Constraints, Varieties, and Algorithms

M.R.C. van Dongen
Department of Computer Science
National University of Ireland, Cork
Western Road, Cork
Ireland

June, 2002

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not been submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced.

M.R.C. van Dongen
June 2002.

Acknowledgements

Working on this thesis was fun but without the help of some people this work would not have been possible. I should like to thank Pat Fitzpatrick for his final and Jim Bowen for his initial supervision. I have greatly benefited from their comments and input. Also, I should like to thank Jim for getting me to Ireland and for allowing me to spend a lot of time on trying out new ideas. Without him I would not have had the opportunity to start this work.

Special thanks should also go to my External Supervisors Peter Jeavons and Dave Cohen for providing excellent comments about the thesis and for making the “viva” an enjoyable event.

Thanks to Pádraic, Neil, and Ruth for keeping up the morale and for interesting discussions. Finally, I should like to thank Olly for his friendship and hospitality, especially in the two months before finalising this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scope of this Thesis

The idea to program with *constraints* for special purposes has been around since the 1960-s. This has resulted in the notion of a *constraint satisfaction problem* (CSP). CSPs can be used to specify, represent, and solve many problems occurring in academia and the "real world."

Gröbner Basis Theory originates from the 1960-s from Bruno Buchberger's PhD dissertation.[1] Gröbner Basis Theory provides algorithms which generalise the Gaussian Elimination Algorithm and the Euclidean Algorithm in the sense that the Gröbner Basis Algorithms remain valid if polynomials "become" multivariate and non-linear.

In this thesis we are concerned with algorithms to solve problems occurring in the areas of Constraint Satisfaction and Gröbner Basis Theory. We borrow ideas from Geometry and Ideal Theory in general and from Gröbner Basis Theory in particular. As part of our presentation we shall point out relationships between notions in Geometry and Gröbner Basis Theory on the one hand and notions in Constraint Satisfaction Theory on the other.

The main contributions of this thesis are as follows:

- Varieties in Geometry are solutions of systems of simultaneous polynomial equations. We shall point out an important relationship between constraints and varieties: Finite constraints are in essence varieties. This relationship opens the door for the application of algorithms from Gröbner Basis Theory to problems occurring in Constraint Satisfaction. It also allows for the integration of discrete and continuous constraints.

- We shall present an elegant algorithm to compute CSPs in directionally solved form with respect to a variable ordering and to compute CSPs which are in globally solved form with respect to all variable orderings. The algorithm relies heavily on the relationship between constraints and varieties, the relationship between varieties and *vanishing* ideals and, the relationship between elimination ideals and Gröbner bases with respect to lexicographical term orders.

[1]This is not completely true. The theory was established in the 1960-s, whereas the name Gröbner Basis was only adopted by Buchberger in the 1970-s as a tribute to his PhD supervisor Wolfgang Gröbner.

- We shall present a novel arc-consistency algorithm for binary CSPs which uses a simple *double-support heuristic* which can improve any existing arc-consistency algorithm. The key insight is that in order to minimise the total number of consistency-checks it is necessary to maximise the number of consistency-checks which find (new) support for two values at a time. At *domain-level* (i.e. after the selection of a constraint that will be used for the next consistency-check) our heuristic selects a consistency-check which, if successful, will increase the number of supported values by as much as possible. We present experimental results which suggest that the heuristic is, indeed, efficient.

  Until recently, it was not known that heuristics which operate at domain-level can have influence on the performance of arc-consistency algorithms which do not repeat consistency-checks. These results are the first results which demonstrate that heuristics which operate at domain-level can have a significant influence on the performance of arc-consistency algorithms.

- We shall study the average time-complexity of two arc-consistency algorithms which only differ in their *domain-heuristics* for the case where there are only two variables in the CSP. To the best of our knowledge these are the first average time-complexity results for arc-consistency algorithms to appear in the constraint literature. The heuristics under investigation are the double-support heuristic and the lexicographical heuristic. The lexicographical heuristic is used for the implementation of most arc-consistency algorithms. Not only do our results clearly indicate that the double-support heuristic is superior to the lexicographical heuristic but also indicate that it is efficient in the sense that, should heuristics exist which are better, then the use of these heuristics can only lead to "marginal" improvements.

  Our average time-complexity results are numbers which can be used to compare the performance of two algorithms. Besides these numbers, we shall also provide good reasons to explain *why* the double-support heuristic is better.

  Finally, we shall provide reasons which justify our choice to study the average time-complexity of domain-heuristics for two-variable CSPs.

- We shall present a generalisation of the well known chronological backtracking algorithm. Our algorithm is a generalisation in the sense that it can use *any* kind of constraint (as opposed to just unary constraints—the domains of the variables) to decompose a problem into smaller problems. Our choice will never increase the local branching factor of the search tree but can make it smaller. We present experimental results of the application of this algorithm to some large problems known from the literature. The results will demonstrate that the generalised backtracking algorithm is promising.

1.2 Outline of this Thesis

The outline of the remainder of this thesis is as follows. In Chapter 2, we shall lay out the main concepts and results of Constraint Satisfaction Theory. Chapter 3 will provide an introduction to

Gröbner Basis Theory. In Chapter 4 we shall present our algorithm to compute CSPs in directionally and globally solved form. In Chapter 5 we shall discuss the "geometry" of constraints and shall present our generalised backtracking algorithm. In Chapter 6 we shall present the double-support heuristic and experimental results which indicate that the heuristic can greatly improve existing arc-consistency algorithms. The average time-complexity results will be presented in Chapter 7. In Chapter 8 we shall present our conclusion and a discussion for future research.

Chapter 2

Constraints

2.1 Introduction to Constraints

2.1.1 Introduction

Constraints are ubiquitous in mathematics, in computer science, and in the "real world." They provide a convenient framework for the description, representation and solution of many problems. In this chapter we shall study constraints.

Before providing a detailed description of constraints and their different disguises, we shall provide a global background to the constraint paradigm. This will be done in the following sections. They provide a rough taxonomy for the different usages of constraints. In Section 2.1.2 we shall provide a brief introduction to *constraint satisfaction problems*. In Section 2.1.3 we shall describe the main characteristics of *constraint logic programming*. The use of *constraints in mathematics* will be covered in Section 2.1.4. A more detailed presentation, one for each of the different usages of constraints, can be found in Section 2.2, Section 2.3, and Section 2.4.

2.1.2 Constraint Satisfaction Problems

In this section we shall provide a short introduction to *constraint satisfaction problems*. As promised in the introduction we shall also study constraint satisfaction problems in greater detail. This will be done in Section 2.2 where we shall also study search.

Constraint satisfaction problems are used to specify, represent, and solve many interesting problems involving variables, the domains from which the variables can take their values, and finally, relations between subsets of the variables. The introduction of the constraint paradigm has resulted in an enormous diversity of approaches to the solution of a cornucopia of problems.

For the moment it suffices to know that the main characteristics of (finite) constraint satisfaction problems are given by the following list:

- A finite Constraint Satisfaction Problem (CSP) constitutes a set of variables, the domains of the variables, and constraints between subsets of the variables.

- The domains of the variables are finite and are known in advance.

- A *constraint* between a subset S of the variables of the CSP is a subset of the Cartesian product of the domains of the variables in S. The membership problem of each of the constraints is decidable as well as tractable. The intuition is that constraints correspond to relations between subsets of the variables thereby limiting the "assignments" to the variables that are "allowed." It can be decided if an assignment is allowed by carrying out constraint membership tests.

Constraint satisfaction theory will be studied in more detail in Section 2.2.

A *constraint programming language* is a programming language which has a *constraint component* and a *programming component*. The *constraint component* (more commonly referred to as the *constraint-store*) is used to represent the variables, their domains and the constraints. In addition it provides algorithms to implement strategies to be used before, during, or after search. The *programming component* provides a language to define constraint satisfaction problems and, if the language implements this, annotations to select algorithms to implement certain strategies during certain stages in the process of solving a problem or transforming it to an *equivalent* problem. Here, a problem is equivalent to another problem if their solutions are the same.

In the following we shall distinguish between different kinds of constraint programming languages. Constraint programming languages which only deal with constraint satisfaction programs will be called *constraint satisfaction programming languages*.

2.1.3 Constraint Logic Programming

Constraint Logic Programming (CLP) languages are logic programming languages enriched with constraints. This section provides a short introduction to constraint logic programming. A more detailed description will be provided in Section 2.3.

Most of the differences between constraint satisfaction problems and constraint logic programming originate from constraint logic programming's roots in logic programming. For example, in constraint logic programming—as opposed to constraint satisfaction programming—the domains of the variables are not always explicitly available. Instead, user-defined predicates can also indirectly determine the set of values from which the variables can take their values.

Constraints in constraint logic programming are used both as input for and output of queries. For example, the query

```
1 <= X, X <= 1
```

when posed in the constraint logic programming language CLP($\mathcal{R}$-Lin), will lead to the answer `X = 1`. This may *seem* to be a solution of the inequalities but *is* an output constraint, namely the constraint which only allows `X` to be `1`. Because the answer given by CLP($\mathcal{R}$-Lin) is not `No` it is understood that the original query is satisfiable and that the returned answer is its equivalent.

Constraint logic programming languages are, in essence, logic programming languages where unification has been replaced by constraint satisfaction, i.e. the unification algorithm used in logic programming has been replaced by an algorithm to decide constraint satisfaction.

2.1.4 Constraints in Mathematics

Another usage of constraints arises in mathematics where certain classes of problems include variables which have to satisfy certain kinds of equations, inequalities or inequations.[1] The problems are formulae that are universally and/or existentially quantified and contain disjunctive and conjunctive operators and equations, inequations and inequalities. These kinds of constraints have been studied by mathematicians for centuries and many kinds of solution techniques have been developed to deal with them.

2.1.5 Outline

In the remainder of this chapter we shall review in greater detail constraint satisfaction, search, constraint logic programming, and the use of constraints in mathematics. Due to the nature of the remainder of this thesis, the emphasis will be on constraint satisfaction and search. Constraint satisfaction problems and search will be discussed in Section 2.2. We shall discuss the constraint logic programming paradigm in Section 2.3. We shall not refer to constraint logic programming in subsequent chapters. Section 2.3 has been included solely for the purposes of comparison and completeness. In Section 2.4 we shall study selected branches in mathematics which deal with constraints. We shall present a summary in Section 2.5.

In the following it will be assumed that the reader is familiar with the notions of logic programming and (chronological) backtracking. Readers not familiar with logic programming may wish to consult [Nilsson and Maluszynski, 1989; Apt, 1997; Bratko, 1986] or any other book on logic programming or PROLOG. Readers not familiar with backtracking may wish to consult [Kondrak and van Beek, 1995; 1997; Nadel, 1989; Dechter and Frost, 1999] or [Ginsberg, 1993; Tsang, 1993].

2.2 Constraint Satisfaction and Search

2.2.1 A Historical Background

The purpose of Section 2.2 is to provide a constraint satisfaction vocabulary and an understanding of how the use of *consistency algorithms* can improve backtrack search. In this section we shall provide an introduction to constraint satisfaction and search from a historical perspective. At the end of this section we shall outline the organisation of the remainder of Section 2.2.

Backtrack Search and Thrashing

There are several methods to find solutions of problems involving constraints and variables the domains of which are finite and are known in advance. The most commonly used method is *chronological backtrack search* [Golomb and Baumert, 1965]. Backtrack search belongs to the

[1] Here and in the remainder of this thesis, equalities are formulae of the form $p = q$, strict inequalities are formulae of the form $p < q$ or $p > q$, inequalities are formulae of the form $p \leq q$ or $p \geq q$, and inequations are formulae of the form $p \neq q$.

generate-and-test family of solution methods.[2] To solve problems, members of this family use a strategy to start with an empty partial solution and to recursively generate, i.e. extend, partial solutions and to carry out tests to see which of these (partial) candidate solutions satisfy the relevant constraints. The (partial) solutions which violate one or more constraint are omitted because they cannot contribute to the solution set.

Chronological backtrack search suffers from a short-term memory—total amnesia, really—and due to the nature of the generate-and-test approach it repeatedly has to rediscover that certain combinations of values are not compatible. This is what is called *thrashing* [Mackworth, 1977]. Due to the size of the search-space, thrashing almost always results in the enumeration of an enormous number of candidate solutions which will never be part of a solution, thus resulting in very longe execution times.

Consistency

The cause of thrashing is that a problem which is formulated as a *constraint satisfaction problem* usually has a relatively low level of *consistency*. Roughly speaking, one problem formulation is less consistent than another if it leaves more room for partial solutions which do not violate any of the constraints between subsets of the variables, but which will nevertheless not participate in any global solution. A formal definition of the notion of consistency will be provided in Section 2.2.4.

Consistency Algorithms as Preprocessors

Researchers who used chronological backtracking soon discovered that making constraint satisfaction problems more consistent is possible at relatively low cost and generally significantly improves the performance of backtrack search. After this discovery many researchers started to use *consistency algorithms* as preprocessors to backtracking, i.e. they transformed constraint satisfaction problems into equivalent constraint satisfaction problems which were guaranteed to have a certain level of consistency (usually higher than the original level of consistency) and then they applied backtracking [Mackworth, 1977; Gaschnig, 1978; Mackworth and Freuder, 1985; 1993; Mohr and Henderson, 1986; Dechter and Dechter, 1987].

Unfortunately, to make problems more consistent *before* search does not always prevent thrashing. Incompatible combinations of values still have to be rediscovered several times during search, albeit at a later stage.

Consistency Maintenance During Search

Early methods to try to overcome the problem that thrashing still occurs after the application of consistency algorithms (before search) *maintained* certain levels of *directional consistency* during search, i.e. they maintained consistency levels during search which depend on the "direction" of search (the ordering on the variables). An early reference is [Haralick and Elliott, 1980] which considers directional consistency methods for the first time. The term di-

[2] Perhaps extend-and-test would have been a better name.

rectional consistency is due to Dechter [Dechter, 1990a] (see also [Dechter and Pearl, 1988a; 1988b]). Dechter's application of directional consistency methods is to *cutset decomposition*, where one tries to "transform" the original constraint satisfaction problem into a constraint satisfaction problem that does not contain "circular dependencies" [Dechter and Meiri, 1989]. These problems can be shown to be solved without much effort [Freuder, 1982].

Note that it is possible—at least in principle—to make a problem completely consistent before the application of search. In general, this is not the preferred approach because the combined costs of making the problem entirely consistent and search will normally outweigh the total costs of search without applying consistency algorithms. Furthermore, to make a problem completely consistent usually involves the introduction of additional constraints, the majority of which involve many variables and are therefore usually very large.

At the moment this thesis was written it was a trend to maintain certain low levels of consistency during search which were independent of the direction of search [Sabin and Freuder, 1994; Bessière *et al.*, 1995; Sabin and Freuder, 1997].

Organisation

The organisation of the remainder of Section 2.2 is as follows. In Section 2.2.2 we shall provide definitions of *constraints*, and *constraint satisfaction problems*. In Section 2.2.3 we shall discuss chronological backtrack search and some of the problems that arise with this algorithm. We shall study the concepts of *satisfiability* and *consistency* in Section 2.2.4. In Section 2.2.5 we shall provide evidence that to maintain a certain level of consistency during search can be worthwhile. In Section 2.2.6 we shall provide a summary.

The reader who is interested in a more detailed introduction to constraints may wish to consult [Dechter, 1992; Tsang, 1993; Marriot and Stuckey, 1998; Früwirth and Abdonnaher, 1997; Smith, 1995].

2.2.2 Constraints

This section is an introduction to *constraints*, *constraint satisfaction problems*, the notion of *satisfiability*, and other notions that will allow us to reason about (properties of) constraint satisfaction problems.

Constraints

In the following, let n be a positive integer, let $X = \{ x_1, \ldots, x_n \}$ be a set of variables, and let $\cdot <_{\text{lex}} \cdot$ be the "usual" lexicographical ordering on the variables in X. It is assumed that $x_i <_{\text{lex}} x_j$, if and only if $i < j$, for $1 \leq i, j \leq n$. Associated with each variable is its *domain*. The domain of x_i is denoted $D(x_i)$.

Definition 2.1 (Constraint). Let $S = \{ x_{i_1}, \ldots, x_{i_m} \}$ be a non-empty set of variables. A set C_S is called a *constraint* between the variables of S if

$$C_S \subseteq \bigtimes_{j=1}^{m} D(x_{i_j}),$$

where $\cdot \times \cdot$ denotes the Cartesian product operator.

Definition 2.2 (Arity). The *arity* of a constraint C_S is given by the cardinality of S. A constraint whose arity is m is called an m-*ary* constraint.

A constraint is called *unary* if its arity is one, *binary* if its arity is two, and *ternary* if its arity is three. A constraint the arity of which is higher than two is usually referred to as a *higher order* constraint.

A constraint C_S contains those and only those tuples that represent the "assignments" to the variables in S that are allowed by the constraint. The notion of a tuple being allowed by a constraint will be formalised in the following paragraphs with the introduction of the notion of *constraint satisfaction*.

