\lambda = X \cup \{\lambda\} \tag{7.7}$$

$$D_\lambda(x) = \begin{cases} D(x) & \text{if } x \in X \\ \{0, \ldots, |C_S| - 1\} & \text{if } x = \lambda \end{cases} \tag{7.8}$$

$$C_\lambda = C - C_S \cup \tau_{T,\lambda}(C_S) \cup \tau_{S-T,\lambda}(C_S). \tag{7.9}$$

a	b	c
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

(a) Relation in the normal representation.

a	λ
0	0
0	1
0	2
1	3
1	4

(b) $\tau_{a,\lambda}$.

b	c	λ
0	0	0
0	1	1
1	1	2
0	1	3
1	1	4

(c) $\tau_{b,c,\lambda}$.

Figure 7.1: Splitting the relation in Figure 7.1(a).

Then $Sol(\mathcal{R}) = \pi_X(Sol(\mathcal{R}_\lambda))$.

Proof. Follows from Equation (7.6) on the page before. □

At first it would seem that splitting a relation only increases the size of a network, however if we are using the CPR, we can usually reduce the total size, as illustrated in Figure 7.2 on the facing page, since removing variables from a constraint may allow further compression. We also note that, when a constraint is split, the two new constraints are linked by a single variable as required in a uniform acyclic network.

For any constraint C_S and set of variables $Z \subset S$, if we replace the constraint C_S with two new constraints $\tau_{Z,\lambda}(C_S)$ and $\tau_{S-Z,\lambda}(C_S)$, the meta variable λ represents a subset of valid tuples in the original constraint C_S.

Example 7.1. If we look at Figure 7.2 on the next page, we see that the assignment $\langle \lambda, 3 \rangle$ in the two split constraints in Figure 7.2(e) and Figure 7.2(f) is equivalent to the following set of assignments in the original constraint:

a	b	c	d
1	1	1	0
1	1	1	1

□

Let $\{S_1, \ldots, S_{\hat{e}}\}$ be a width 1 ordering of the dual variables of an acyclic network (i.e., an ordering obtained by a breadth-first search in the tree rooted at an arbitrary dual variable). For any dual variable S_i and its parent S_j, let $T_{ij} = S_i - S_j$. By definition, (cf., Definition 5.6 on page 67) the original variables in T_{ij} are contained only in the dual variable S_i and its children, not in S_j or any other dual variable at higher levels in the tree. If $T_{ij} \neq \emptyset$ it is therefore possible to split the constraint C_{S_i} and replace the original variables T_{ij} in C_{S_i} with a single meta variable that represents the subset of valid tuples for T_{ij}. The benefits are twofold: We potentially reduce the size of C_{S_i} and the two new relations are linked by a single meta variable as required in a uniform acyclic network.

a	b	c	d
0	0	0	1
0	0	1	0
0	0	1	1
1	1	0	1
1	1	1	0
1	1	1	1

(a) Uncompressed relation.

a	b	c	d
0	0	0	1
0	0	1	$\{0,1\}$
1	1	0	1
1	1	1	$\{0,1\}$

(b) Relation in CPR.

a	b	λ
0	0	0
0	0	1
1	1	2
1	1	3

(c) $\tau_{a,b,\lambda}$.

c	d	λ
0	1	0
1	$\{0,1\}$	1
0	1	2
1	$\{0,1\}$	3

(d) $\tau_{c,d,\lambda}$.

a	b	λ
0	0	$\{0,1\}$
1	1	$\{2,3\}$

(e) $\tau_{a,b,\lambda}$ in CPR.

c	d	λ
0	1	$\{0,2\}$
1	$\{0,1\}$	$\{1,3\}$

(f) $\tau_{c,d,\lambda}$ in CPR.

Figure 7.2: Splitting a relation stored in the CPR. Note that the size of the relation in (a) is 24, (b) is 18, (c) and (d) combined is 26, and (e) and (f) combined is 17.

If the new constraint containing T_{ij} shares more than 1 variable with its children, they must be joined to a single constraint. But this join operation is "local" in the sense that the resulting constraint does not influence the size of any constraint which we need to process afterwards.

We can apply the preceding ideas to construct a uniform acyclic network that correctly propagates information. By processing the edges of the join tree in a bottom up fashion, we know that, whenever we process an edge $\{S_i, S_j\}$, where S_i is the child and S_j the parent node, the subtree rooted at S_i is a uniform acyclic network. This is formalized in the algorithm UNIFORM-NETWORK shown in Algorithm 7.3 on the next page.

In line 7 we test if the child constraint C_L contains variables that can be split. If this is the case, lines 8-10 split the child node and add the "local" part to C and create a new constraint C_S which is the join of the parent constraint C_P and the non-local part of the child constraint. If we cannot split the child constraint, line 12–13 simply join the child with the parent to get the constraint C_S.

We now check if we can split C_S. If C_S does not share any variables

with another constraint (i.e., it is a root node) or all the variables in S are shared with adjacent constraint, we simply replace the parent node C_P with C_S (line 18). Otherwise, C_S is split on the variables in S that are not used in adjacent constraints (lines 20–21).

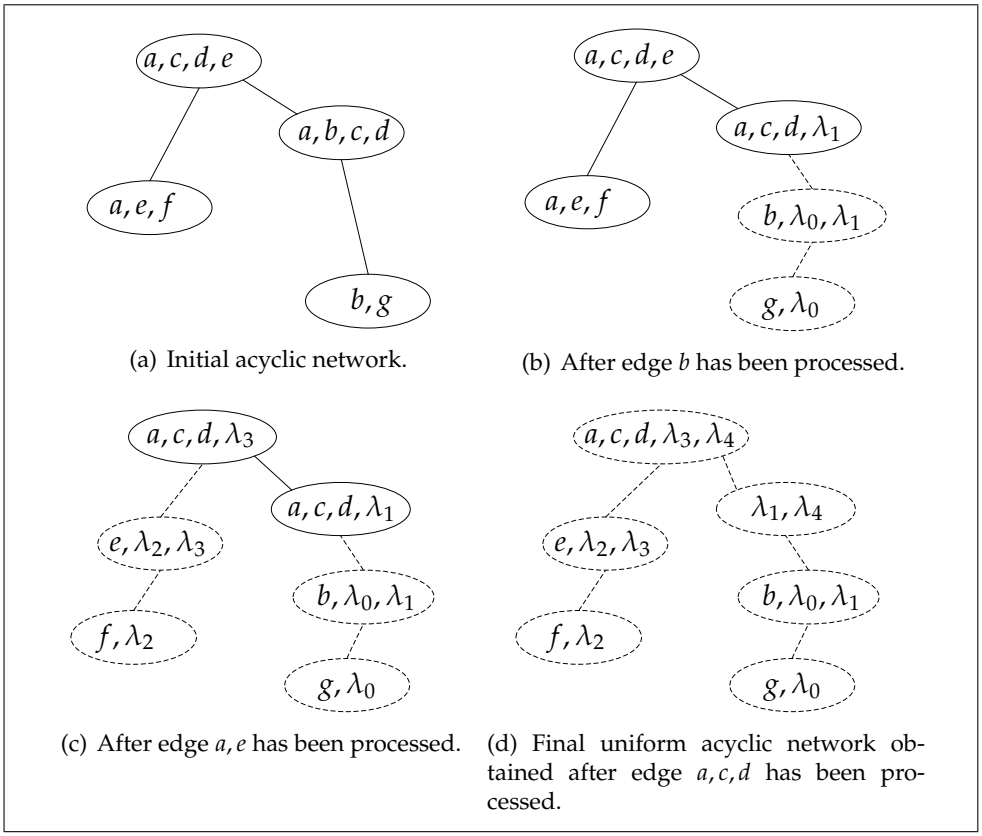
```
UNIFORM-NETWORK(T, R̂)
 1  ▷ T is the join tree for R̂
 2  i ← 0
 3  ▷ Traverse T bottom up
 4  for each unprocessed edge {C_L, C_P} ∈ T
 5     ▷ Assume C_L is the lower level node and C_P its parent
 6     C ← C − {C_L}
 7     if |L − P| > 0
 8        C ← C ∪ τ_{L−P,λ_i}(C_L)
 9        S ← P ∪ {λ_i}
10        C_S ← τ_{L∩P,λ_i}(C_L) ⋈ C_P
11     else
12        S ← L ∪ P
13        C_S ← C_L ⋈ C_P
14     ▷ J contains variables that C_P shares width adjacent vertices
15     J ← {x | x ∈ P ∩ Q, Q ≠ L, {C_P, C_Q} ∈ T}
16     K ← S − J
17     if J = ∅ ∨ K = ∅
18        Replace C_P with C_S
19     else
20        C ← C ∪ τ_{K,λ_{i+1}}(C_S)
21        Replace C_P with τ_{J,λ_{i+1}}(C_S)
22     i ← i + 2
23     Mark edge {C_L, C_P} as processed
```

Algorithm 7.3: Transforming an acyclic network into a uniform acyclic network.

Example 7.2. Figure 7.3 on the facing page illustrates how the algorithm UNIFORM-NETWORK constructs a uniform acyclic network from the acyclic network shown in Figure 5.4(e) on page 69. The following table shows the contents of the different sets when each edge is being processed. The column "New Constraints" contains the constraints that are added in lines 10 and 18.