Constraint Satisfaction

In the following, tuples will play the role of "assignments" to variables. Let $S = \{ x_{i_1}, \ldots, x_{i_m} \}$ be a non-empty set of variables. An S-tuple (or $(x_{i_1}, \ldots, x_{i_m})$-tuple) is understood to be an "assignment" to $(x_{i_1}, \ldots, x_{i_m})$. An S-tuple $(v_{i_1}, \ldots, v_{i_m})$ is the simultaneous "assignment" $x_{i_1} = v_{i_1}, \ldots, x_{i_m} = v_{i_m}$. It will be implicitly assumed that if $(v_{i_1}, \ldots, v_{i_m})$ is an S-tuple then the indices $i_1, \ldots, i_m$ comply with the ordering $\cdot <_{\text{lex}} \cdot$ on the variables $x_{i_1}, \ldots, x_{i_m}$. If x_i is a variable then $\{ x_i \}$-tuples will also be referred to as x_i-tuples.

Definition 2.3 (Constraint Satisfaction). Let S and T be non-empty sets of variables, and let C_S be a constraint between the variables in S. A T-tuple t is said to *satisfy* C_S if either $S \not\subseteq T$ or $S \subseteq T$ and the projection of t onto S is a member of C_S.

Definition 2.4 (Consistency-Check). Let C_S be a constraint and let s be an S-tuple. A test of the form $s \in C_S$ is called a *consistency-check*.

Consistency-checks are decidable. Usually, consistency-checks are understood to be tractable. Standard measures for the complexity of constraint satisfaction problems are the average and the worst number of consistency-checks needed to solve that problem. In the remainder of this thesis, we shall—for reasons which will become apparent—sometimes use *support-check* instead of consistency-check.

Constraint Satisfaction Problems

Definition 2.5 (Constraint Satisfaction Problem). A *Constraint Satisfaction Problem* (CSP) is a triple of the form (X, D, C), where X is a set of variables, D is a function that maps each variable in X to its domain, and C is a set containing constraints on subsets of the variables in X.

Note that without loss of generality we may assume that if (X, D, C) is a CSP then $X = \cup_{C_S \in C} S$ because we can always add the unary constraint $C_{\{ x \}} = D(x)$ for every $x \in X$ for which there is no unary constraint. From now on we shall assume that every constraint satisfaction problem (X, D, C) is such that $X = \cup_{C_S \in C} S$.

In the following, a CSP will be called *finite* if the cardinalities of the domains of the variables of the CSP are finite.

Associated with every CSP is its *constraint graph*. Constraint graphs are hyper-graphs. For every variable of the CSP there is a vertex in the graph and vice versa. For every constraint C_S of the CSP there is an edge E_S in the constraint graph between the vertices corresponding to the variables in S and vice versa.

The maximum domain size of the variables of a CSP is usually denoted d in the constraint literature. The number of edges in the constraint graph of a CSP is usually denoted e. A CSP whose constraints all have an arity of two or less is called a *binary* CSP.

Definition 2.6 (Satisfiability of CSP). Let $(\, X, D, C \,)$ be a CSP, and let Z be a set of variables such that $X \subseteq Z$. A Z-tuple is said to *satisfy* the CSP if it satisfies each of the constraints in C.

A CSP is called *satisfiable* if there is a tuple which satisfies the CSP and *unsatisfiable* otherwise.

The *micro-structure* of a binary CSP is a detailed way of depicting a CSP [Freuder, 1993]. Figure 2.1 depicts the micro-structure of the constraint satisfaction problem $(\, X, D, C \,)$, where $X = \{\, x, y \,\}$, $D(x) = \{\, 0, 1 \,\}$, $D(y) = \{\, 0, 1 \,\}$, $C = \{\, C_{\{\,x\,\}}, C_{\{\,y\,\}}, C_{\{\,x,y\,\}} \,\}$, $C_{\{\,x\,\}} = \{\, 0, 1 \,\}$, $C_{\{\,y\,\}} = \{\, 0 \,\}$ and $C_{\{\,x,y\,\}} = \{\, (\,0, 0\,), (\,0, 1\,), (\,1, 0\,) \,\}$. The domains of the variables are rep-

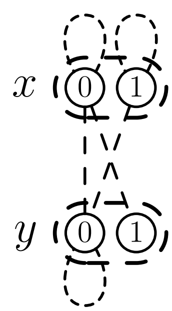

Figure 2.1: Micro-structure of CSP.

resented by the dashed oval-shaped structures. The values in the domains of the variables are represented by the circles in the domains. The binary constraints are represented by the edges between a value in the domain of one variable and a value in the domain of another variable. The unary constraints are represented by the remaining edges. An S-tuple is in a constraint if and only if its values are connected by an edge between the variables in S.

From now on unary constraints will be omitted from micro-structures of CSPs. It will be implicitly assumed that each value in the domain of each variable satisfies the unary constraint on that variable.

Definition 2.7 (Solution). Let $\mathcal{C} = (\, X, D, C \,)$ be a CSP. An X-tuple that satisfies $\mathcal{C}$ is called a *solution* of $\mathcal{C}$.

Definition 2.8 (Equivalence of CSPs). Two CSPs $(\, X, D_1, C_1 \,)$ and $(\, X, D_2, C_2 \,)$ are called *equivalent* if and only if their solutions are the same.

2.2.3 Chronological Backtrack Search

In this section we briefly recall some properties of the chronological backtrack search algorithm to solve CSPs and discuss some of the problems which occur with this algorithm.

Backtracking Algorithms

Constraint satisfaction problems whose variables all have domains of finite cardinality are frequently solved using the chronological backtracking algorithm. Variants of this algorithm range from:

- a plain version [Golomb and Baumert, 1965] with a predefined variable and value ordering; to
- forward checking [Haralick and Elliott, 1980] and MAC [Sabin and Freuder, 1994] which try to prevent errors by maintaining certain levels of local consistency; and to
- backjumping [Gaschnig, 1978], conflict directed backjumping [Prosser, 1993] and dynamic backtracking [Ginsberg and McAllester, 1994] which try to prevent local mistakes by locating "the" cause of previous errors.

For surveys of backtracking the reader may wish to consult [Kondrak and van Beek, 1995; 1997; Nadel, 1989; Dechter and Frost, 1999]. Basic properties of backtracking can be found in [Ginsberg, 1993; Tsang, 1993].

An algorithm to solve problem $\mathcal{P}$ is called a *sound* algorithm to solve $\mathcal{P}$ if, whenever it returns $\mathcal{S}$, then $\mathcal{S}$ is a solution of $\mathcal{P}$. An algorithm to solve problem $\mathcal{P}$ is called a *complete* algorithm to solve $\mathcal{P}$ if it returns all solutions of $\mathcal{P}$. Backtracking is sound and complete.

Search Trees

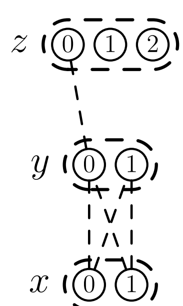

Figure 2.2: Micro-structure of a three-variable CSP.

Consider the constraint satisfaction problem (X, D, C), where $X = \{x, y, z\}$, $D(x) = D(y) = \{0, 1\}$, $D(z) = \{0, 1, 2\}$, $C_{\{x,y\}} = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ and $C_{\{y,z\}} = \{(0, 0)\}$. The micro-structure of the CSP is depicted in Figure 2.2.

A chronological backtracker which uses the (standard) lexicographical ordering heuristics on the variables and the values in their domains to solve the CSP, will visit the nodes in the search-tree depicted in Figure 2.3 using an in-order traversal. The subscripts to the right of the nodes indicate the visiting order of the nodes.

The shortest paths from the root to the leaves of the tree that are printed in bold face correspond to the solutions of the CSP. The backtracker starts off by making x the current variable and by setting its value to 0. It will then try to find which combinations of y and z are compatible with the current choice for x. It will need six consistency-checks to do this—one check for each of the six members in the Cartesian product of the domains of y and z.

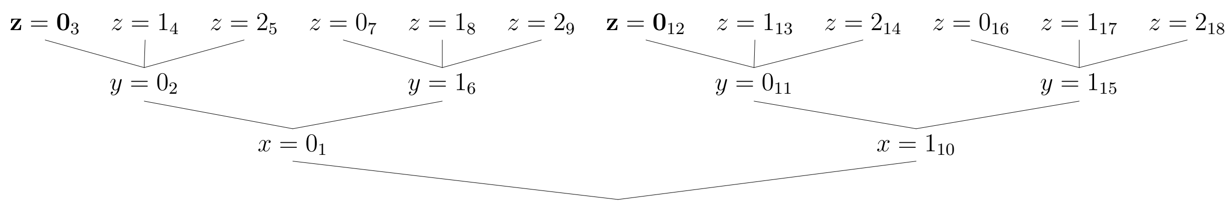

Figure 2.3: Chronological backtrack search tree.

To complete the search the backtracker will assign 1 to x and will try to find out which combinations for y and z are compatible with this new assignment to x. The consistency-checks it requires to find these compatible assignments are $(\,1, v\,) \in C_{\{\,x,y\,\}}$ and $(\,v, w\,) \in C_{\{\,y,z\,\}}$, for $0 \leq v \leq 1$, and $0 \leq w \leq 2$. The consistency-checks of the form $(\,v, w\,) \in C_{\{\,y,z\,\}}$ are exactly the same as the ones that were needed to find the values that were compatible with the assignment $x = 0$. In total it will need twelve consistency-checks on $C_{\{\,y,z\,\}}$ alone.

Chronological backtracking cannot remember any consistency-check it has carried out before. This is one of its greatest deficiencies.

Note that each of the consistency-checks that failed is a consequence of the fact that some of the members of the domain of y and z do not participate in any of the solutions. That some of these members are not part of any solution can be discovered by examining the constraint $C_{\{\,y,z\,\}}$. The removal of these values, 1 from $C_{\{\,y\,\}}$, and 1 and 2 from $C_{\{\,z\,\}}$, *before* the search had started would have prevented all consistency-checks that failed.

An important lesson can be learned from this example because problems like the one discussed here can be part of larger problems and may have to be solved over and over again. To remove values from the domains of the variables which cannot contribute to solutions of such sub-problems will not change the solutions of the whole problem but will improve the search because each time the sub-problem has to be solved, the values do not have to be taken into account.

2.2.4 Consistency

In this section we shall study the notion of *consistency* in constraint satisfaction, certain properties of CSPs that have a certain level of consistency, and the use of consistency as a preprocessor of CSPs to improve search.

One of the innovating contributions of constraint satisfaction to artificial intelligence is the notion of consistency. Early studies on consistency focussed on special cases of consistency [Montanari, 1974] (see also [Montanari and Rossi, 1988]). A standard reference for special cases of consistency is also [Mackworth, 1977]. The special cases of consistency referred to before can all be captured as instances of the more general notion of k-consistency of CSPs [Freuder, 1978]. The following paragraphs provide a definition of *k-consistency*.

Before we can define k-consistency we need the notion of *k-satisfiability*.

Definition 2.9 (k-Satisfiability). Let $(\, X, D, C\,)$ be a CSP and let k be a positive integer. A T-tuple is said to *k-satisfy* the CSP if it satisfies each of the constraints $C_S \in C$ for which $S \subseteq T$ and for which $|S| \leq k$.

A CSP is called k-satisfiable if for every subset T of X whose cardinality is k there is a T-tuple which k-satisfies the CSP.

Definition 2.10 (k-Consistency). Let X be a set containing $n > 0$ variables, let $(\, X, D, C\,)$ be a CSP, and let $2 \leq k \leq n$ be an integer. The CSP is *1-consistent* if for every $x \in X$ it holds that $D(x) = C_{\{\,x\,\}} \neq \emptyset$. The CSP is *$k$-consistent* if for every S-tuple of the form $(\, v_{i_1}, \ldots, v_{i_{k-1}}\,)$ which $(k-1)$-satisfies the CSP it holds that for every $T \in \{\, S \cup \{\,x\,\} \,:\, x \in X \setminus S\,\}$ there is a T-tuple $(\, w_{j_1}, \ldots, w_{j_k}\,)$ which k-satisfies the CSP and has the property that $i_p = j_q \implies v_{i_p} = w_{j_q}$, for $1 \leq p < k$ and $1 \leq q \leq k$.

From an intuitive point of view, a CSP is k-consistent if for every S-tuple of cardinality $k-1$ which $(k-1)$-satisfies the CSP it is possible, for each of the remaining variables $x \in X \setminus S$, to find an x-tuple such that the result of extending the S-tuple by the x-tuple is an $(\{\,x\,\} \cup S)$-tuple which k-satisfies the CSP.

Note that this operation of extending S-tuples is exactly what happens all the time during backtrack-search when values are assigned to the current variable and consistency-checks are carried out to decide whether these assignments satisfy the constraints between the current and past variables. CSPs with higher levels of consistency are easier to search in the sense that they usually allow for fewer errors to be made during backtrack search.

Definition 2.11 (Strong k-Consistency). Let k be a positive integer. A CSP is *strongly k-consistent* if it is j-consistent for all $1 \leq j \leq k$.

It is important to note that a CSP involving n variables which is strongly n-consistent is satisfiable.

The following theorem relates the level of *strong* k-consistency of binary CSPs and the maximum domain size of the variables. The reader is referred to [Dechter, 1990b] for proof and further details.

Theorem 2.12 (From Local to Global Consistency). *Let $\mathcal{C} = (\, X, D, C\,)$ be a* CSP *such that the maximum arity of the constraints in C is r, and $\max(\{\,|D(x)| \,:\, x \in X\,\}) = k$. If $\mathcal{C}$ is strongly $(k(r-1)+1)$-consistent, then it is globally consistent. In particular, if $\mathcal{C}$ is binary and strongly $(k+1)$-consistent then it is globally consistent.*

Binary CSPs that are 1-consistent are called *node-consistent*. Binary CSPs that are 2-consistent are called *arc-consistent*. The reasons for this are historical.

The time-complexity of transforming a binary CSP into its arc-consistent equivalent as a function of the maximum domain size d and the number of binary constraints e is $\mathbf{O}(ed^2)$. This result is due to [Mohr and Henderson, 1986] who presented an optimal arc-consistency algorithm called AC-4. Note that it is straightforward to see that it should not be necessary to spend more than ed^2 consistency-checks because there cannot be more than ed^2 such checks and each check can be remembered at the cost of a $\mathbf{O}(ed^2)$ space-complexity. However, the number ed^2 can become quite large and it is a challenge to find algorithms with a lower space complexity than $\mathbf{O}(ed^2)$. Another algorithm to transform a CSP into its arc-consistent equivalent is AC-3 [Mackworth, 1977]. The worst-case time-complexity of AC-3 is $\mathbf{O}(ed^3)$. It was observed in [Wallace, 1993] that AC-3 almost always performed better than AC-4 despite the fact that AC-3 has a worse worst-case time-complexity.

Another optimal arc-consistency algorithm called AC-7 is presented in [Bessière *et al.*, 1995] (See also [Bessière *et al.*, 1999]). This algorithm performs much better than AC-3 and AC-4 on average.

A binary CSP is called *connected* if its constraint graph is connected. A graph that is a finite collection of trees is called a *forest* . Binary CSPs that are arc-consistent and whose constraint graphs are forests can be solved (in the sense of returning a solution if it exists) without backtracking [Freuder, 1982]. These results of [Freuder, 1982] were generalised in [Freuder, 1985].

As already indicated, the use of consistency algorithms to transform CSPs into equivalent problems that are more consistent usually improves search. It takes at most nd consistency-checks to remove from the domains of the variables those values that do not satisfy the unary constraints on those variables. The costs of this are much less than the costs of backtracking whereas the removal of a few values from the domains of one or more variables significantly reduces the size of the search-space, which is of the same order as $\prod_{x \in X} |D(x)|$.

Similarly, making a CSP arc-consistent is a good investment because the overhead required to make the CSP arc-consistent is low in comparison with the costs of search. The removal of even a few values from the domains of one or more variables reduces the number of occasions where local mistakes can be made during search thus reducing the number of candidate solutions that have to be considered. Practical evidence in the form of experiments where node-consistency and arc-consistency algorithms were applied *before* search have supported the claim that the combined costs of consistency and search are less than the costs of searching without the application of these consistency algorithms.

2.2.5 Consistency Maintenance During Search

In this section we shall study the relationship between the maintenance of low levels of consistency during search and the complexity of the subsequent search.

Propagate and Generate

Up to around the 1990s it was commonly agreed upon by the constraint satisfaction community that the application of consistency-algorithms was to make CSPs node-consistent (1-consistent) and arc-consistent (2-consistent) *before* search and that search was to be carried out alone while maintaining a level of consistency which was stronger than node-consistency but not as strong as arc-consistency [Sabin and Freuder, 1994]. See [Nadel, 1987; 1989] for a study of selected backtracking algorithms and a classification of such algorithms according to the *degree* of arc-consistency they maintained. In Nadel's terminology these degrees were numbers which were strictly between 0 and 1.