Edge	L	P	S	J	K	New constraints
$\{b\}$	$\{b,g\}$	$\{a,b,c,d\}$	$\{a,b,c,d,\lambda_0\}$	$\{a,c,d\}$	$\{b,\lambda_0\}$	$\{C_{g,\lambda_0}, C_{b,\lambda_0,\lambda_1}\}$
$\{a,e\}$	$\{a,e,f\}$	$\{a,c,d,e\}$	$\{a,c,d,e,\lambda_2\}$	$\{a,c,d\}$	$\{e,\lambda_2\}$	$\{C_{f,\lambda_2}, C_{e,\lambda_2,\lambda_3}\}$
$\{a,c,d\}$	$\{a,c,d,\lambda_3\}$	$\{a,c,d,\lambda_1\}$	$\{a,c,d,\lambda_3,\lambda_4\}$	$\{\}$	$\{a,c,d,\lambda_3,\lambda_4\}$	$\{C_{\lambda_3,\lambda_4}\}$

□



(a) Initial acyclic network.

(b) After edge b has been processed.

(c) After edge a, e has been processed.

(d) Final uniform acyclic network obtained after edge a, c, d has been processed.

Figure 7.3: Constructing a uniform acyclic network. Dashed lines represent edges and nodes that have been processed.

7.2.1 Correctness and Complexity

Theorem 7.4. *Let $\hat{\mathcal{R}} = (\hat{X}, \hat{D}, \hat{C})$ be an acyclic network. Let $\hat{\mathcal{R}}^u$ be the network obtained by running* UNIFORM-NETWORK *on $\hat{\mathcal{R}}$. Then*

1. *$\hat{\mathcal{R}}^u$ is equivalent to $\hat{\mathcal{R}}$, and*
2. *$\hat{\mathcal{R}}^u$ is a uniform acyclic network.*

Proof. To prove the first part, it is sufficient to prove that, when an edge in the join tree of $\hat{\mathcal{R}}$ is processed, it does not change $\pi_{\hat{X}}(Sol(\hat{\mathcal{R}}^u))$. We first note that we have the following invariant in line 14:

$$Sol(\hat{\mathcal{R}}) = \pi_{\hat{X}}(Sol(\hat{X}, \hat{D}, \hat{C} - \{C_P\} \cup \{C_S\})). \tag{7.10}$$

If we enter the branch at lines 12-13 this invariant should be clear. If we enter the branch at lines 8-10 the invariant follows from Lemma 7.3 and the

fact that $L - P = L - L \cap P$. If we enter line 18 we have thus proved equivalence. If we enter lines 20–21 the equivalence follows from Lemma 7.3 and the fact that $K = S - J$. This proves the first claim.

We note that two constraints C_L and C_P connected by an edge is replaced by at most 3 new constraints (lines 8, 20, and 21). These new constraints are connected with the meta variables λ_i and λ_{i+1} and form a non-cyclic path. This proves the last claim. □

As we approach the top of the tree, the domain size of the meta variables added in lines 8 and 20 grows, since each domain value represents a number of valid assignments to the original variables contained in the subtree rooted at C_L. In the worst case, when the constraints in the subtrees cannot be compressed, the domain size is exponential in the number of original variables contained in the subtrees. This follows from the fact that the number of solutions to the subproblems induced by the subtrees is exponential in the number of variables and we must be able to address each solution by using the meta variable, since this is the only connection to the remaining variables.

The complexity of processing and edge is thus equivalent to finding the solutions to a network induced by the variables of constraint C_P and all its subtrees. Each subtree contribute with a single meta variable, having domain size exponential in the number of variables in the subtree. If we let u denote the number of subtrees, d_λ the size of the largest domain in a meta variable contained in a direct child constraint of C_P, the complexity can be expressed as $\mathcal{O}(d^{|S|+u} + d_\lambda^{|S|+u})$.

7.3 Experimental Results

The algorithms for uniform acyclic networks have been implemented and their performance have been evaluated using the methods described in Chapter 6.

When an edge must be selected for processing in line 4 of UNIFORM-NETWORK, there may be several candidate edges available. In the present implementation, I simply use Equation (5.6) to select the edge connecting a constraint pair with the smallest connectivity factor.

The compilation results obtained when transforming an acyclic network into a uniform acyclic network are shown in Table 7.2 on the facing page. The results of running simulations on the uniform acyclic networks are shown in Table 7.3 on the next page.

Name	Time	Memory	$\|R\|$	t	$\|\kappa(R)\|$	t_κ
renault	10.6	16	4.61×10^5	3.96×10^4	117732	3682
plan-2-25	1854	219	6.47×10^7	3.00×10^6	4529709	166586
plan-31		◇				
car	1	4	3.10×10^4	1.82×10^3	16679	486
shelf	767.3	87	2.83×10^7	3.30×10^6	3011637	49054
cf0-td	326.6	61	2.09×10^8	2.27×10^7	1648769	34919
heq	2.7	5	1.69×10^5	3.07×10^3	96968	1306
ns11	2423	172	9.89×10^7	8.10×10^6	2019483	37209
c432	33.9	39	4.99×10^5	2.50×10^4	189224	5011
c499		◇				
c880	◇					
c1355		◇				
c1908		◇				
c2670		◇				
c3540		◇				
c5315		◇				
c6288		◇				
c7552	◇					

Table 7.2: Compilation results for creating a uniform acyclic network.

	%(CPU time $\leq$ X)								
Name	0.001	0.005	0.01	0.05	0.1	0.5	2.0	Avg	Max
renault	72%	88%	92%	99%	100%	100%	100%	$<$ 0.01	0.06
plan-2-25	82%	83%	84%	89%	93%	98%	100%	0.03	1.53
car	86%	98%	100%	100%	100%	100%	100%	$<$ 0.01	0.01
shelf	20%	36%	36%	59%	82%	98%	100%	0.07	0.64
cf0-td	32%	43%	57%	88%	93%	100%	100%	0.02	0.24
heq	94%	99%	99%	100%	100%	100%	100%	$<$ 0.01	0.02
ns11	45%	62%	70%	89%	93%	100%	100%	0.02	0.37

Table 7.3: Results of running simulations of ADD-CONSTRAINT on the uniform acyclic network.

7.4 Discussion of Results

The results obtained by simulating a user interacting with the constraint solver on a uniform acyclic network show, that the running time is less than a few seconds in the worst case. This should be acceptable for interactive use.

If we were to simulate the use of RESTORATION (which has running time proportional to the number of constraints that have been added, cf., Table 7.1 on page 97), the worst-case running time would probably be acceptable. This follows from the fact that very few constraints exhibit the

worst-case behavior, as can be seen in the cumulative distribution function.

The use of uniform acyclic networks is not without problems. The time needed for compilation increases, and some instances cannot be transformed due to the memory usage of the transformation algorithm.

An initial investigation reveals at least one source of this problem: The frequent use of the join, the project, and the compress operator results in a lot of redundant information in the sense that when we expand a CPR constraint into the normal representation, many duplicate tuples exist.

To asses the amount of redundant information, a brute force algorithm was implemented that, for each tuple t in a constraint, removes all tuples that are completely contained in t. For a constraint C_S, a tuple $u \in C_S$ is contained in a tuple $t \in C_S$ if $u_{|x} \subseteq t_{|x}$ for all $x \in S$.

Using this algorithm on the "c499" instance, resulted in the construction of a uniform acyclic network at the expense of using several CPU days. It also showed that, in extreme cases, as many as 95% of the tuples in a CPR constraint could be removed. More realistically, the average amount of tuples removed was around 45%.

CHAPTER 8

Conclusion

The principal achievements of this work — listed in order of appearance — have been:

I. In Chapter 3, I surveyed of some of the fundamental definitions, concepts and algorithms relating to the area of classical constraint satisfaction. The decision version of the constraint satisfaction problem was proved $\mathcal{NP}$-complete in Section 3.2.

II. In Chapter 4, I identified a number of usability requirements relating to the construction of an interactive constraint solver.

III. In Section 4.3, I formally defined 3 fundamental operations that form the basis of an interactive constraint solver. Algorithms with polynomial running time were proposed for these operations in Section 4.5.

IV. Chapter 5 contained a survey of two existing methods for compiling constraint networks: array-based logic and tree clustering. In Section 5.4, I showed how these methods could be combined to create a new compilation method.

V. All algorithms and methods presented in Chapters 4, 5 and 7 have been implemented. Chapter 6 contained an overview of the implementation

VI. In Section 6.4, I presented the results of an experimental evaluation of the proposed methods. The results showed the new compilation method to be applicable to most of the networks considered.

    The results also showed that the running time of the algorithms for the fundamental operations was too long for interactive use in all but the smallest instances.

VII. To improve the response time of the fundamental operations, I defined the *uniform acyclic network* in Chapter 7 and showed how it enabled the fundamental operations to be carried out on constraints stored using the Cartesian product representation. A tree transformation heuristic was proposed to transform an acyclic network into a uniform acyclic network.

VIII. An experimental evaluation, presented in Section 7.3, showed that the running time of the algorithms for the fundamental operations, when applied to uniform acyclic networks, was acceptable for interactive use in all instances where a uniform acyclic network could be constructed.

Constructing a uniform acyclic network turned out to be practical for most networks arising from the area of product configuration. For networks in other categories, the proposed tree transformation algorithm failed in most cases. The failure was caused by excessive memory usage.