The agreement by the constraint satisfaction community was questioned and evidence was provided that the Maintenance of *full* Arc-Consistency (MAC) during search was more efficient on average than searching while maintaining lower levels of consistency [Sabin and Freuder, 1994]. This was supported by [Bessière *et al.*, 1995] where results were presented of the application of a MAC-algorithm to problems. With the exception of the Zebra Problem the problems that were solved were beyond the scope of methods which only maintained consistency levels of less than two.

Backtracking algorithms which maintain consistency levels of two and more during search are sometimes referred to as *propagate-and-generate* (as opposed to generate-and-test) because these algorithms use constraint propagation to obtain a certain level of consistency and then generate the next part of the solution.

Justification of MAC

The reason why the maintenance of arc-consistency during search works is similar to the reason why it is good to make problems arc-consistent before search. It is because an "assignment" to the current variable in backtrack search can be viewed as the removal of values from a unary constraint. This is a process by which values in the domains of other variables may directly or indirectly lose support, i.e. it is a process by which problems may lose a certain degree of consistency. The loss of consistency normally leads to more local mistakes during search. A few assignments can lead to large inconsistencies. If the level of consistency is low it is relatively easy to maintain that level of consistency during search compared to the costs of search alone. Experimental results have reconfirmed this several times.

2.2.6 Summary

Constraint satisfaction problems are a good vehicle to specify, represent, and solve certain classes of problems. A frequently used algorithm to solve constraint satisfaction problems is backtracking. One of the problems with backtracking is that it leads to thrashing because backtracking has a short-term memory and has to rediscover facts over and over again.

One of the major contributions of constraint satisfaction theory to artificial intelligence is the notion of consistency. The improvement of backtracking by maintaining certain levels of consistency and exploiting knowledge about the level of consistency of CSPs has brought problems into the realm of feasible tasks that would have remained intractable without these improvements.

2.3 Constraint Logic Programming

2.3.1 Introduction

In this section we shall briefly discuss constraint logic programming (CLP). It is assumed that the reader is familiar with logic programming, unification, and constraint satisfaction. The reader who is not familiar with logic programming may wish to consult [Nilsson and Maluszynski, 1989; Apt, 1997; Bratko, 1986]. The reader not familiar with unification is referred to [Baader and Siekmann, 1993; Siekmann, 1989] or [Lassez *et al.*, 1988]. The reader not familiar with constraint satisfaction may wish to consult Section 2.2 and the references presented therein.

Constraint logic programming is a generalisation of logic programming in the sense that it does not depend on *unification* to decide satisfiability. Instead, it uses constraint satisfaction. Another interesting feature of constraint logic programming is that it allows for constraints to appear both in input and output. To understand why this is more general than the logic programming approach, one may observe that *syntactic equality* is the only way by which things can be unified in logic programming. For example, if $+ : \mathbb{C} \times \mathbb{C} \mapsto \mathbb{C}$ is the addition operator as "usual" (not to be confused with a logic programming functor) and $\cdot = \cdot$ is a syntactically sugared version of the equality relation, then it is impossible to use the built-in logic programming machinery to infer that $\{ x + y = 0, x - y = 0 \}$ entails $x = 0$. Adding constraint satisfaction to logic programming languages enlarges the set of formulae which are provably satisfiable.

The first real constraint logic programming language is PROLOG II which allows equations and inequations over *rational trees*. Successors of PROLOG II are all instances of constraint logic programming languages [Colmerauer, 1984; 1987; 1990].

Another constraint programming language is CLP($\mathcal{R}$-Lin), which is an extension of PROLOG with linear inequalities over the reals [Jaffar and Lassez, 1986; 1987a; 1987b; Jaffar *et al.*, 1993; Heintze *et al.*, 1992]. A generalisation of CLP($\mathcal{R}$-Lin) is RISC(CLP) which is not restricted to linear (in)-equalities [Hong, 1992] (see also [Hong and Ratschan, 1995]).

The interested reader may wish to consult [Jaffar and Maher, 1994] for an excellent survey of constraint logic programming.

The remainder of this section is as follows. Section 2.3.2 discusses CLP($\mathcal{R}$-Lin). RISC(CLP) is discussed in Section 2.3.3. A selection of other constraint logic programming languages is described Section 2.3.4. A summary is provided in Section 2.3.5.

2.3.2 CLP with Linear Inequalities over the Reals

This section discusses some aspects of CLP($\mathcal{R}$-Lin) [Jaffar and Lassez, 1986; 1987a; 1987b; Jaffar *et al.*, 1993; Heintze *et al.*, 1992]. The discussion provides some background about the implementation of CLP($\mathcal{R}$-Lin), the soundness of the implementation, and the differences between the CLP($\mathcal{R}$-Lin) approach and the constraint satisfaction programming approach.

In the following it is assumed without loss of generality that systems of equations and inequalities do not contain equations. This is justified because if a and b are real then $a = b \iff a \leq b \wedge b \leq a$.

CLP($\mathcal{R}$-Lin) is one of the earliest examples of a constraint logic programming language. CLP($\mathcal{R}$-Lin) extends PROLOG with linear inequalities over the reals. The CLP($\mathcal{R}$-Lin)-engine works just like PROLOG's resolution-machinery except for the fact that it can also decide the satisfiability of linear inequalities over the reals. The first phase of Dantzig's *Simplex Algorithm* is used to decide the satisfiability of linear inequalities [Dantzig, 1963] (See also [Schrijver, 1996]). If it turns out that a certain branch in the search tree becomes infeasible due to an inconsistency in the linear inequalities then backtracking takes place. If variables become ground in the process of deciding satisfiability and if inequalities exist containing terms that are not linear, then the values of the ground variables are substituted into the higher order inequalities thereby possibly introducing inequalities that were not entailed by the old ones.

In the process of deciding satisfiability it is of utmost importance to restrict the number of inequalities by removing redundant inequalities [Lassez *et al.*, 1989]. Integrations of the Simplex Algorithm and constraint engines which remove the need to copy the constraint-store are discussed in [Jaakola, 1990] and [Van Hentenryck and Ramachandran, 1994].

The constraint logic programming approach of CLP($\mathcal{R}$-Lin) differs from the constraint satisfaction programming approach in the sense that constraint satisfaction programming maintains consistency of the *whole* problem on a *local* level, whereas CLP($\mathcal{R}$-Lin) maintains consistency of *part* of the problem (linear inequalities) on a *global* level. Both approaches have proved to work.

CLP($\mathcal{R}$-Lin) has been a great success both in and outside academia. Unfortunately, the implementation of CLP($\mathcal{R}$-Lin) is not sound because it depends on the underlying hardware for carrying out floating point operations. Concerns about the efficiency of the implementation have led to a decision to drop soundness. Though the efficiency certainly has contributed a lot to its success, questions should be asked about the applicability of an unsound implementation of CLP($\mathcal{R}$-Lin) as a general constraint programming tool.

2.3.3 CLP over the Reals

In this section we shall discuss RISC(CLP) which was developed at the Research Institute for Symbolic Computation (RISC) in Linz, Austria [Hong, 1992].

RISC(CLP) is another member of the constraint logic programming family. It is a generalisation of CLP($\mathcal{R}$-Lin) in the sense that it can be used to decide the satisfiability of sentences in the *first-order theory of the reals* (FOTR). The first-order theory of the reals roughly consists of quantified conjunctions and disjunctions of equalities, strict inequalities, inequalities, and inequations. It is known since the 1930s that FOTR is decidable [Tarski, 1951]. Tarski also provided a decision algorithm. The complexity of Tarski's method is not optimal and problems in the first-order theory over the reals are generally very difficult. Recent work by Collins and Hong has brought many problems from this theory into the realm of tractability that were intractable with other methods [Collins and Hong, 1991]. Also of interest to the reader may be [Hong, 1991] which contains an interesting comparison of the tractability of methods to decide problems in the first-order theory of the reals. More about the first-order theory of the reals can be found in Section 2.4.2.

The algorithm used to implement RISC(CLP) uses *quantifier elimination* to translate quantified formulae of FOTR to equivalent formulae of FOTR which contain strictly fewer quantified variables. At the heart of the algorithm lies Collins' Cylindrical Algebraic Decomposition (CAD) with improvements by Collins and Hong. The interested reader is referred to [Collins and Hong, 1991; Mishra, 1993] for more information about the CAD-algorithm. RISC(CLP) keeps formulae in FOTR consistent and backtracks as soon as inconsistencies occur.

The RISC(CLP)-engine uses Gröbner bases to simplify the constraint-store [Buchberger and Hong, 1991]. This was shown to speed up the computation.

Unfortunately, the implementation of RISC(CLP) is not available for use outside of RISC. RISC(CLP) is sound. According to Hong it is slow [Hong, 1992].

The difference between the constraint satisfaction programming approach and the constraint logic programming approach of RISC(CLP) is that the former keeps the whole problem partially consistent, whereas the latter keeps a *part* of the problem *globally* consistent. This is exactly what constitutes the difference between constraint satisfaction programming and CLP($\mathcal{R}$-Lin).

2.3.4 Other CLP Dialects

This section discusses some selected members of the family of constraint logic programming languages.

A system closely related to RISC(CLP) is CAL (Contrainte Avec Logique) [Aiba *et al.*, 1988; Sakai and Aiba, 1989; Sakai and Sato, 1990; Aiba and Hasegawa, 1992]. CAL uses Gröbner bases to decide satisfiability of polynomial equations over the field of the complex numbers. The main differences with RISC(CLP) are—of course—the domains of computation and the fact that CAL can only decide satisfiability over equations. Having said that, it should be noted that inequations can be easily added to CAL because (in fields) $a \neq b$ is satisfiable if and only if $c \times (a - b) = 1$ is satisfiable. Like all the constraint logic programming languages discussed before, CAL keeps part of the problem (equations over the complex numbers) globally consistent. CAL is sound.

A constraint logic programming based on interval arithmetic is CLP(BNR) developed at Bell Northern Research [Older and Benhamou, 1993]. CLP(BNR) allows equations over boolean formulae and constraints over integral domains as well as floating point intervals. It uses interval arithmetic to *narrow* the domains of the variables involved in constraints and answers over the reals are returned as intervals which contain all the solutions. Here, an interval is narrowed if it is transformed to a subset of that interval. The change of order of constraints in CLP(BNR) may result in domains the bounds of which differ in precision. This illustrates the fact that, in general, CLP(BNR) depends on operational semantics.

Unlike the other constraint programming languages we have seen until now CLP(BNR) only keeps part of the problem locally consistent.

Variations on the theme of CLP(BNR) are discussed in [Benhamou *et al.*, 1994; Benhamou, 1995]. An approach where different kinds of methods are combined to make problems globally more consistent is discussed [Benhamou and Granvilliers, 1996]. Three different methods were used to make problems consistent: (a) local consistency techniques, (b) symbolic rewriting (Gröbner basis computation—a global consistency technique), and (c) interval methods (another local consistency technique). They provide some (relatively small) examples where the transformation of sets of equations to Gröbner bases as a preprocessing method decreased the overall solution time because the Gröbner bases were better suited for their interval algorithms.

2.3.5 Summary

Constraint logic programming has proved itself an interesting paradigm for the expression of problems in the form of programs, the solution of these programs, and the computation of a representation of the solutions in the form of constraints. Constraint logic programming languages frequently keep part of the problem globally consistent as opposed to constraint satisfaction programming which keeps the whole problem partially consistent. Work is undergoing to combine local and global consistency techniques.

2.4 Related Work in Mathematics

2.4.1 Introduction

In this section we shall study certain kinds of constraints which occur in mathematics. We shall provide references to the existing mathematical literature as part of the presentation.

The remainder of this section is as follows. In Section 2.4.2 we shall provide an introduction to the *first-order theory of the reals* (FOTR). In Section 2.4.3 we shall discuss applications of the first-order theory of the reals. We shall provide a brief summary in Section 2.4.4.

2.4.2 The First-Order Theory of the Reals

This section formally defines FOTR [Renegar, 1992a; 1992b; 1992c; Arnon, 1988; Arnon and Mignotte, 1988]. The decision problem for the first-order theory of the reals is the problem of determining the truth-values of certain kinds of formulae. These formulae may involve universal and existential quantifiers as well as the Boolean disjunctive and conjunctive connectives. At the "lowest" level, formulae are comparisons of polynomials whose coefficients are real. Valid sentences in FOTR satisfy the following four rules:

1. If $\mathbf{x}$ is a row matrix of variables, $p(\mathbf{x})$ a polynomial whose variables are a subset of the variables in $\mathbf{x}$ and whose coefficients are real, and Δ one of the comparison operators in $\{=, \neq, <, >, \leq, \geq\}$ then $p(\mathbf{x})\Delta 0$ is a sentence in FOTR.

2. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are sentences in FOTR and $\oplus$ is $\vee$ (disjunction) or $\wedge$ (conjunction) then $\mathcal{S}_1 \oplus \mathcal{S}_2$ is a sentence in FOTR.

3. If $\mathcal{S}$ is a sentence in FOTR, $\mathbf{x}$ is a row matrix of variables, and $\mathcal{Q}$ is one of the quantifiers in $\{\exists, \forall\}$ then $(\mathcal{Q}\mathbf{x})(\mathcal{S})$ is a sentence in FOTR. Here, the quantification of $\mathcal{Q}\mathbf{x}$ over $\mathcal{S}$ is

the obvious one, namely the quantification of the variables in $\mathbf{x}$ over $\mathcal{S}$. If $\mathcal{Q} = \exists$ then the quantification is existential. If $\mathcal{Q} = \forall$ then the quantification is universal.

4. If $\mathcal{S}$ is a sentence in FOTR then so is $\neg\mathcal{S}$.

It can be shown that every formula in FOTR can be written as an equivalent formula which has the following form:

$$(\mathcal{Q}_1\mathbf{x}_1 \in \mathbb{R}^{c_1})(\mathcal{Q}_2\mathbf{x}_2 \in \mathbb{R}^{c_2}) \cdots (\mathcal{Q}_n\mathbf{x}_n \in \mathbb{R}^{c_n})(P(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)), \tag{2.1}$$

where c_i is a positive integer, and $\mathcal{Q}_i \in \{\forall, \exists\}$, for $1 \leq i \leq n$, such that $\mathcal{Q}_i \neq \mathcal{Q}_{i+1}$, for $1 \leq i < n$, and where $P(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ is a quantifier free Boolean formula of the form:

$$\begin{array}{cccc} p_1(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) & \Delta_1 & 0 & \oplus_1 \\ \vdots & \vdots & \vdots & \vdots \\ p_{\tau-1}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) & \Delta_{\tau-1} & 0 & \oplus_{\tau-1} \\ p_\tau(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) & \Delta_\tau & 0, & \end{array}$$

where p_k is a real polynomial, and $\Delta_k \in \{=, \neq, <, >, \leq, \geq\}$, for $1 \leq k \leq \tau$, and where $\oplus_j \in \{\vee, \wedge\}$, for $1 \leq j < \tau$. Formulae like the one in Equation (2.1) are called *sentences*. The reader is referred to [Renegar, 1992a] for further details.

Example 2.13 (Sentence). The following formula is a sentence in FOTR:

$$(\exists[i] \in \mathbb{R}^1)(\forall[j] \in \mathbb{R}^1)(i - j > 0).$$

The sentence is obviously false.

Tarski has provided a decision algorithm for sentences in FOTR where there is only one existential quantifier [Tarski, 1951]. By making slight modifications to the algorithm, this can be turned into a decision method for FOTR. Tarski's method is by no means optimal and better methods have since then been discovered [Renegar, 1992a; 1992b; 1992c; Collins and Hong, 1991]. This thesis provides no complexity results for sentences in FOTR. The only statement that will be provided about the complexity of the decision problem of FOTR is that this problem is significantly more difficult than other commonly arising problems. [Hong, 1991] compares several algorithms to decide problems in FOTR and gives an indication of the solution time of certain problems which are currently tractable. It demonstrates that—at least at the moment it was written—tractable problems should be decided with algorithms whose worst-case time-complexities are not optimal.

Let $X = \{x_1, \ldots, x_n\}$ be a non-empty set of variables and let (X, D, C) be a finite CSP, where $C = \{C_{S_1}, \ldots, C_{S_m}\}$, $S_i = \left\{x_{i_1}, \ldots, x_{i_{k_i}}\right\}$, and $1 \leq k_i \leq n$, for $1 \leq i \leq m$. Without loss of generality it may be assumed that the domains of the variables are subsets of the reals. It is important to notice that the *decision* CSP—the problem of deciding whether there exists an "assignment" to the variables in X that satisfies each of the constraints—is an instance of FOTR.

As a consequence of this, techniques that are used to decide FOTR are also applicable to the decision CSP. The name "decision CSP" is due to [Bowen and Bahler, 1991].

It is not difficult to show that the decision CSP is an instance of FOTR. For example, let $S = \{ x_1, \ldots, x_n \}$, and let C_S be the constraint given by

$$C_S = \{ (v_{11}, \ldots, v_{1n}), \ldots, (v_{m1}, \ldots, v_{mn}) \} ,$$

then C_S is satisfiable if and only if the following sentence in FOTR is true:

$$(\exists(x_1, \ldots, x_n) \in \mathbb{R}^n)((x_1 = v_{11} \land \cdots \land x_n = v_{1n}) \lor \cdots \lor (x_1 = v_{m1} \land \cdots \land x_n = v_{mn})).$$

A CSP is satisfiable if there is a member of the Cartesian product of the domains of the variables which satisfies each of the constraints. It is left as an exercise for the reader to formulate the satisfiability of a general CSP as a sentence in FOTR.

2.4.3 Applications of FOTR

The first order theory of the reals has many applications and this section briefly discusses some of them. Our discussion is by no means complete and we shall only discuss approaches which are related to constraint logic programming.

An interesting application of FOTR is to quantifier elimination for special cases [Weispfenning, 1993]. Obvious applications of FOTR are to the satisfiability of systems of equations. This opens the door to geometric theorem proving (see e.g. [Kutzler and Stifter, 1986; Kapur, 1986; Sturm and Weispfenning, 1996]).

Less obvious applications are to decision problems that may arise as part of optimisation problems [Weispfenning, 1994] (see also [Hong and Văsaru, 1996]).

As pointed out in the previous section, the decision problem for certain kinds of CSPs are also instances of FOTR. Questions concerning the continuity of functions can also be expressed in FOTR.

In [Weispfenning, 1997] examples are provided of FOTR to the analysis of electronic networks and hydraulic networks (see also [Weispfenning, 1994; 1996]). An interesting side effect of the method used is that—as in constraint logic programming—expressions are returned that are necessary and sufficient conditions to guarantee that certain properties in these networks hold. The expressions that were returned are useful because they can be used to locate critical parts of the design. A disadvantage of the method is that the expressions which are returned are large and a lot of work has to be put into their simplification [Dolzmann and Sturm, 1995; Dolzmann *et al.*, 1996; Dolzmann and Sturm, 1997a; 1997b].

2.4.4 Summary

Special instances of problems occurring in the first-order theory of the reals have been studied for centuries. It is only since the 1930s that a general decision method has been known to decide any problem in this theory.

The area of applications of FOTR is vast and ranges from certain classes of decision CSPs to optimisation problems and geometric theorem proving.

It has only been since much more recently that methods have become available that can be applied to less trivial problems in FOTR.

2.5 Summary

In this chapter we have studied the use of constraints in different areas and methods to solve them. In particular we have studied constraint satisfaction problems (CSPs), constraint logic programming (CLP), and the first order theory of the reals (FOTR).

Constraint satisfaction problems are usually solved with variations of the chronological backtracking algorithm. The low level of consistency which most CSPs have in common causes thrashing and frequently leads to virtually endless search. The maintenance of low levels of consistency during search has proved to significantly improve the efficiency of search.

Constraints also occur in constraint logic programming. Different constraint programming languages may have different domains of computation. Domains of computation range from linear inequalities over the reals (CLP($\mathcal{R}$-Lin)), to equalities over the complex numbers (CAL), to sentences over the reals (RISC(CLP)), and many more. Some constraint logic programming languages maintain global consistency over their domain of computation. This has also proved to reduce thrashing.

Constraint satisfaction programming and constraint logic programming have different approaches to the level of consistency they maintain during search. Constraint satisfaction programming maintains partial levels of consistency over the problem as a whole, whereas many constraint logic programming languages maintain global consistency over a part of the problem. It remains to be seen if a combination of the two approaches could lead to an improvement.

The first-order theory of the reals allows for the formulation of many interesting problems. Since the 1930s this theory has been known to be decidable. A common solution technique for FOTR is quantifier elimination (sometimes restricted to special cases). Recent work has provided quantifier elimination algorithms which bring the decidability of interesting problems into the realm of feasibility. It can be shown that the decidability of the satisfiability of finite constraint satisfaction problems can be formulated as decision problems in FOTR.

Chapter 3

Varieties, Ideals and Gröbner Bases

3.1 Introduction

This chapter is an introduction to *varieties*, *ideals*, and *Gröbner bases*. The purpose of this chapter is to provide the required background for the next two chapters of this thesis.

Polynomial ideals are a useful tool for the expression and solution of many problems occurring in mathematics and the "real" world. Ideals can be used for the decision of the satisfiability of systems of equations, variable elimination, solution of simultaneous equations, and so on. Varieties are sets consisting of the common zeros of polynomial ideals. As such, they are intimately related to polynomial ideals. It will turn out that there also is a close relationship between certain kinds of varieties and finite constraints. Finite constraints—as we shall see in the following chapter—are in essence varieties. This allows for the translation of constraints to polynomial ideals, thereby allowing for the application of algorithms from ideal theory. The existence of the translation technique also means that it is possible to integrate discrete CSPs and problems of a continuous nature. This possibility, being interesting in itself, will not be further explored in this thesis.

Buchberger provided an algorithm (the Buchberger algorithm) for the computation of Gröbner bases in the 1960s. Gröbner bases provide useful tools in polynomial ideal theory. They allow for the decidability and solution of many problems which before their invention were not known to be solvable. Gröbner bases allow for the solution of each of the following problems:

ideal membership is a given polynomial a member of a given ideal?

consistency problem do the members of a given ideal have common zeros?

variable elimination eliminate certain variables from a given ideal.

dimension what is the dimension of a given ideal?

counting if a given ideal is zero-dimensional, i.e. if the members of the ideal have a non-zero finite number of common zeros, how many such common zeros are there?

As observed by Buchberger, there is also a relationship between *completion algorithms* such as the Knuth-Bendix algorithm [Knuth and Bendix, 1970] and *critical pair algorithms* like the Buchberger algorithm (See [Buchberger, 1987; Winkler, 1984]). Buchberger's observation allowed for improvements to the Knuth-Bendix algorithm similar to improvements—usually called Buchberger's criteria—made to the Buchberger algorithm.

In the remainder of this chapter we shall explain how each of the aforementioned problems can be solved using Gröbner basis theory and we shall provide the required mathematical background. In addition, we shall point out some relationships between constraint satisfaction problems and algebra. The style of presentation will be informal. It is only required that the reader is familiar with the notion of a *ring* in algebra. Throughout, we shall use examples to introduce ideas and exemplify certain concepts. It is not the purpose of this chapter to provide deep insights but to provide insights which are "easy" to remember and grasp. The reader interested in a complete presentation of Gröbner bases and the required background is referred to [Becker and Weispfenning, 1993; Adams and Loustaunau, 1994]. The reader interested in Gröbner bases and their relationship with geometry is referred to [Cox *et al.*, 1996]. The reader interested in a short introduction to Gröbner bases is referred to [Buchberger, 1985].

The remainder of this chapter is as follows. In Section 3.2 we shall study polynomial ideals and varieties and point out some relationships between ideals, varieties and constraints. In Section 3.3 we shall provide a formal definition of Gröbner bases. Section 3.4 consists of a presentation of several Gröbner basis algorithms for the decision and solution of problems in polynomial ideal theory.

3.2 Ideals and Varieties

3.2.1 Introduction

In this section we shall study ideals and varieties. The presentation will be of an informal nature.

Systems of polynomial equations have been studied for millenia. This has led to the notion of a *polynomial ideal* which captures properties of such systems very well. *Varieties* are solution sets of systems of polynomial equations. Certain classes of ideals and varieties turn out to be very closely related to finite constraints and we shall study their relationship.

Beside "ordinary" ideals, we shall also study special ideals: *vanishing ideals* which characterise varieties, *elimination ideals* which have an application to the solution of systems of polynomial equations, and *radical ideals* which correspond to certain kinds of vanishing ideals.

3.2.2 Ideals

In this thesis we shall only consider commutative rings with unity, i.e. we shall only consider rings $(R, \cdot, +)$ where multiplication is commutative and where there is a special member $1 \in R \setminus \{ 0 \}$ such that $(\forall r \in R)(r = 1 \cdot r)$.

Definition 3.1 (Ideal). Let R be a ring. A subring I of R is called an *ideal* of R if $(\forall r \in R)(\forall i \in I)(ri \in I)$.

Definition 3.2 (Generating System of Ideal). Let I be an ideal of ring R. A *generating system* of I is a subset, say F, of I such that every member i of I can be written as a sum of the form:

$$i = \sum_{f \in F} \phi_i(f) f,$$

where $\phi_i(\cdot)$ is some function from F to R which has *finite support*, i.e. $\phi_i(\cdot)$ has the property that there are only finitely many $f \in F$ such that $\phi_i(f) \neq 0$. An ideal is said to be *generated* by $F \subseteq R$ if F is a generating system of I. The ideal generated by F will be denoted $\langle F \rangle$. The ideal generated by $\{ f_1, \ldots, f_m \}$ will also be denoted $\langle f_1, \ldots, f_m \rangle$.

The *sum* of two ideals I and J is the set containing all sums of members from I and J. The sum of I and J is denoted $I + J$ and is given by

$$I + J = \{ i + j : (i, j) \in I \times J \},$$

where $\cdot \times \cdot$ is the Cartesian product operator. It is a straightforward exercise to prove that $I + J$ is, again, an ideal.

Polynomial rings are a special kind of rings. We shall denote the polynomial ring in (commuting) variables $X = \{ x_1, \ldots, x_n \}$ over ring R as $R[X]$ or as $R[x_1, \ldots, x_n]$. It contains all polynomials (including the zero polynomial) whose coefficients are in R and whose terms are power-products of the members of X.

A non-zero member u of ring R is called a *unit* if there is a $v \in R$ such that $uv = 1$. A ring is called a *field* if every non-zero member of that ring is a unit. With the exception of a single section in Chapter 4 where we shall consider other fields as well, the only field which we shall consider in this thesis is $\mathbb{C}$ and we shall write k for that field.

An ideal I of ring R is called a *proper* ideal of R if $I \subset R$. An ideal which has a generating system whose cardinality is finite is called a *finitely generated* ideal. A ring is called *Noetherian* if every ideal of that ring is finitely generated. The following theorem is better known as Hilbert's Basis Theorem.

Theorem 3.3. *Let R be a Noetherian ring, then $R[x]$ is also a Noetherian ring.*

A field is a Noetherian ring because it only contains the two ideals $\{ 0 \}$ and $\langle 1 \rangle$. One of the consequences of Hilbert's Basis Theorem is that ideals in polynomial rings with coefficients in fields are finitely generated. Although they have not been defined yet, this is a good point to state that Gröbner bases are finite generating systems of polynomial ideals with some interesting properties.

Let X and $W \subseteq X$ be non-empty sets of variables and let I be an ideal of $k[W]$. The smallest ideal (with respect to inclusion) of $R = k[X]$ containing I will be denoted I_R. It is called the *R-module* of I.

3.2.3 Varieties

An affine[1] *variety* in mathematics is a set which has the property that it is the smallest set (with respect to inclusion) containing the common zeros of some set of polynomials from some poly-

[1] The word affine means connected.

nomial ideal. Let $X = \{ x_1, \ldots, x_n \}$ be a non-empty set of variables, and let $F \subseteq k[X]$. The *variety* of F is the set where each of the polynomials in F *vanishes*, i.e. "becomes" zero. We shall normally order variables according to the "usual" lexicographical ordering $\cdot <_{\text{lex}} \cdot$. Normally, we shall assume that $x_i <_{\text{lex}} x_j \iff i < j$. The following provides a formal definition of a variety.

Definition 3.4 (Variety). Let n be a positive integer, and let $X = \{ x_1, \ldots, x_n \}$ be a set of variables such that $x_i <_{\text{lex}} x_j \iff i < j$, for $1 \leq i, j \leq n$. Furthermore, let k be a field, and let $F \subseteq k[X]$. The *variety* of F in k^n is denoted $\mathrm{V}(F)$ and is defined as follows:

$$\mathrm{V}(F) = \{ (v_1, \ldots, v_n) \in k^n \ : \ F \subseteq \langle x_1 - v_1, \ldots, x_n - v_n \rangle \} .$$

Note that $F \subseteq \langle x_1 - v_1, \ldots, x_n - v_n \rangle$ if and only if the substitution of $(v_1, \ldots, v_n)$ for $(x_1, \ldots, x_n)$ into each of the members of F is zero. It is a trivial exercise to show that $\mathrm{V}(F) = \mathrm{V}(\langle F \rangle)$ for every $F \subseteq k[X]$.

It is important to note that if $F \subseteq k[x_1, \ldots, x_n]$ and $V = \mathrm{V}(F)$ then the ordering on the variables decides which member of a tuple $(v_1, \ldots, v_n) \in V$ corresponds to which variable. Our lexicographical variable ordering is such that v_i corresponds to x_i and vice versa, for $1 \leq i \leq n$.

Example 3.5 (Variety (1)). Let $k = \mathbb{C}$, let $f = x + y - 1$, and let $g = x - y - 1$. The intersection of the lines $f = 0$ and $g = 0$ is the same as the variety in k^2 of the ideal I of $k[x, y]$ which is generated by $\{ f, g \}$. The following demonstrates how to find the intersection.

$$\begin{aligned}
\mathrm{V}(I) &= \mathrm{V}(\langle f, g \rangle) \\
&= \mathrm{V}(\langle x + y - 1, x - y - 1 \rangle) \\
&= \mathrm{V}(\langle x + y - 1, (x - y - 1) + (x + y - 1) \rangle) \\
&= \mathrm{V}(\langle x + y - 1, 2x - 2 \rangle) \\
&= \mathrm{V}(\langle x + y - 1, x - 1 \rangle) \\
&= \mathrm{V}(\langle (x + y - 1) - (x - 1), x - 1 \rangle) \\
&= \mathrm{V}(\langle y, x - 1 \rangle) \\
&= \{ (1, 0) \} .
\end{aligned}$$

Example 3.6 (Variety (2)). Let $k = \mathbb{C}$, let $f = x^2 + y^2 - 1$, and let $g = y - x^2$. Interpretations for f and g are that they represent the unit circle $x^2 + y^2 = 1$ and the parabola $y = x^2$, respectively. The intersection of the two is the set of common zeros of f and g. See Figure 3.1 for a graphical depiction.

Let $I = \langle f, g \rangle$. The common zeros of f and g in k^2, the intersection of the circle and the parabola in k^2, the complex solutions of the system of simultaneous equations $x^2 + y^2 = 1$ and $y = x^2$ and $\mathrm{V}(I)$ are all the same. The following shows how to find $\mathrm{V}(\langle f, g \rangle)$.

$$\begin{aligned}
I &= \left\langle x^2 + y^2 - 1, y - x^2 \right\rangle \\
&= \left\langle 1 \times (x^2 + y^2 - 1) - y \times (y - x^2), y - x^2 \right\rangle \\
&= \left\langle yx^2 + x^2 - 1, y - x^2 \right\rangle \\
&= \left\langle 1 \times (yx^2 + x^2 - 1) - x^2 \times (y - x^2), y - x^2 \right\rangle \\
&= \left\langle x^4 + x^2 - 1, y - x^2 \right\rangle .
\end{aligned}$$

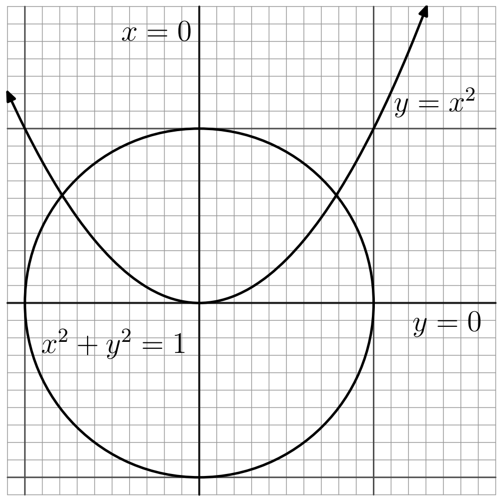

Figure 3.1: Intersection of circle and parabola.

Therefore, $\mathrm{V}(\langle f, g \rangle) = \mathrm{V}(\langle x^4 + x^2 - 1, y - x^2 \rangle)$. The generating system

$$\{ x^4 + x^2 - 1, y - x^2 \}$$

of I clearly indicates that there are at most four complex solutions. It does this as follows. The degree of the univariate polynomial in x is four. Therefore, there are at most four different zeros for x for which this polynomial *vanishes*, i.e. "becomes" zero. The polynomial $y - x^2$ is *linear* in y. For every zero of $x^4 - x^2 - 1$ for x there is exactly one y for which the polynomial $y - x^2$ vanishes. Therefore, the number of zeros is at most four.

The polynomial $x^4 - x^2 - 1$ has fewer than four zeros if and only if it has zeros whose multiplicity are greater than one. It has four zeros if and only if the multiplicity of each of its zeros is one. As will be shown later in this chapter a simple algorithm exists to transform polynomials like $x^4 + x^2 - 1$ into a polynomial h such that h has the same zeros and such that the multiplicity of each of the zeros of h is 1. With the aid of this algorithm it is possible to determine that there are exactly four zeros.[2] To find these zeros is the subject of a story with which this thesis will not be concerned. See for example [Becker and Weispfenning, 1993, Algorithm STURMSEQ] for an algorithm to isolate the real zeros of a uni-variate polynomial.