Creating an implementation can be a two edged sword. At one hand an implementation provides practical evidence that the proposed methods work and it enables an experimental evaluation to be carried out. On the other hand, a lot of time is spent on the implementation, correcting small bugs etc. Time that could otherwise be spent to further improve the theoretical results.

I feel, however, that the benefits in this case have outweighed the drawbacks. Partly because I favor theoretical results that are useful in practice but also because experiments can provide new insights. One example is in the performance of the fundamental operations. From a theoretical point of view, we are usually content when a polynomial time solution can be found for a difficult problem. However, experiments showed that while the algorithms for the fundamental operations had polynomial running time on an acyclic network, the response time achieved for most instances, was not short enough to be used interactively. We therefore had to impose further restrictions on the structure of the network for the methods to be useful in an interactive setting.

In summary, the methods I have proposed can be used to solve many real-life constraint satisfaction problems interactively.

8.1 Directions for Further Work

There are a number of issues that I have chosen not to pursue within the scope of this thesis. The following is a list of suggestions for further work related to the material presented in the preceding sections.

I. Finding a good decomposition is important for the running time of the compilation methods. In the present implementation a heuristic is used for triangulating the primal graph. Bodlaender et al. [2003] describe a number of pre-processing rules for triangulation of probabilistic networks defined on undirected graphs similar to the primal graphs of constraint networks. These rules allow a triangulation to be computed in the pre-processed graph and subsequently mapped back to the original graph without loss of optimality. They note that, for many instances, an optimal triangulation is found by pre-processing and, for most remaining instances the pre-processed graph was small enough to be optimally triangulated using an exact algorithm.

II. Practical experiments with other decomposition methods, such as hinge decomposition [Gyssens et al., 1994], should be carried out.

III. The heuristic used for selecting the order in which relations are to be joined is rather simple. The *join ordering problem* for finding and optimal ordering in which relations should be joined, is a well known $\mathcal{NP}$-complete problem from the area of database research. It is usually solved using dynamic programming, but the large number of constraints present in most real-life constraint networks makes this intractable. Steinbrunn et al. [1997] analyze heuristic, randomized, and genetic algorithm solutions to the join ordering problem. It would be interesting to explore the effects of using more advanced heuristics in the context of constraint satisfaction.

IV. The memory usage seems to be the limiting factor when creating a uniform acyclic network. At some point during the bottom up processing of the tree, the constraints become too large to join. An interesting property at this point, is that the subtrees that have already been processed are uniform acyclic networks. We thus have a forest of uniform acyclic networks.

    It may be possible to combine these uniform acyclic networks without having to join all the remaining nodes. If we could thus spend more time combining the subtrees and in return save memory, we may broaden the range of constraint networks that can be handled.

V. The use of Cartesian products to represent constraints seems to be crucial for the practical applicability of the methods presented in this thesis. But no theoretical results are, to the best of my knowledge, known about the use of Cartesian products to represent constraints. It may be possible to strengthen the complexity results for creating and solving uniform acyclic networks if more knowledge is gained on the use of Cartesian Products.

VI. The use of Cartesian products also cause problems with an increasingly amount of redundant information being created when constraints are repeatedly processed. Finding efficient methods to remove this redundancy, or even better, improving the processing methods to avoid redundancy, should broaden the range of constraint networks that can be handled by the methods proposed in this thesis.

APPENDIX A

Source Code

All the methods and algorithms described in this thesis have been implemented in C++. For the official version of this thesis, I enclose the source code, (comprising approximately 7000 lines of non-comment C++ code in 40 files) on a supplemental CD-ROM. For other readers, it is possible to download the files from:

`www.diku.dk/forskning/performance-engineering/jeppe/`

The code has been developed using Microsoft Visual C++ .Net. The solution file `all.sln` located in the `Source/` directory, contain all the sub projects, which can be compiled by rebuilding the solution.

The code can also be compiled using GCC. The author has successfully compiled the code with GCC v3.2 running under the Cygwin[1] environment. A `makefile` is located in the `Source/` directory.

In order to compile the code a number of packages[2], listed in Table A.1 on the next page, must be installed and the makefile/project file must be updated to reflect the installation paths. The code used for reading the binary benchmark instances and verifying the implementation, relies on proprietary code supplied by Array Technology A/S, so this code is not included. In order to do something useful with the implementation, constraint network instances thus have to be specified directly in the source code by using the supplied classes. Examples can be seen in the directory `Source/test`.

[1] `www.cygwin.com`

[2] CppUnit is only required for performing unit tests.

Package	Version	Available from
STLport	4.5.3	www.stlport.org
Boost	1.29.0	www.boost.org
CppUnit	1.8.0	cppunit.sourceforge.net

Table A.1: Packages required for compilation.