A field k is called *algebraically closed* if every non-constant polynomial in $k[x]$ has a zero. The *algebraic closure* of k is the smallest superset of k (with respect to inclusion) which is algebraically closed.

An ideal I of $k[X]$ is called *consistent* if $1 \notin I$ and *inconsistent* otherwise. It is a trivial exercise to prove that an ideal I of a ring R is a proper ideal of R if and only if $1 \notin I$ [Becker and Weispfenning, 1993, Lemma 1.39]. It is not difficult to see that if $1 \in I$ then $\mathrm{V}(I) = \emptyset$. If k is an algebraically closed field the converse is also true. This is stated as the following theorem.

[2]The zeros are given by $\sqrt{\sqrt{5} - 1/2}$, $-\sqrt{\sqrt{5} - 1/2}$, $i\sqrt{\sqrt{5} + 1/2}$ and $-i\sqrt{\sqrt{5} + 1/2}$.

Theorem 3.7 (Hilbert's Weak Nullstellensatz). *Let k be an algebraically closed field, let X be a non-empty set of variables, and let I be an ideal of $k[X]$, then I is consistent if and only if I is a proper ideal of $k[X]$ if and only if $1 \notin I$ if and only if $\mathrm{V}(I) \neq \emptyset$.*

The reader is referred to [Cox *et al.*, 1996, Hilbert's Weak Nullstellensatz] for proof and further details.

Note that if $I \subseteq k[X]$ then Hilbert's Weak Nullstellensatz provides us with information about the existence of common zeros of the members of an ideal. If k is algebraically closed then there are no common zeros if and only if $1 \in I$. Even if k is not algebraically closed then there are no zeros if $1 \in I$.

3.2.4 Vanishing Ideals

The *ideal* of a variety V is the ideal containing *all* polynomials that *vanish* at V, i.e. it contains all the polynomials that "become" zero at each of the members of V. It is for this reason that the ideal of V is also referred to as the *vanishing* ideal of V. The following provides a formal definition of a vanishing ideal.

Definition 3.8 (Vanishing Ideal). Let $X = \{ x_1, \ldots, x_n \}$ be a non-empty set of variables, let k be a field and let $V \subseteq k^n$. The *vanishing ideal* of V is denoted $\mathrm{I}(V)$ and is defined as follows:

$$\mathrm{I}(V) = \{ f \in k[X] \,:\, (\forall(v_1, \ldots, v_n) \in V)(f \in \langle x_1 - v_1, \ldots, x_n - v_n \rangle) \} .$$

It is a trivial exercise to show that vanishing ideals are, indeed, ideals. Note that Definition 3.8 implies that the following holds:

$$\begin{aligned}
\mathrm{I}(V) &= \{ f \in k[X] \,:\, (\forall(v_1, \ldots, v_n) \in V)(f \in \langle x_1 - v_1, \ldots, x_n - v_n \rangle) \} \\
&= \{ f \in k[X] \,:\, (\forall v \in V)(f \in \mathrm{I}(\{ v \})) \} \\
&= \{ f \in k[X] \,:\, f \in \cap_{v \in V} \mathrm{I}(\{ v \}) \} \\
&= k[X] \cap \bigcap_{v \in V} \mathrm{I}(\{ v \}) \\
&= \bigcap_{v \in V} (k[X] \cap \mathrm{I}(\{ v \})) \\
&= \bigcap_{v \in V} \mathrm{I}(\{ v \}).
\end{aligned}$$

This equivalence will allow us—as will be shown further on in this thesis—to transform a constraint network to a generating system of a polynomial ideal whose common zeros are the solutions of the network thus allowing for the application of algorithms from ideal (read Gröbner basis) theory to problems occurring in constraint satisfaction theory.

Example 3.9 (Vanishing Ideal). Let $k = \mathbb{C}$, let $n = 1$, let $V = \{ 0 \}$, and let $I = \mathrm{I}(V)$ be the vanishing ideal of $V \subset k$. Then

$$\begin{aligned} I &= \{\, f \in k[x] \,:\, (\forall v \in \{\, 0 \,\})(f \in \langle\, x - v \,\rangle) \,\} \\ &= \{\, f \in k[x] \,:\, f \in \langle\, x - 0 \,\rangle \,\} \\ &= \{\, f \in k[x] \,:\, f \in \langle\, x \,\rangle \,\} \\ &= k[x] \cap \langle\, x \,\rangle \\ &= \langle\, x \,\rangle \,. \end{aligned}$$

An ideal I of ring R is called a *maximal* ideal of R if I is a proper ideal of R and $I + J \in \{\, I, R \,\}$ for every ideal J of R. Ideals of the form $\mathrm{I}(\{\, v \,\})$ are maximal ideals. The maximal ideal $I = \langle\, x_1 - v_1, \ldots, x_n - v_n \,\rangle$ of $k[x_1, \ldots, x_n]$ is the vanishing ideal of the single point $(\, v_1, \ldots, v_n \,)$ of the affine space k^n.

3.2.5 Elimination Ideals

Another special kind of ideal is an *elimination ideal*. These have several applications. One application is to finding the common zeros of polynomials. The following provides a formal definition of elimination ideals.

Definition 3.10 (Elimination Ideal). Let k be a field, let X and $W \subseteq X$ be non-empty sets of variables, and let I be an ideal of $k[X]$. The *elimination ideal* of I with respect to W is the ideal $I \cap k[W]$. The elimination ideal of I with respect to W is denoted as $\mathrm{I}_W(I)$.

The elimination ideal of $I \subseteq k[X]$ with respect to W consists of all the polynomials in I each of whose terms involve only power-products of variables in W.

Example 3.11 (Elimination Ideal (1)). Let $k = \mathbb{C}$, and let

$$I = \left\langle\, (x-2)^2 + (y-2)^2 - 4, y - (x-2)^2 \,\right\rangle \subseteq k[x, y].$$

We can eliminate y from I by "substituting" $(x-2)^2$ for y "in" I. This leads to the elimination ideal $I \cap k[x] = \langle\, x^4 - 8x^3 + 21x^2 - 20x + 4 \,\rangle$.

One application of elimination ideals is to the solution of systems of simultaneous polynomial equations by recursively eliminating variables and extending partial solutions.

Example 3.12 (Elimination Ideal (2)). Let $k = \mathbb{C}$. The system

$$E = \left\{\, (x-2)^2 + (y-2)^2 = 4, y = (x-2)^2 \,\right\}$$

can be solved using elimination ideals. Solving E corresponds to computing the set of common zeros $V \subseteq k^2$ of the polynomials in $F = \{\, (x-2)^2 + (y-2)^2 - 4, y - (x-2)^2 \,\} \subseteq k[x, y]$. $V = \mathrm{V}(F) = \mathrm{V}(I)$, where $I = \langle\, F \,\rangle$. We discovered in Example 3.11 that to eliminate y from I leads to the elimination ideal $I \cap k[x] = \langle\, x^4 - 8x^3 + 21x^2 - 20x + 4 \,\rangle = \left\langle\, (x-2)^2(x-2+\sqrt{3})(x-2-\sqrt{3}) \,\right\rangle$. The common zeros of the elements in $I \cap k[x]$ are given by $x \in \{\, 2-\sqrt{3}, 2, 2+\sqrt{3} \,\}$. For each common zero for x of the members of $I \cap k[x]$ there is exactly one extended zero for $(\, x, y \,)$ of the members of $I \cap k[x,y] = I$. The solutions of E are therefore given by $\{\, (\, 2-\sqrt{3}, 3 \,), (\, 2, 0 \,), (\, 2+\sqrt{3}, 3 \,) \,\}$, i.e. the equations in E are simultaneously true precisely when $(\, x, y \,) = (\, 2-\sqrt{3}, 3 \,)$ or $(\, x, y \,) = (\, 2, 0 \,)$ or $(\, x, y \,) = (\, 2+\sqrt{3}, 3 \,)$.

As will be shown further on in this chapter, Gröbner Basis Theory provides an algorithm to compute elimination ideals.

In the process of finding the common zeros of polynomials using elimination ideals it frequently occurs that an ideal is "projected" onto a "smaller" ideal. After the common zeros of the members of the smaller ideal have been located, each such common zero has to be extended to a common zero of the members of the "bigger" ideal. This extension process, as we shall see in the following example (adapted from [Cox *et al.*, 1996, p. 115]) is not always guaranteed to succeed.

Example 3.13 (Failing Extension). Let $k = \mathbb{C}$, let $X = \{\, x, y, z \,\}$, let $F = \{\, xz - 1, yz - 1 \,\}$, and let $I = \langle\, F \,\rangle \subset k[X]$. It can be shown that the elimination ideal of I with respect to $\{\, x, y \,\}$ is given by:

$$I \cap k[x,y] = \{\, y - x \,\} \,.$$

Therefore, the common zeros of the members of $I \cap [x,y]$ are given by $\{\, (\, v, v \,) \,:\, v \in k \,\}$. However, the partial solution $(\, 0, 0 \,)$ cannot be extended to a solution including z.

In the previous example we have seen that the extension of partial solutions is not always guaranteed to succeed. In the following paragraphs we shall present a theorem which provides a sufficient condition to guarantee the extension of partial solutions. Before doing so we remind the reader that if I is an ideal of $k[x_1, \ldots, x_n]$ then $(\, v_1, \ldots, v_n \,) \in \mathrm{V}\,(I)$ if and only if every polynomial $f \in I$ vanishes for the simultaneous substitution $(\, v_1, \ldots, v_n \,)$ for $(\, x_1, \ldots, x_n \,)$ in f, that is:

$$(\, v_1, \ldots, v_n \,) \in \mathrm{V}\,(I) \iff I \subseteq \langle\, x_1 - v_1, \ldots, x_n - v_n \,\rangle \,.$$

Before we study the Extension Theorem which provides a sufficient condition for the extension of partial solutions, we have to define the notion of the *leading coefficient* of a variable in a multivariate polynomial. This notion will only be used in this part of the thesis. Let $f \in k[x_1, \ldots, x_n]$ be a non-zero polynomial. Notice that we can uniquely write every non-zero f as a sum of the form $f = \sum_{i=0}^{\alpha} c_i x_n^i$, for suitably chosen α and c_i such that $c_\alpha \neq 0$ and $c_i \in k[x_1, \ldots, x_{n-1}]$, for $0 \leq i \leq \alpha$. We call c_α the *leading coefficient* of x_n in f.

The following theorem can be found in slightly different form in [Cox *et al.*, 1996, Theorem 3, Page 115]. The theorem provides a sufficient condition to guarantee when partial solutions can be extended.

Theorem 3.14 (Extension). *Let $X = \{\, x_1, \ldots, x_n \,\}$ be a set containing at least two variables, let $k = \mathbb{C}$, let $F = \{\, f_1, \ldots, f_m \,\} \subset k[X] \setminus \{\, 0 \,\}$, and let $I = \langle\, F \,\rangle \subseteq k[X]$. Finally, let g_i be the leading coefficient of x_n in f_i, for $1 \leq i \leq m$. If $I \cap k[x_1, \ldots, x_{n-1}] \subseteq \langle\, x_1 - v_1, \ldots, x_{n-1} - v_{n-1} \,\rangle$ and $\langle\, g_1, \ldots, g_m \,\rangle \not\subseteq \langle\, x_1 - v_1, \ldots, x_{n-1} - v_{n-1} \,\rangle$ then there exists $v_n \in k$ such that $I \subseteq \langle\, x_1 - v_1, \ldots, x_n - v_n \,\rangle$.*

The Extension Theorem states that every partial solution for $x_1, \ldots, x_{n-1}$ can be extended to a partial solution for $x_1, \ldots, x_n$ if the leading coefficients of x_n of the polynomials in F do not vanish simultaneously.[3] The reader is referred to [Cox *et al.*, 1996, Theorem 3, Page 115] for proof and further details about the Extension Theorem and Elimination Theory.

Note that in Example 3.13 the leading coefficients of z of the members of the generating system of the ideal vanish simultaneously for the case where $x = y = 0$. Therefore, the extension of the partial solution $x = y = 0$ was not guaranteed.

Related to the notion of an elimination ideal is that of the *dimension* of an ideal.

Definition 3.15 (Dimension of Ideal). Let X be a non-empty set of variables and I a proper ideal of $k[X]$. The *dimension* of I is defined as:

$$\max(\{\, |W| \,:\, W \subseteq X, I \cap k[W] = \{\,0\,\}\,\}).$$

Without proof it is stated that *zero-dimensional ideals* are proper ideals the cardinality of the varieties of which is finite.

3.2.6 Radical Ideals

Another special kind of ideals are *radical* ideals. Radical ideals in $k[x_1, \ldots, x_n]$ and varieties in k^n are closely related. If k is algebraically closed then there is a one-to-one relationship between the two.

Definition 3.16 (Radical Ideal). Let X be a non-empty set of variables. A proper ideal I of $k[X]$ is called a *radical* ideal of $k[X]$ if it satisfies the property that:

$$(\forall m \in \mathbb{N} \setminus \{\,0\,\})(\forall p \in k[X])(p^m \in I \Longrightarrow p \in I).$$

The *radical* of an ideal I is the ideal $\{\, f \in k[X] \,:\, (\exists m \in \mathbb{N} \setminus \{\,0\,\})(f^m \in I)\,\}$. The radical of I is denoted $\sqrt{I}$.

Note that an ideal I is a radical ideal if and only if $I = \sqrt{I}$.

Example 3.17 (Radical Ideal (1)). Let $k = \mathbb{C}$ and let $I = \langle\, x^2 \,\rangle$. Clearly $\sqrt{I} = \langle\, x \,\rangle$. The variety $V \subset k$ of $I \subset k[x]$ contains the common zeros of the members of I, i.e. $V = \mathrm{V}(I) = \{\,0\,\}$. The ideal J of the variety V contains all univariate polynomials in x which vanish at 0. In order for a univariate polynomial in x to vanish at 0 it has to be of the form $x \times f$, where f is some polynomial in $k[x]$. Therefore, $J = \mathrm{I}(V) = \{\, x \times f \,:\, f \in k[X]\,\} = \langle\, x \,\rangle$. Note that $x \in J$, $x \times x = x^2 \in J$ and $x \notin I$. Therefore, $I = \langle\, x^2 \,\rangle \subset \langle\, x \,\rangle = \sqrt{I} = J$,

Example 3.18 (Radical Ideal (2)). Let $k = \mathbb{C}$ and let I be the ideal given by:

$$I = \left\langle\, x^2(x-1), (x^2+1)(x-1) \,\right\rangle \subset k[x].$$

[3] Note that the Extension Theorem only provides a sufficient condition for extension. It does not state that the extension is impossible if the leading coefficients do vanish simultaneously.

Then I is radical. For example,

$$\begin{aligned} I &= \langle\, x^2(x-1), (x^2+1)(x-1) \,\rangle \\ &= \langle\, x^2(x-1), (x^2+1)(x-1) - x^2(x-1) \,\rangle \\ &= \langle\, x^2(x-1), x-1 \,\rangle \\ &= \langle\, x-1 \,\rangle , \end{aligned}$$

and it is not difficult to see that $I = \sqrt{I}$.

Example 3.19 (Radical Ideal (3)). Let $k = \mathbb{C}$ and let I be the ideal given by:

$$I = \langle\, x^2(x-1) \,\rangle \subset k[x].$$

It is not difficult to see that I is not a radical ideal. For example, each polynomial in I is of the form $fx^2(x-1)$, for some $f \in k[x]$. For the particular choice of $f = x-1$ it follows that $(x-1)x^2(x-1) = (x(x-1))^2 \in I$. Since I contains $(x(x-1))^2$ it follows that $\sqrt{I}$ contains $x(x-1)$, whereas $x(x-1)$ is not in I.

Let k be any field and let $p \in k[x]$ be a polynomial with $m \geq 1$ distinct zeros $v_1, \ldots, v_m$. The *square-free part* of p is defined as $\prod_{i=1}^{m}(x - v_i)$. A polynomial which is equal to its square-free part (up to multiplication by a constant) is called a *square-free polynomial*. The square-free part of a non-constant univariate polynomial $f \in k[x]$ can be computed by dividing f by $\gcd(f, \frac{df}{dx})$, where $\frac{df}{dx}$ is the first derivative of f with respect to x, and $\gcd(p, q)$ is the *greatest common divisor* of p and q. The reader is referred to [Cox *et al.*, 1996, Proposition 12, Page 179] for proof and further details.

A non-zero univariate polynomial is called *monic* if its leading coefficient is 1. A polynomial $f \in k[x] \setminus \{\, 0 \,\}$ is called *irreducible* if $f = gh$ implies that either g or h is a unit. A polynomial in $k[x]$ is called *separable* if it does not have multiple zeros in $K[x]$, where K is the *algebraic closure* of k. A field k is called *perfect* if every irreducible polynomial in $k[x]$ is separable. The field of the complex numbers $\mathbb{C}$ is perfect [Becker and Weispfenning, 1993, p. 311]. Finite fields are also perfect [Becker and Weispfenning, 1993, Corollary 7.73].

The following lemma can be found as [Becker and Weispfenning, 1993, Lemma 8.19].

Lemma 3.20 (Zero-Dimensional Radical Ideal). *Let $k = \mathbb{C}$, let $X = \{\, x_1, \ldots, x_n \,\}$ be a set of variables, and let I be a zero-dimensional ideal of $k[X]$. Furthermore, let f_i be the unique monic polynomial of minimal degree in $I \cap k[x_i]$ and let g_i be the square-free part of f_i, for $1 \leq i \leq n$. Then*

$$\sqrt{I} = I + \langle\, g_1, \ldots, g_n \,\rangle .$$

Each of the polynomials f_i can be computed by computing a generating system of the elimination ideal $I \cap k[x_i]$.

As will be pointed out further on in this section, Gröbner Basis Theory provides a simple algorithm which, if provided with a generating system of an ideal, can be used to compute generating systems of elimination ideals of that ideal. Together with Lemma 3.20, the algorithm can be used to compute a generating system of the radical of a zero-dimensional ideal.

The following theorem can be found as [Cox *et al.*, 1996, Proposition 16, Chapter 4].

Theorem 3.21 (Radical Ideal Intersection). *If I and J are radical ideals of $k[X]$ then $I \cap J$ is also a radical ideal of $k[X]$.*

In general, it does not hold that $I+J$ is radical if I and J are radical. For example, let $I = \langle\, x \,\rangle$ and $J = \langle\, x + y^2 \,\rangle$, then I and J are radical but $I + J = \langle\, x, x + y^2 \,\rangle = \langle\, x, x - x + y^2 \,\rangle = \langle\, x, y^2 \,\rangle$ is not. However, if I and J are zero-dimensional radical ideals then we can prove that for special cases their sum is zero-dimensional and radical. This is formulated as the following proposition which will turn out to be useful in the following chapter.

Proposition 3.22 (Radicality). *Let m be a positive integer. For each positive integer i less than or equal to m let X_i be a non-empty set of variables. Furthermore, let $X = \cup_{i=1}^{m} X_i$, let $R = k[X]$, let I_i be a zero-dimensional radical ideal of $k[X_i]$ and let R_i be the R-module of I_i, for $1 \leq i \leq m$. Finally, let $J \subseteq k[X]$ be the ideal given by*

$$J = \sum_{i=1}^{m} R_i.$$

Then either J is inconsistent or J is a zero-dimensional radical ideal of R.

Proof. Assume J is consistent. I_i is zero-dimensional and radical. By Lemma 3.20, I_i contains a non-constant square-free polynomial of minimal degree in $k[x_{i_j}]$, for each of the variables $x_{i_j} \in X_i$, for $1 \leq i \leq m$. Note that $I_i \subseteq R_i \subseteq J$, for $1 \leq i \leq m$. Therefore, J also contains a non-constant square-free polynomial of minimal degree in $k[x]$, for each of the variables $x \in X$. J is consistent and contains a non-constant square-free polynomial of minimal degree in $k[x]$, for each of the variables $x \in X$. Therefore, $J \cap k[x]$ contains a unique non-constant square-free polynomial of minimal degree in $k[x]$, for each $x \in X$. It now follows from Lemma 3.20 that J is zero-dimensional and radical. $\square$

3.2.7 Ideal-Variety Correspondence

In this section we shall study the relationship between ideals and varieties in greater detail. In particular we shall study the relationship between radical ideals and varieties, the relationship between *intersection* of two ideals and the union of their varieties, and the relationship between the *sum* of two ideals and the intersection of their varieties.

Radical Ideals and Varieties

The following theorem is important because it provides information about the structure of vanishing ideals.

Theorem 3.23 (Hilbert's Strong Nullstellensatz). *Let k be an algebraically closed field. If I is an ideal of $k[X]$ then the radical of I and the vanishing ideal of the variety of I are equal, i.e. $\sqrt{I} = \mathrm{I}(\mathrm{V}\,(I))$.*

The reader is referred to [Cox *et al.*, 1996, Hilbert's Strong Nullstellensatz, p. 174] for proof and further details.

Hilbert's Strong Nullstellensatz (Theorem 3.23) relates ideals, varieties, and their radicals. The following theorem provides us with more information about their relationship. The reader is referred to [Cox *et al.*, 1996, Ideal-Variety Correspondence Theorem,p. 175] for proof and further details.

Theorem 3.24 (Ideal-Variety Correspondence). *Let k be an algebraically closed field, then the maps*

$$\textit{affine varieties} \xrightarrow{\mathrm{I}} \textit{radical ideals}$$

and

$$\textit{radical ideals} \xrightarrow{\mathrm{V}} \textit{affine varieties}$$

are inclusion-reversing bijections which are inverses of each other.

The theorem allows us to transform radical ideals to varieties and back without losing information. We shall frequently make use of this relationship.

Example 3.25 (Ideal-Variety Correspondence). Let $k = \mathbb{C}$, and let $I = \langle\, x^2(x-1)\,\rangle$, let $J = \langle\, x(x-1)\,\rangle$, and let $K = \langle\, x\,\rangle$ be ideals of $k[x]$. Then $I \subseteq J \subseteq K$, and it follows from the first part of Theorem 3.24 that $\mathrm{V}\,(I) \supseteq \mathrm{V}\,(J) \supseteq \mathrm{V}\,(K)$. The following demonstrates that this is, indeed, true.

$$\begin{aligned} \mathrm{V}\,(I) &= \{\,0,1\,\} \\ &\supseteq \{\,0,1\,\} \\ &= \mathrm{V}\,(J) \\ &= \{\,0,1\,\} \\ &\supseteq \{\,0\,\} \\ &= \mathrm{V}\,(K)\,. \end{aligned}$$

Note that I is not radical, whereas J and K are. Since $J \subset K$, it follows from the second part of Theorem 3.24 that $\mathrm{V}\,(J) \supset \mathrm{V}\,(K)$. The following demonstrates that this is, indeed, true.

$$\begin{aligned} \mathrm{V}\,(J) &= \{\,0,1\,\} \\ &\supset \{\,0\,\} \\ &= \mathrm{V}\,(K)\,. \end{aligned}$$

Intersection of Ideals

Another interesting relationship is that between the intersection of ideals and the union of their varieties. The following theorem is proved in [Cox *et al.*, 1996, Chapter 4.3, Theorem 15].

Theorem 3.26 (Intersection versus Union). *If I and J ideals of $k[X]$ then*

$$\mathrm{V}\,(I \cap J) = \mathrm{V}\,(I) \cup \mathrm{V}\,(J)\,.$$

This equivalence also will turn out to be very convenient. Provided we have an algorithm for ideal intersection, we can compute a generating system of an ideal $I \cap J$ whose variety is the union of the varieties of two other ideals I and J. The relationship will allow us—as we shall see in the following chapter—to "translate" a constraint to an ideal. As will be demonstrated in Section 3.4.5, a simple algorithm to intersect two ideals does, indeed, exist.

Example 3.27 (Intersection of Ideals versus Union of Varieties). Let $I = \langle\, x - 1 \,\rangle$ and $J = \langle\, x - 2 \,\rangle$ then

$$\begin{aligned} \mathrm{V}\left(I \cap J\right) &= \mathrm{V}\left(\langle\, (x-1)(x-2) \,\rangle\right) \\ &= \{\, 1, 2 \,\} \\ &= \{\, 1 \,\} \cup \{\, 2 \,\} \\ &= \mathrm{V}\left(\langle\, x - 1 \,\rangle\right) \cup \mathrm{V}\left(\langle\, x - 2 \,\rangle\right) \\ &= \mathrm{V}\left(I\right) \cup \mathrm{V}\left(J\right). \end{aligned}$$

The first equality is justified because the intersection of I and J contains all polynomials which are in I (i.e. are a multiple of $x - 1$) and also in J (i.e. are a multiple of $x - 2$).

In general, it can also be shown that if I and J are ideals then $\mathrm{V}\left(I \cap J\right) = \mathrm{V}\left(I \cdot J\right)$, where $I \cdot J$ is the *product* of I and J, i.e. $I \cdot J = \{\, ij \,:\, (\, i, j \,) \in I \times J \,\}$. Note that even if I and J are radical this does not always mean that $I \cap J = I \cdot J$. For example, $\langle\, x \,\rangle \cap \langle\, x \,\rangle = \langle\, x \,\rangle \neq \langle\, x^2 \,\rangle = \langle\, x \,\rangle \cdot \langle\, x \,\rangle$. If I and J are radical ideals then $I \cdot J \subset I \cap J$. The reader is referred to [Cox *et al.*, 1996, Chapter 4.3] for further information.

Sums of Ideals

The following theorem is proved in [Cox *et al.*, 1996, Chapter 4.3, Theorem 4].

Theorem 3.28 (Sum versus Intersection). *If I and J ideals of $k[X]$ then*

$$\mathrm{V}\left(I + J\right) = \mathrm{V}\left(I\right) \cap \mathrm{V}\left(J\right).$$

As already indicated we shall demonstrate further on in this thesis how to convert a constraint to an ideal. The relationship between the sum of two ideals and the intersection of their varieties will allow us to construct an ideal whose variety is equal to the solutions of a constraint network.

Notice that from an intuitive point of view it is pretty easy to see that Theorem 3.28 must, indeed, hold. The common zeros of two sets of polynomials are equal to the common zeros of the union of those sets, which in their turn are equal to the intersection of the common zeros of those two sets.

Example 3.29 (Sum of Ideals versus Intersection of Varieties (1)). Let $k = \mathbb{C}$, let $I = \langle\, x(x-1) \,\rangle$, and let $J = \langle\, x(x-2) \,\rangle$ be ideals of $k[x]$, then $V = \mathrm{V}\left(I\right) = \{\, 0, 1 \,\}$, and

$W = \mathrm{V}(J) = \{0, 2\}$. According to Theorem 3.28 $\mathrm{V}(I + J) = V \cap W = \{0\}$. The following demonstrates that this does, indeed, hold.

$$\begin{aligned}
\mathrm{V}(I+J) &= \mathrm{V}(\langle x(x-1)\rangle + \langle x(x-2)\rangle) \\
&= \mathrm{V}(\langle x(x-1)\rangle + \langle x(x-1) - x(x-2)\rangle) \\
&= \mathrm{V}(\langle x(x-1)\rangle + \langle x\rangle) \\
&= \mathrm{V}(\langle x\rangle) \\
&= \{0\}.
\end{aligned}$$

Example 3.30 (Sum of Ideals versus Intersection of Varieties (2)). Let $I = \langle x(x-1), y-x\rangle \subset k[x,y]$ and $J = \langle x(x+1), y+x\rangle \subset k[x,y]$. It is not difficult to see that $\mathrm{V}(I) = \{(0,0), (1,1)\}$ and that $\mathrm{V}(J) = \{(0,0), (-1,1)\}$. Furthermore,

$$\begin{aligned}
\mathrm{V}(I+J) &= \mathrm{V}(\langle x(x-1), y-x, x(x+1), y+x\rangle) \\
&= \mathrm{V}(\langle x(x-1), (y-x)/2 + (y+x)/2, x(x+1), y+x\rangle) \\
&= \mathrm{V}(\langle x(x-1), y, x(x+1), y+x\rangle) \\
&= \mathrm{V}(\langle x(x-1), y, x(x+1), x\rangle) \\
&= \mathrm{V}(\langle x, y\rangle) \\
&= \{(0,0)\} \\
&= \{(0,0), (1,1)\} \cap \{(0,0), (-1,1)\} \\
&= \mathrm{V}(I) \cap \mathrm{V}(J).
\end{aligned}$$

Distributive Property

In this section we shall prove a proposition about zero-dimensional radical ideals which will be needed in the following chapter. Before we prove the proposition is true, we present the following lemma, which is a special case of Proposition 3.22.

Corollary 3.31. *Let I and J be zero-dimensional radical ideals of $k[X]$, then either $I + J$ is inconsistent or is a zero-dimensional ideal of $k[X]$.*

Proof. Trivial. □

Proposition 3.32 (Distributive Property). *If I, J and K are zero-dimensional radical ideals of $k[X]$ then*

$$I \cap (J + K) = (I \cap J) + (I \cap K).$$

Proof. Assume $J + K$ is inconsistent. By Hilberbert's Weak Nullstellensatz (Theorem 3.7),

$J + K = \langle 1 \rangle$. Therefore,

$$\begin{aligned} I \cap (J + K) &= I \cap \langle 1 \rangle \\ &= I \\ &= I \cdot \langle 1 \rangle \\ &= I \cdot (J + K) \\ &= I \cdot J + I \cdot K \\ &\subseteq (I \cap J) + (I \cap K). \end{aligned}$$

The reverse inclusion also holds, because $I \cap (J + K)$ contains $I \cap J$ and contains $I \cap K$. By virtue of it being an ideal, $I \cap (J + K)$ must therefore also contain $(I \cap J) + (I \cap K)$.

Assume $J + K$ is consistent. The ideal-variety correspondence (Theorem 3.24) allows us to derive the following equivalence between $\mathrm{V}(I \cap (J + K))$ and $\mathrm{V}((I \cap J) + (I \cap K))$.

$$\begin{aligned} \mathrm{V}(I \cap (J + K)) &= \mathrm{V}(I) \cup \mathrm{V}(J + K) \\ &= \mathrm{V}(I) \cup (\mathrm{V}(J) \cap \mathrm{V}(K)) \\ &= (\mathrm{V}(I) \cup \mathrm{V}(J)) \cap (\mathrm{V}(I) \cup \mathrm{V}(K)) \\ &= \mathrm{V}(I \cap J) \cap \mathrm{V}(I \cap K) \\ &= \mathrm{V}((I \cap J) + (I \cap K)). \end{aligned}$$

The varieties of the ideals $I \cap (J + K)$ and $(I \cap J) + (I \cap K)$ are equal. We shall use the ideal-variety correspondence (Theorem 3.24) to prove that the two ideals are equal by showing that they are radical.

By Corollay 3.31, $J + K$ is radical. I is also radical and it follows from Theorem 3.21 that $I \cap (J + K)$ is radical. By Hilbert's Weak Nullstellensatz, $(I \cap J) + (I \cap K)$ is consistent because $I \cap (J + K)$ is consistent and the varieties of the two ideals are equal. By Theorem 3.21, $I \cap J$ and $I \cap K$ are zero-dimensional and radical. We conclude the proof by observing that $(I \cap J) + (I \cap K)$ is radical by Corollay 3.31. □

Note that in general the distributive property may not always hold if the ideals are not radical or zero-dimensional. For example, let $I = \langle y + x \rangle$, let $J = \langle x^2 \rangle$, and let $K = \langle y^2 \rangle$. Then

$$\begin{aligned} I \cap J &= \left\langle (y + x)x^2 \right\rangle; \\ I \cap K &= \left\langle (y + x)y^2 \right\rangle; \\ (I \cap J) + (I \cap K) &= \left\langle (y + x)x^2, (y + x)y^2 \right\rangle; \\ I \cap (J + K) &= \langle y + x \rangle \cap (\left\langle y^2 \right\rangle + \left\langle x^2 \right\rangle) \\ &= \langle y + x \rangle \cap (\left\langle y^2 - x^2 \right\rangle + \left\langle x^2 \right\rangle) \\ &= \langle y + x \rangle \cap (\langle (y + x)(y - x) \rangle + \left\langle x^2 \right\rangle), \end{aligned}$$

and it is not difficult to see that $I \cap (J + K)$ contains $\langle (y + x)(y - x) \rangle$, whereas $(I \cap J) + (I \cap K)$ does not.

3.3 Gröbner Bases

3.3.1 Introduction

In this section we shall study *term orders* and *Gröbner bases*. Like the previous sections our treatment will be informal.

Gröbner bases are finite generating systems of polynomial ideals with the additional property that the *leading terms* of their members with respect to a *term order* characterise the leading terms of the members of the ideal with respect to the same term order. It will turn out that this allows for the decision and solution of many problems in ideal theory.

3.3.2 Term Orders

Term orders are to Gröbner bases what variable orderings are to the Gaussian Elimination Algorithm. They are a generalisation of variable orders (orders on linear terms) in the sense that they are also orders on non-linear terms. Term orders preserve the ordering which is induced by division, thereby allowing for a generalisation of quotient and remainder (with respect to a term order). In our presentation we shall not need quotients and remainders with respect to term orders. Instead, we shall rely on the notion of *normal form* of a polynomial with respect to a finite set of non-zero polynomials and a term order. The remainder of this section is an introduction to term orders and normal forms. The reader is referred to [Becker and Weispfenning, 1993] and [Cox *et al.*, 1996] for further information about division with respect to a term order.

Let $X = \{ x_1, \ldots, x_n \}$ be a non-empty set of variables. The set containing all power-products of the members of X will be denoted $\mathbb{T}_X$. Formally,

$$\mathbb{T}_X = \{ x_1^{\alpha_1} \times \cdots \times x_n^{\alpha_n} \,:\, (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n \} .$$

Definition 3.33 (Term Order). Let X be a non-empty set of variables. A *term order* $\prec$ on $\mathbb{T}_X$ is a total order on $\mathbb{T}_X$ with the additional property that 1 is the smallest member of $\mathbb{T}_X$ with respect to $\prec$ and that for every $u, v \in \mathbb{T}_X$ it holds that whenever $u \prec v$ it must be true that $tu \prec tv$ for every $t \in \mathbb{T}_X$.