A.1 Copyright

All source code is copyright ©2003, Jeppe Nejsum Madsen. It is distributed under the GNU General Public License[3]. All files contain the following copyright statement:

```
/*
 * Copyright (C) 2003 Jeppe Nejsum Madsen, nejsum@diku.dk
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
```

A.2 Overview

The following table lists the source files available.

Directory `csplib`	
column.hpp	Definition of Column base classes
column.cpp	Implementation of non-template column methods
common.hpp	Main include file
compress.cpp	Implementation of compress/uncompress algorithms
consistency.cpp	Implementation of consistency algorithms
constraint_network.hpp	Definition of ConstraintNetwork class
constraint_network.cpp	Implementation of ConstraintNetwork class
graph.hpp	Definition of primal & dual graph classes

[3]The full license text is available at http://www.gnu.org/copyleft/gpl.html

graph.cpp	Algorithms for creating and manipulating primal/dual graphs
join.cpp	Implementation of join algorithm
relation.hpp	Definition of Relation class
relation.cpp	Implementation of Relation class
uniquerows.hpp	Definition of complement columns
uniquerows.cpp	Implementation of complement columns
variable.hpp	Definition of variable classes
variable.cpp	Implementation of non-template variable methods

Directory `csplib/detail`

column_base.hpp	Base class for implementation of templated column classes
column_bitset.hpp	Column methods specific for compressed column as bit vector
column_compressed.hpp	Column methods specific for compressed column as vector
column_impl.hpp	Main include file for column implementation
column_traits.hpp	Traits classes for all column cell types
column_uncompressed.hpp	Column methods specific for uncompressed columns

Directory `csplib/utility`

formatter.hpp	Definition of class for formatted progress output
formatter.cpp	Implementation of class for formatted progress output
hashfunction.hpp	Definition of STL compatible strongly universal hash function
hashfunction.cpp	Implementation of strongly universal hash function
newbitset.hpp	Implementation of bit vector

Directory `compilecsp`

build.cpp	Implementation of preprocessing and array-based logic algorithms
compilecsp.cpp	Source file for main executable of the compile methods
csp.hpp	Definition of CSP class
csp.cpp	Implementation of CSP methods
treeclustering.cpp	Implementation of tree clustering and uniform network algorithms

Directory `cspruntime`

cspruntime.cpp	Source for main executable of the runtime methods
runtime_network.hpp	Definition of class RuntimeNetwork
runtime_network.cpp	Implementation of fundamental operations
simulatorui.hpp	Definition of simple console based interactive constraint solver
simulatorui.cpp	Implementation of simple console based interactive constraint solver

REFERENCES

J. AMILHASTRE, H. FARGIER, AND P. MARQUIS. Consistency restoration and explanations in dynamic CSPs — application to configuration. *Artificial Intelligence* **135**:199–234, 2002.

S. ARNBORG, D. G. CORNEIL, AND A. PROSKUROWSKI. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods* **7**(2):277–284, 1987.

F. BACCHUS, X. CHEN, P. VAN BEEK, AND T. WALSH. Binary vs. non-binary constraints. *Artificial Intelligence* **140**:1–37, 2002.

C. BEERI, R. FAGIN, D. MAIER, AND M. YANNAKAKIS. On the desirability of acyclic database schemes. *Journal of the ACM* **30**(3):479–513, 1983.

C. BERGE. *Graphs and Hypergraphs*. North-Holland, 1973.

C. BESSIÈRE. Arc-consistency and arc-consistency again. *Artificial Intelligence* **65**(1):179–190, 1994.

C. BESSIÈRE AND J. RÉGIN. Refining the basic constraint propagation algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 309–315. Morgan Kaufmann, 2001.

J. BITNER AND E. REINGOLD. Backtrack programming techniques. *Communications of the ACM* **18**:651–656, 1975.

H. L. BODLAENDER. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing* **25**(6): 1305–1317, 1996.

H. L. BODLAENDER, A. M. C. A. KOSTER, AND F. VAN DEN EIJKHOF. Preprocessing rules for triangulation of probabilistic networks. Technical Report UU-CS-2003-001, Institute for Information and Computing Sciences, Utrecht University, 2003.

P. A. BONCZ, S. MANEGOLD, AND M. L. KERSTEN. Database architecture optimized for the new bottleneck: Memory access. *The VLDB Journal* **9**: 231–246, 2000.

F. BOUQUET AND P. JÉGOU. Using OBDDs to handle dynamic constraints. *Information Processing Letters* **67**(1):111–120, 1997.

F. BRGLEZ AND H. FUJIWARA. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proceedings of International Symposium on Circuits and Systems*, pages 663–698, 1985.