Example 3.34 (Lexicographical Term Order). Let $X = \{ u, v \}$ and let $\prec$ be the order on $\mathbb{T}_X$ such that $u^{\alpha_1} v^{\beta_1} \prec u^{\alpha_2} v^{\beta_2}$ if either $(\alpha_1 < \alpha_2)$ or $(\alpha_1 = \alpha_2 \wedge \beta_1 < \beta_2)$. Then $\prec$ is a term order on $\mathbb{T}_X$. For obvious reasons it is called the *lexicographical* term order such that $u \prec v$.

For example, let $\prec$ be the term order such that $1 \prec x \prec x^2 \prec \cdots \prec y \prec xy \prec x^2y \prec \cdots$. It is the lexicographical term order such that x precedes y.

Example 3.35 (Total Degree Order). Let $X = \{ u, v \}$ and let $\prec$ be the order on $\mathbb{T}_X$ such that $u^{\alpha_1} v^{\beta_1} \prec u^{\alpha_2} v^{\beta_2}$ if either $(\alpha_1 + \beta_1 < \alpha_2 + \beta_2)$ or $(\alpha_1 + \beta_1 = \alpha_2 + \beta_2 \wedge \alpha_1 < \alpha_2)$. Then $\prec$ is a term order on $\mathbb{T}_X$. It is called the *total degree* order such that $u \prec v$.

For example, let $\prec$ be the term order such that $1 \prec x \prec y \prec x^2 \prec xy \prec y^2 \prec x^3 \prec \cdots$. It is the total degree order such that x precedes y.

A *monomial* is the product of a non-zero constant and a term. The most significant term of a non-zero polynomial with respect to a term order is called the *leading term* of that polynomial with respect to that term order. The leading term of a non-zero polynomial p with respect to the term order $\prec$ will be denoted $\mathrm{lt}_\prec(p)$. The *leading monomial* of a non-zero polynomial with respect to a term order is the monomial of that polynomial whose term is the leading term of that polynomial with respect to that term order. The leading monomial of p with respect to $\prec$ will be denoted $\mathrm{lm}_\prec(p)$. The *leading coefficient* of a polynomial p with respect to a term order $\prec$ is equal to $\mathrm{lm}_\prec(p)/\,\mathrm{lt}_\prec(p)$. The leading coefficient of p with respect to $\prec$ will be denoted $\mathrm{lc}_\prec(p)$.

Let $\cdot \mid \cdot$ be the relation defined on terms such that $u \mid v$ if u divides v. Furthermore, let $\cdot \nmid \cdot$ be the relation defined on terms such $u \nmid v$ if u does not divide v.

The following defines a normal form of a polynomial with respect to a set of polynomials and a term order.

Definition 3.36 (Normal Form). Let X be a finite set of variables, let $\prec$ be a term order on $\mathbb{T}_X$, let F be a finite subset of $k[X] \setminus \{ 0 \}$, and let $p \in k[X]$ a polynomial. Then $q \in k[X]$ is called a *normal form* of p with respect to F and $\prec$ if $q - p \in \langle F \rangle$ and for all $f \in F$ none of the terms of q are divided by $\mathrm{lt}_\prec(f)$.

It is important to notice that normal forms are not unique. For example, let $\prec$ be the lexicographical term order such that $x \prec y$, let $F = \{ y^2 - x, y - x \}$, and let $p = y^2$, then both $x = p - (y^2 - x)$ and $x^2 = p - (y + x)(y - x)$ are normal forms of p with respect to F and $\prec$.

3.3.3 Definition of Gröbner Bases

In this section we shall define the notion of a *Gröbner basis* and that of a *reduced Gröbner basis* of an ideal with respect to a term order.

Definition 3.37 (Gröbner Basis). Let k be a field, $X \neq \emptyset$ a set of variables, $I \subseteq k[X]$ an ideal and $\prec$ a term order. A set $G \subseteq I \setminus \{ 0 \}$ is called a *Gröbner basis* of I with respect to $\prec$ if the cardinality of G is finite and if

$$(\forall f \in I \setminus \{ 0 \})(\exists g \in G)(\mathrm{lt}_\prec(g) \mid \mathrm{lt}_\prec(f)).$$

Let $\prec$ be a term order and let $G \subset k[X]$ be a Gröbner basis of some ideal with respect to $\prec$. Finally, let $f \in k[X]$ be any polynomial. There exists an algorithm for the computation of a normal form p of f with respect to G and $\prec$. It can be shown that p is unique up to multiplication by a constant in k. The reader is referred to [Becker and Weispfenning, 1993, Chapter 5] for further details. From now on we shall write $\mathrm{nf}_\prec(G, f)$ for “the” normal form of f with respect to G and $\prec$ and we shall assume that it is monic or zero.

The set containing the terms of a polynomial f is denoted $\mathrm{terms}(f)$.

Definition 3.38 (Reduced Gröbner Basis). Let k be a field, let $X \neq \emptyset$ be a set of variables, let $I \subseteq k[X]$ be an ideal and let $\prec$ be a term order. A Gröbner basis $G \subseteq I \setminus \{ 0 \}$ of I with respect to $\prec$ is called a *reduced Gröbner basis* of I with respect to $\prec$ if each of its members is monic and

$$(\forall g \in G)(\forall f \in G \setminus \{ g \})(\forall t \in \mathrm{terms}(f))(\mathrm{lt}_\prec(g) \nmid t).$$

Let $\prec$ be a term order and let F be a finite generating system of an ideal I. Given F and $\prec$, the *Buchberger Algorithm* can be used to compute a Gröbner basis G of I with respect to $\prec$. Given G and $\prec$ it is a straightforward exercise to compute a reduced Gröbner basis of I with respect to $\prec$. The reader is referred to [Becker and Weispfenning, 1993, Chapter 5] for further details. For efficient implementations of the Buchberger algorithm the reader may wish to consult [Becker and Weispfenning, 1993, Chapter 5.5] and [Giovini *et al.*, 1991].

3.4 Gröbner Basis Algorithms

3.4.1 Introduction

In this section we shall study Gröbner basis algorithms to solve each of the following problems:

ideal membership problem Given a finite generating system of an ideal $I \subseteq k[X]$ and a polynomial $p \in k[X]$, decide if $p \in I$;

consistency problem Given a finite generating system of ideal $I \subseteq k[X]$, decide if I is consistent;

variable elimination Given a finite generating system of ideal $I \subseteq k[X]$ and a subset W of X, compute a generating system of $I \cap k[W]$;

ideal intersection Given finite generating systems of finitely many ideals, compute a generating system of their intersection;

zero-dimensionality decision problem Given a finite generating system of an ideal I of $k[X]$, decide if I is zero-dimensional;

cardinality Given a finite generating system of zero-dimensional ideal $I \subset k[X]$, compute the cardinality of the variety of I;

extension Given a finite generating system of a zero-dimensional radical ideal $I \subset k[x_1, \ldots, x_n]$, compute a Gröbner basis which contains generating systems of each of the elimination ideals $I \cap k[x_1, \ldots, x_m]$, for $1 \leq m \leq n$. It will be shown that such Gröbner bases allow for the extension of partial solutions of the form $x_1 = v_1, \ldots, x_{m-1} = v_{m-1}$ to partial solutions of the form $x_1 = v_1, \ldots, x_m = v_m$, for $m = 2, \ldots, n$.

We shall consider these problems and the Gröbner basis algorithms to solve them in the following sections.

3.4.2 Ideal Membership

In this section we shall study the *ideal membership problem*, i.e. we shall study the problem of deciding whether a given polynomial $f \in k[X]$ is a member of a given ideal $I \subseteq k[X]$.

Theorem 3.39 (Ideal Membership Problem). *Let $\prec$ be a term order, let G be a Gröbner basis of I with respect to $\prec$, and let $f \in k[X]$. Then $f \in I$ if and only if $\mathrm{nf}_{\prec}(G, f) = 0$.*

The reader is referred to [Becker and Weispfenning, 1993, Theorem 5.55] for proof and further details.

Note that Theorem 3.39 provides an algorithm for the ideal membership problem. To decide if f is in the ideal generated by F, select any term order $\prec$, compute a Gröbner basis G of $\langle F \rangle$ with respect to $\prec$ and compute the normal form of f with respect to $\prec$ and G. Then f is in the ideal generated by F if and only if the normal form is zero.

3.4.3 Consistency of Ideals

In this section we shall briefly discuss an algorithm to decide if ideals are consistent.

Remember that an ideal is called consistent if it does not contain 1 and inconsistent otherwise. The *consistency problem* to decide if the ideal I is consistent is nothing but the ideal membership problem $f \in I$ for the special case $f = 1$. If $\prec$ is a term order, and G is a Gröbner basis of I with respect to $\prec$ then I is consistent if and only if $\mathrm{nf}_{\prec}(G, 1) = 0$.

Reduced Gröbner bases of inconsistent ideals are all equal to $\{ 1 \}$. Reduced Gröbner bases of consistent ideals do not contain 1. If the reduced Gröbner basis of I is available this makes it even easier to decide the consistency problem of I.

3.4.4 Elimination Ideals

In this section we shall study the problem of how to compute generating systems of elimination ideals.

Let $X = \{ x_1, \ldots, x_n \}$ be a non-empty set of variables, let $I \subseteq k[X]$ be an ideal, and let $\prec$ be the lexicographical term order such that $x_i \prec x_j \iff i < j$, for $1 \leq i, j \leq n$. Furthermore, let G be a Gröbner basis of I with respect to $\prec$. Finally, let m be any positive integer less than or equal to n and let f be any non-zero member of $I \cap k[x_1, \ldots, x_m]$. By definition, G contains a member whose leading term with respect to $\prec$ divides the leading term of f with respect to $\prec$. What is more, if $g \in G$ and if $\mathrm{lt}_{\prec}(g) \mid \mathrm{lt}_{\prec}(f)$ then each of the terms in $g - \mathrm{lm}_{\prec}(g)$ are smaller (with respect to $\prec$) than $\mathrm{lt}_{\prec}(g)$ and it follows that $g \in I \cap k[x_1, \ldots, x_m]$. Clearly, G contains Gröbner bases of each of the elimination ideals $I \cap k[x_1, \ldots, x_m]$, for $1 \leq m \leq n$.

Variable elimination has become a straightforward exercise. To eliminate the variables $W \subset X$ from I compute a Gröbner basis of I with respect to any lexicographical term order $\prec$ which has the property that the variables in $X \setminus W$ are the least significant ones. Next eliminate from the Gröbner basis the non-constant polynomials which are in $k[X]$ but not in $k[X \setminus W]$. The resulting set is a Gröbner basis of $I \cap k[X \setminus W]$ with respect to $\prec$.

3.4.5 Ideal Intersection

In this section we shall provide an algorithm to compute the intersection of ideals. The reader is referred to [Becker and Weispfenning, 1993, Corollary 6.20] for a proof.

The following theorem describes a relationship between a set of ideals and the intersection of its members.

Theorem 3.40 (Intersection of Ideals). *Let $X \neq \emptyset$ be a finite set of variables, and let $I = \{ I_1, \ldots, I_n \}$ be a non-empty set of ideals of $k[X]$. Furthermore, let $Y = \{ y_1, \ldots, y_n \}$ be a set of variables such that Y and X are disjoint. Then*

$$\bigcap_{I_i \in I} I_i = k[X] \cap S,$$

where

$$S = \left\langle 1 - \sum_{i=1}^{n} y_i \right\rangle + \sum_{i=1}^{n} y_i I_i.$$

With the tools presented so far, this makes it a trivial exercise to compute a generating system of the intersection of a set of ideals. First compute a generating system of S and then use the algorithm sketched in Section 3.4.4 to eliminate from S the variables that are in Y.

Example 3.41 (Ideal Intersection). Let $V = \{ (1,1), (4,2) \} \subset k^2$. Note that V can also be interpreted as a constraint on two variables, say x and y. In this example we shall show how to construct a generating system of the vanishing ideal of V in $k[x,y]$.

The point $(1,1)$ corresponds to the simultaneous "assignment" $x = 1$ and $y = 1$, i.e. it corresponds to the maximal ideal $\langle x-1, y-1 \rangle$. Similarly, $(4,2)$ corresponds to the maximal $\langle x-4, y-2 \rangle$. Therefore,

$$\begin{aligned}
V &= \{ (1,1), (4,2) \} \\
&= \{ (1,1) \} \cup \{ (4,2) \} \\
&= \mathrm{V}(\langle x-1, y-1 \rangle) \cup \mathrm{V}(\langle x-4, y-2 \rangle) \\
&= \mathrm{V}(\langle x-1, y-1 \rangle \cap \langle x-4, y-2 \rangle) \\
&= \mathrm{V}(\langle F \rangle),
\end{aligned}$$

where F can be computed using Theorem 3.40. The application of Theorem 3.40 and variable elimination with respect to a lexicographical term order $\prec$ such that x and y are the least significant variables and such that $x \prec y$ leads to:

$$F = \left\{ x^2 - 5x + 4, 3y - x - 2 \right\}.$$

It is left as an exercise to the reader to verify that $V = \mathrm{V}(\langle F \rangle)$.

3.4.6 Zero-Dimensional Ideals

In this section we shall present the notion of *reduced terms* of an ideal with respect to a term order, and theorems about and algorithms for zero-dimensional ideals. We shall first present the Triangular Form Theorem. It relates the cardinality of the variety of a zero-dimensional ideal and the leading terms of its Gröbner bases. Next, we shall define the notion of the *reduced terms* of an ideal with respect to a term order. Finally, we shall present the Counting Theorem which provides an algorithm to compute the cardinality of the variety of a zero-dimensional radical ideal by inspecting the leading terms of a Gröbner basis of that ideal.

The following theorem provides an algorithm for the detection of zero-dimensional ideals.

Theorem 3.42 (Triangular Form). *Let $X = \{ x_1, \ldots, x_n \}$, let I be a proper ideal of $k[X]$, and let $\prec$ be any lexicographical term order on the variables in X. Then I is zero-dimensional if and only if for each $x_i \in X$ every Gröbner basis of I with respect to $\prec$ contains a polynomial whose leading term with respect to $\prec$ is of the form $x_i^{\alpha_i}$ for some positive integer α_i.*

The reader is referred to [Becker and Weispfenning, 1993, Theorem 6.54 (i) and (iv)] for proof and further details.

Example 3.43 (Triangular Form). Let $X = \{ x_0, x_1, x_2 \}$, and let $I \subset k[X]$ be the ideal generated by G, where

$$G = \{ x_0, x_1 + x_0, x_2 + x_1 + x_0 \} .$$

G is a Gröbner basis with respect to the lexicographical term order $\prec$ where $x_0 \prec x_1 \prec x_2$. It is not difficult to see that I is zero-dimensional and that $\mathrm{V}(G) = \{ (0, 0, 0) \}$. The leading terms of the members of the Gröbner basis with respect to $\prec$ are given by x_2, x_1 and x_0. For every member x_i of X the basis contains a polynomial whose leading terms with respect to $\prec$ is of the form x_i, i.e. x_i^1. By Theorem 3.42 I is zero-dimensional.

Definition 3.44 (Initial Ideal). Let $\prec$ be a term order, and let I be an ideal of $k[X]$. The *initial ideal* of I with respect to $\prec$ is the ideal which is generated by the leading terms with respect to $\prec$ of the non-zero members of I. The initial ideal of I with respect to $\prec$ is denoted $\mathrm{in}_\prec(I)$. Formally,

$$\mathrm{in}_\prec(I) = \langle \{ \mathrm{lt}_\prec(f) \, : \, f \in I \setminus \{ 0 \} \} \rangle .$$

Definition 3.45 (Reduced Terms). Let $I \subseteq k[X]$ be an ideal and $\prec$ a term order. The *reduced terms* of I with respect to $\prec$ are the terms in $\mathbb{T}_X$ which are not in $\mathrm{in}_\prec(I)$.

It follows from the definition of a Gröbner basis that the reduced terms of I with respect to $\prec$ are the monomials of $k[X]$ that cannot be divided by any of the leading terms (with respect to $\prec$) of the members of G.

The following theorem provides an algorithm for computing the cardinality of the variety of a zero-dimensional radical ideal.

Theorem 3.46 (Counting). *Let k be a any field, let K be the algebraic closure of k, let $\prec$ be a term order, and let I be a zero-dimensional ideal of $k[x_1, \ldots, x_n]$. The number of common zeros of the members of I in K^n is less than or equal to the number reduced terms of I with respect to $\prec$. If k is perfect and I is radical then equality holds.*

The reader is referred to [Becker and Weispfenning, 1993, Theorem 8.32] for proof and further information.

Example 3.47 (Counting). Let $X = \{ x, y \}$, and let $\prec$ be any term order such that $x \prec y$. Furthermore, let

$$\begin{aligned} g_1 &= x^2 - x; \\ g_2 &= y - x; \\ G &= \{ g_1, g_2 \} ; \\ I &= \langle G \rangle . \end{aligned}$$

G is the reduced Gröbner basis of I with respect to $\prec$, G does not contain 1, and therefore I is proper. Note that I contains the square-free univariate polynomial $x(x-1) = x^2 - x$. I also contains the polynomial

$$\begin{aligned} g_1 + g_2(y + x - 1) &= x^2 - x + (y - x)(y + x - 1) \\ &= x^2 - x + y^2 - x^2 - y + x \\ &= y^2 - y \\ &= y(y-1), \end{aligned}$$

which is also square-free. It follows from Lemma 3.20 that I is zero-dimensional and radical. Therefore, Theorem 3.46 can be applied. The reduced terms of I with respect to $\prec$ can be read off from the Gröbner basis G. They are the same as the terms in $\mathbb{T}_X$ which cannot be divided by any of the leading terms of the members of G with respect to $\prec$. They are 1 and x. It follows from Theorem 3.46 that the variety of I contains two members. Indeed, the zeros of x in g_1 are given by 0 and 1 and for each zero of x in g_1 there is exactly one zero for y in g_2. If $x = 0$ then $y = 0$ and if $x = 1$ then $y = 1$. The common zeros of I are $(0, 0)$ and $(1, 1)$. There is one zero for every reduced term of I with respect to $\prec$.

3.4.7 Extension of Solutions

In this section we shall study a proposition which will be used in Chapter 4. The proposition provides a sufficient condition for the extension of partial solutions. Before we present the proposition we recall two theorems which were presented earlier in this chapter.

Remember that Theorem 3.14 (Extension Theorem) provides sufficient conditions for the successful extension of solutions which vanish in elimination ideals to solutions which vanish everywhere. Theorem 3.42 (Triangular Form Theorem) provides an algorithm to decide if proper ideals are zero-dimensional. For every ideal $I \subset k[X]$ and every lexicographical term order $\prec$ the Triangular Form Theorem states that I is zero-dimensional if and only if, for every $x_i \in X$, the reduced Gröbner basis of I with respect to $\prec$ contains a polynomial whose leading term with respect to $\prec$ is equal to $x_i^{\alpha_i}$, for some positive integer α_i.

We can combine these theorems for the following proposition.

Proposition 3.48 (Extension). *If I is a proper zero-dimensional ideal of $k[x_1, \ldots, x_n]$ then any common zero $x_1 = v_1, \ldots, x_{m-1} = v_{m-1}$ of the members of $I \cap k[x_1, \ldots, x_{m-1}]$ can be extended to a common zero $x_1 = v_1, \ldots, x_m = v_m$ of the members of $I \cap k[x_1, \ldots, x_m]$, for $1 \leq m \leq n$.*

Proof. Let $\prec$ be the lexicographical term order such that $x_i \prec x_j \iff i < j$. By the Triangular Form Theorem, for every $1 \leq i \leq n$, the reduced Gröbner basis G of I with respect to $\prec$ contains a polynomial whose leading term with respect to $\prec$ is of the form $x_i^{\alpha_i}$, for some $\alpha_i > 0$. The leading coefficients of these polynomials cannot vanish and by the Extension Theorem the extension of partial solutions in $I \cap k[x_1, \ldots, x_{m-1}]$ to partial solutions in $I \cap k[x_1, \ldots, x_m]$ is guaranteed, for $1 < m \leq n$. $\square$

Chapter 4

CSPs in Solved Form

4.1 Introduction

In this chapter we shall study a new technique to transform any CSP with extensional constraints to a CSP which is in *directionally solved form* with respect to a certain variable ordering and to a CSP which is in *globally solved form*. Both kinds of CSPs guarantee backtrack-free search and guarantee that all solutions can be found without encountering "dead-ends."

The process of transforming a CSP to a CSP which is in directionally solved form consists of three steps. The algorithm for the computation of CSPs in globally solved form is almost similar, except for the last step. The three steps rely heavily on the relationship between constraints, varieties, ideals, and Gröbner bases. First, the CSP is transformed to a generating system of a polynomial ideal. Next, the system is transformed to the reduced Gröbner basis of its ideal with respect to a lexicographical term order. Finally, the reduced Gröbner basis is transformed to a CSP.

The remainder of this chapter is as follows. In Section 4.2 we shall define the notions of a CSP *in directionally* and *globally solved form*, describe the properties of such CSPs, and discuss the related literature. This is followed by Section 4.3 where we shall recall three theorems from the previous chapter which shall be required in the remainder of this chapter. In Section 4.4 we shall discuss the relationship between constraints, varieties and ideals. We shall present the algorithm for the computation of CSPs in directionally solved form and shall present a proof for the correctness of the algorithm in Section 4.5. Examples of the application of the algorithm to some problems will be presented in Section 4.6. We shall present our concluding remarks and provide suggestions for future work in Section 4.7.

4.2 Basic Definitions

In this section we shall define the notion of a CSP in directionally solved form and that of a CSP in globally solved form, discuss the relevant literature, and mention possible applications.

Definition 4.1 (CSP in Directionally Solved Form). Let $X = \{ x_1, \ldots, x_n \}$ be a non-empty set of variables, let $\mathcal{C} = (X, D, C)$ be a CSP, and let $\prec$ be an ordering on the variables in X such that $x_i \prec x_j \iff i < j$. Then $\mathcal{C}$ is in *directionally solved form with respect to* $\prec$ if either:

- $\mathcal{C}$ is unsatisfiable and $\emptyset = C_{\{ x_1 \}} \in C$; or
- $\mathcal{C}$ is satisfiable, there is a constraint of the form $\emptyset \neq C_{\{ x_1 \}} \in C$, and for all integers i, $1 < i \leq n$, if $(v_1, \ldots, v_{i-1})$ satisfies $\mathcal{C}$ then there exists a member $v_i \in D(x_i)$ such that $(v_1, \ldots, v_i)$ also satisfies $\mathcal{C}$.

A CSP in directionally solved form can be solved efficiently in the sense that no backtracking is required in the process of finding one of its solutions or deciding that no such solution exists. All its solutions can be found without encountering dead-ends by extending partial solutions. If the CSP is unsatisfiable then this can be found out easily by inspecting the unary constraint (node-consistency) on the least significant variable with respect to $\prec$.

Example 4.2 (CSP in Directionally Solved Form). Let $X = \{ x_0, x_1, x_2, x_3 \}$, let $\prec$ be the order such that $x_i \prec x_j \iff i < j$, and let (X, D, C) be the CSP, where

$$\begin{aligned}
C &= \left\{ C_{\{ x_0 \}}, C_{\{ x_0, x_1 \}}, C_{\{ x_1, x_2 \}}, C_{\{ x_1, x_2, x_3 \}} \right\};\\
D(x_0) &= \{ 0, 1 \};\\
D(x_1) &= \{ 0, 1 \};\\
D(x_2) &= \{ 0, 1 \};\\
D(x_3) &= \{ 0, 1 \};\\
C_{\{ x_0 \}} &= \{ 0, 1 \};\\
C_{\{ x_0, x_1 \}} &= \{ (0, 0), (0, 1), (1, 1) \};\\
C_{\{ x_1, x_2 \}} &= \{ (0, 0), (1, 1) \};\\
C_{\{ x_1, x_2, x_3 \}} &= \{ (0, 0, 0), (1, 1, 1) \}.
\end{aligned}$$

The CSP is in directionally solved form with respect to $\prec$. The reasons for this are as follows:

1. The smallest variable in X with respect to $\prec$ is x_0. The constraint $C_{\{ x_0 \}}$ is non-empty.
2. Every tuple $(v_0, \ldots, v_{m-1})$ which does not violate any constraint can be extended to a tuple $(v_0, \ldots, v_m)$ which does not violate any constraint, for $1 \leq m \leq 3$. For example, none of the members in $C_{\{ x_0 \}}$ violate any constraint. Every member of $C_{\{ x_0 \}}$ can be extended to an (x_0, x_1)-tuple which does not violate any constraint. The set containing these tuples is $C_{\{ x_0, x_1 \}}$. Every tuple in $C_{\{ x_0, x_1 \}}$ can also be extended to an (x_0, x_1, x_2)-tuple which does not violate any constraint. The set containing all these tuples is given by $\{ (0, 0, 0), (0, 1, 1), (1, 1, 1) \}$. Each member of this set can in its turn be extended to an (x_0, x_1, x_2, x_3)-tuple which does not violate any constraint. The set containing these tuples is given by $\{ (0, 0, 0, 0), (0, 1, 1, 1), (1, 1, 1, 1) \}$. The members of this set are the solutions of (X, D, C).

The purpose of a backtrack-free search is to compute a single solution without encountering a dead-end or to decide that no such solution exists. Freuder provides sufficient conditions and an algorithm for the case where the constraint-graph of the CSP is a tree [Freuder, 1982]. He generalises this for arbitrary binary CSPs by relating the width of the constraint-graph to the level of (i, j)-consistency of the CSP [Freuder, 1985].

As observed by Dechter and Pearl weaker properties may also ensure backtrack-free search [Dechter and Pearl, 1988a]. They propose directional consistency methods for binary CSPs.

Dechter and Van Beek seem to have been the first to pose and answer the question of how to compute directionally solved CSPs [Dechter and van Beek, 1995; 1997]. They present an algorithm called DRC (Directional-Relational-Consistency) which can transform *any* CSP to an equivalent CSP which is in directionally solved form. Their CSPs are created by the repeated addition of constraints and repeated restriction of constraints by removing those partial solutions that cannot be extended. A CSP is in *globally solved form* if it is in directionally solved form with respect to all variables orders. Dechter and Van Beek also present algorithm ARC (Adaptive-Relational-Consistency) to compute CSPs in globally solved form [Dechter and van Beek, 1995].

Besides differences in the domain of computation—we have to translate to and from the polynomial ring, whereas Dechter and Van Beek do not—the main difference is that Dechter and van Beek repeatedly intersect, join, project, and add constraints, whereas we use a Gröbner basis approach. Further on in this chapter we shall demonstrate that by changing one step in our algorithm we can compute CSPs which are in globally solved form.

4.3 Related Mathematics

In this section we shall briefly recall three theorems from Chapter 3 upon which we shall heavily rely in the remainder of this chapter. The first theorem is the Triangular Form Theorem (Theorem 3.42). It provides an algorithm for the detection of zero-dimensional ideals. The second theorem is the Counting Theorem (Theorem 3.46). It can be used to determine the cardinality of the variety of a zero-dimensional radical ideal. The third theorem is the *Extension Theorem* (Theorem 3.14). Given an *elimination order* it provides a sufficient condition for the extension of (partial) solutions for the first $n-1$ variables to (partial) solutions for the first n variables.

4.3.1 Triangular Form Theorem

The Triangular Form Theorem (Theorem 3.42) provides a relationship between zero-dimensional ideals and the leading terms of the members of a Gröbner basis with respect to a lexicographical term order of that ideal. If G is the Gröbner basis of some ideal $I \subseteq k[x_1, \ldots, x_n]$ with respect to a lexicographical term order $\prec$ then I is zero-dimensional if and only if for $1 \leq i \leq n$, G contains a polynomial whose leading term with respect to $\prec$ is of the form $x_i^{\alpha_i}$, for some $\alpha_i > 0$.

4.3.2 Counting Theorem

The Counting Theorem (Theorem 3.46) relates the cardinality of the variety of a zero-dimensional radical ideal and the reduced terms of the ideal with respect to a term order. If $\prec$ is a term order then the number of common zeros of the members of a zero-dimensional radical ideal I of $k[X]$ is equal to the number of reduced terms of I with respect to $\prec$. This number is equal to the number of terms in $\mathbb{T}_X$ that cannot be divided by any of the leading terms with respect to $\prec$ of the members of the Gröbner basis of I with respect to $\prec$.

4.3.3 Extension Theorem

The Extension Theorem (Theorem 3.14) provides a sufficient condition to ensure the extension of partial solutions. As a special case (Proposition 3.48) it guarantees that if I is a proper zero-dimensional ideal of $k[x_1, \ldots, x_n]$ then any common zero $(\, v_1, \ldots, v_{m-1} \,)$ of $I \cap k[x_1, \ldots, x_{m-1}]$ can be extended to a common zero $(\, v_1, \ldots, v_m \,)$ of $I \cap k[x_1, \ldots, x_m]$, for $1 < m \leq n$.

4.4 Ideals, Varieties, and Constraints

This section uses the ideal-variety correspondence discussed in Section 3.2.7 to translate finite constraints to varieties and polynomial ideals and vice versa. The theory to be presented allows for the transformation of finite CSPs to polynomial ideals and vice versa. The presentation is of an informal nature. The reader is referred to [Cox *et al.*, 1996, Chapter 4] for a more formal presentation.[1] As usual, k denotes the field of the complex numbers.

In the following, let $X = \{\, x_1, \ldots, x_n \,\}$ be a finite set of variables and $(\, X, D, C \,)$ a finite CSP. As usual we shall assume that the order on the variables is such that x_i precedes x_j if and only if $i < j$. Without loss of generality we shall assume that the domains of the variables in X are subsets of k.

It is recalled that a proper ideal I of ring R is called a *maximal* ideal of R if $I + J \in \{\, I, R \,\}$ for every ideal J of R. Maximal ideals of $k[x_1, \ldots, x_n]$ are of the form $\langle\, x_1 - v_1, \ldots, x_n - v_n \,\rangle$, for suitably chosen $v_1, \ldots, v_n \in k$. Their varieties are of the form $\{\, (\, v_1, \ldots, v_n \,) \,\}$, i.e. each such ideal corresponds to a single point in k^n. Note that maximal ideals of $k[X]$ are radical.

Let n be a positive integer, let $X = \{\, x_1, \ldots, x_n \,\}$, let S be a non-empty set of variables, and let C_S be a non-empty constraint in C. Every $(\, v_{i_1}, \ldots, v_{i_m} \,) \in C_S$ corresponds to some point $(\, v_{i_1}, \ldots, v_{i_m} \,)$ of k^m and hence with the maximal (as well as radical) ideal

$$\langle\, x_{i_1} - v_{i_1}, \ldots, x_{i_m} - v_{i_m} \,\rangle$$

of $k[S]$. C_S can therefore be described as follows:

$$C_S = \bigcup_{(\, v_{i_1}, \ldots, v_{i_m} \,) \in C_S} \mathrm{V}\left(\langle\, x_{i_1} - v_{i_1}, \ldots, x_{i_m} - v_{i_m} \,\rangle\right). \tag{4.1}$$

[1]Note that [Cox *et al.*, 1996, Chapter 4] does not cover constraints but *does* cover the relationship between ideals and varieties in great detail.

Equation (4.1) states that C_S is the union of finitely many varieties of k^m. The union of varieties is again a variety. Therefore, C_S is a variety of k^m. Remember that the union of the varieties of ideals is equal to the variety of the intersection of those ideals (Theorem 3.26). Let $R = k[X]$, let $\emptyset \subset S \subseteq X$, and let J be an ideal of $k[S]$. It is recalled that $J_R \subseteq k[X]$ is the R-module of ideal J. With these definitions, Equation (4.1) is tantamount to:

$$C_S = \mathrm{V}\left(I\right),$$

where $I \subset k[S]$ is given by:

$$I = (\bigcap_{v \in C_S} \mathrm{I}(\{\,v\,\}))_R.$$

Note that I is the intersection of radical ideals. By Theorem 3.21, I is radical. C_S is non-empty, and by Hilbert's Weak Nullstellensatz (Theorem 3.7), I is consistent. Since $\mathrm{V}\left(I\right) = C_S$, I is zero-dimensional. Let $V_S \subset k^n$ be the variety of I_R.

From now, for every $C_S \in C$, let V_S denote the variety in k^n which can be constructed from the constraint C_S as laid out in the previous paragraph, i.e. let

$$V_S = \mathrm{V}\left(\cap_{(\,v_{i_1},\ldots,v_{i_m}\,) \in C_S} \langle\, x_{i_1} - v_{i_1}, \ldots, x_{i_m} - v_{i_m}\,\rangle_R\right).$$

From now on V_S will be called the *variety* of the constraint C_S. The set $\mathcal{S} \subset k^n$ of values which satisfy the CSP that we started with is equal to the intersection of the varieties of the constraints in C:

$$\mathcal{S} = \bigcap_{C_S \in C} V_S. \tag{4.2}$$

It is recalled from Section 3.2.7 that the variety of the sum of ideals is equal to the intersection of the varieties of these ideals. Therefore, Equation (4.2) is equivalent to

$$\mathcal{S} = \mathrm{V}\left(\sum_{C_S \in C} \mathrm{I}(V_S)\right). \tag{4.3}$$

Let $J = \mathrm{I}(\mathcal{S}) \subseteq k[X]$. It is recalled from Chapter 2 that without loss of generality we can assume that $X = \cup_{C_S \in C} S$. By Proposition 3.22, J is inconsistent or zero-dimensional and radical.

Example 4.3 (Constraint/Variety/Ideal Relationship). Let $X = \{\,x, y\,\}$, let $D(x) = C_{\{\,x\,\}} = \{\,-2, -1, 0, 1\,\}$, let $D(y) = C_{\{\,y\