R. E. BRYANT. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* **C-35**(8):677–691, 1986.

E. F. CODD. A relational model of data for large shared data banks. *Communications of the ACM* **13**(6):377–387, 1970.

M. C. COOPER. An optimal *k*-consistency algorithm. *Artificial Intelligence* **41**(1):89–95, 1989.

T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

G. B. DANTZIG. *Linear Programming and Extensions*. Princeton University Press, 1963.

J. DE KLEER AND G. J. SUSSMAN. Propagation of constraints applied to circuit synthesis. *Circuit Theory and Applications* **8**:127–144, 1980.

R. DECHTER. Bucket elimination: a unifying framework for processing hard and soft constraints. *ACM Computing Surveys* **28A**(4), 1996.

R. DECHTER AND A. DECHTER. Belief maintenance in dynamic constraint networks. In *Proceedings of the 7th Annual Conference of the American Association of Artificial Intelligence*, pages 37–42. AAAI Press/MIT Press, 1988.

R. DECHTER AND J. PEARL. Tree clustering for constraint networks. *Artificial Intelligence* **38**(3):353–366, 1989.

Y. EL FATTAH AND R. DECHTER. An evaluation of structural parameters for probabilistic reasoning: Results on benchmark circuits. In *Proceedings of the 12th Annual Conference on Uncertainty in Artificial Intelligence*, pages 244–251. Morgan Kaufmann, 1996.

C. ERBAS, S. SARKESHIK, AND M. M. TANIK. Different perspectives of the n-queens problem. In *Proceedings of the 1992 ACM Annual Conference on Communications*, pages 99–108. ACM Press, 1992.

F. FRAYMAN. User-interaction requirements and its implications for efficient implementations of interactive constraint satisfaction systems. In *Working Notes of the 1st International Workshop on User-Interaction in Constraint Satisfaction*, pages 31–41, 2001.

E. C. FREUDER. Synthesizing constraint expressions. *Communications of the ACM* **21**(11):958–966, 1978.

E. C. FREUDER. A sufficient condition for backtrack-free search. *Journal of the ACM* **29**(1):24–32, 1982.

E. C. FREUDER. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 227–233. AAAI Press/MIT Press, 1991.

M. R. GAREY AND D. S. JOHNSON. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

J. GASCHNIG. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79124, Carnegie Mellon University, 1979.

I. GENT, E. MACINTYRE, P. PROSSER, P. SHAW, AND T. WALSH. The constrainedness of arc consistency. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, pages 327–340. Springer-Verlag, 1997.

A. GEORGE AND J. LIU. The evolution of the minimum degree ordering algorithm. *SIAM Review* **31**(1):1–19, 1989.

G. GOTTLOB, N. LEONE, AND F. SCARCELLO. Hypertree decompositions and tractable queries. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems*, pages 21–32. ACM Press, 1999a.

G. GOTTLOB, N. LEONE, AND F. SCARCELLO. On tractable queries and constraints. In *Database and Expert Systems Applications, 10th International Conference*, volume 1677 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 1999b.

G. GOTTLOB, N. LEONE, AND F. SCARCELLO. A comparison of structural CSP decomposition methods. *Artificial Intelligence* **124**(2):243–282, 2000.

M. GYSSENS, P. JEAVONS, AND D. COHEN. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence* **66**(1): 57–89, 1994.

T. HAGERUP. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

T. HAGERUP, P. B. MILTERSEN, AND R. PAGH. Deterministic dictionaries. *Jorunal of Algorithms* **41**(1):69–85, 2001.

C. HAN AND C. LEE. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence* **36**(1):125–130, 1988.

C. A. R. HOARE. *Essays in Computing Science*. Prentice-Hall, 1989.

P. D. HUBBE AND E. C. FREUDER. An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 421–427. AAAI Press/MIT Press, 1992.

J. KATAJAINEN AND J. N. MADSEN. Performance tuning an algorithm for compressing relational tables. In *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, volume 2368 of *Lecture Notes in Computer Science*, pages 398–407. Springer-Verlag, 2002.

V. KLEE AND G. J. MINTY. How good is the simplex algorithm? In *Inequalities III. Proceedings of the 3rd Symposium on Inequalities*, pages 159–175. Academic Press, 1972.

G. KONDRAK AND P. VAN BEEK. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence* **89**:365–387, 1997.

V. KUMAR. Algorithms for constraints satisfaction problems: A survey. *AI Magazine* **13**(1):32–44, 1992.

A. K. MACKWORTH. On reading sketch maps. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 598–606. William Kaufmann, 1977a.

A. K. MACKWORTH. Consistency in networks of relations. *Artificial Intelligence* **8**(1):99–118, 1977b.

J. N. MADSEN. Algorithms for compressing and joining relations. CPH STL Report 2002-1, Department of Computing, University of Copenhagen, 2002. Available at `http://www.cphstl.dk`.

D. MAIER. *The Theory of Relational Databases*. Computer Science Press, 1983.

J. MCHUGH. Holy cow, no one's done this! *Forbes* **57**(11):122–128, 1996.

W. MENZEL. Constraint satisfaction for robust parsing of spoken language. *Journal of Experimental and Theoretical Artificial Intelligence* **10**(1): 77–89, 1998.

N. C. MEYERS. Traits: A new and useful template technique. *C++ Report* **7**, 1995. Available at http://www.cantrip.org/traits.html.

I. MIGUEL AND Q. SHEN. Hard, flexible and dynamic constraint satisfaction. *Knowledge Engineering Review* **14**(3):199–220, 1999.

I. MIGUEL AND Q. SHEN. Solution techniques for constraint satisfaction problems: foundations. *Artificial Intelligence Review* **15**(4):243–267, 2001a.

I. MIGUEL AND Q. SHEN. Solution techniques for constraint satisfaction problems: advanced approaches. *Artificial Intelligence Review* **15**(4):269–293, 2001b.

S. MITTAL AND B. FALKENHAINER. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32. AAAI Press/The MIT Press, 1990.

R. MOHR AND T. C. HENDERSON. Arc and path consistency revisited. *Artificial Intelligence* **28**(2):225–233, 1986.

G. L. MØLLER. *On the Technology of Array-Based Logic*. Ph. D. thesis, Technical University of Denmark, 1995. Available at http://www.arraytechnology.com/documents/lic.pdf.

U. MONTANARI. Network of constraints: Fundamental properties and applications to picture processing. *Information Sciences* **7**:95–132, 1974.

T. MORE, JR. Axioms and theorems for a theory of arrays. *IBM Journal of Research and Development* **17**(2):135–175, 1973.

J. NIELSEN. *Usability Engineering*. Morgan Kaufmann, 1994.

J. PEARSON AND P. JEAVONS. A survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, Royal Holloway University of London, 1997.

N. ROBERTSON AND P. D. SEYMOUR. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms* **7**(3):309–322, 1986.

D. SABIN AND E. C. FREUDER. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 10–20. Springer-Verlag, 1994.

D. SABIN AND E. C. FREUDER. Configuration as composite constraint satisfaction. In *Proceedings of the 1st Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996.

T. SCHIEX. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proceedings of the 8th Conference on Uncertainty in Artificial Intelligence*, pages 268–275. Morgan Kaufmann, 1992.

J. G. SIEK, L.-Q. LEE, AND A. LUMSDAINE. *Boost Graph Library*. Addison-Wesley, 2001.

M.-C. SILAGHI, D. SAM-HAROUD, AND B. FALTINGS. Ways of maintaining arc consistency in search using the cartesian representation. In *Proceedings of ERCIM'99*, Lecture Notes in Artificial Intelligence, pages 173–187. Springer-Verlag, 1999.

M. SINGH. Path consistency revisited. *International Journal on Artificial Intelligence Tools* **5**:127–141, 1996.

T. SOININEN AND I. NIEMELÄ. Developing a declarative rule language for applications in product configuration. In *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1999.

T. SOININEN, I. NIEMELÄ, J. TIIHONEN, AND R. SULONEN. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposiom on Answer Set Programming*, pages 195–201. AAAI Press, 2001.

M. STEINBRUNN, G. MOERKOTTE, AND A. KEMPER. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal* **6**(3): 191–208, 1997.

K. SYCARA, S. F. ROTH, N. SADEH-KONIECPOL, AND M. S. FOX. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics* **21**(6):1446–1461, 1991.

R. E. TARJAN AND M. YANNAKAKIS. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing* **13**(3):566–579, 1984.

E. TSANG. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

N. R. VEMPATY. Solving constraint satisfaction problems using finite state automata. In *Proceedings of the 10th Annual Conference of the American Association of Artificial Intelligence*, pages 453–458, 1992.

R. J. WALLACE. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 239–245. Morgan Kaufmann, 1993.

R. WEIGEL AND B. FALTINGS. Compiling constraint satisfaction problems. *Artificial Intelligence* **115**(2):257–287, 1999.

K. WEIHE. Covering trains by stations or the power of data reduction. In *Online Proceedings of the 1st Workshop on Algorithms and Experiments*, 1998. Available at http://rtm.science.unitn.it/alex98/proceedings.html.

M. YANNAKAKIS. Computing the minimum fill-in is NP-complete. *SIAM Journal of Algebraic and Discrete Methods* **2**:77–79, 1981.

B. YU AND H. J. SKOVGAARD. A configuration tool to increase product competitiveness. *IEEE Intelligent Systems* **13**(4):34–41, 1998.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS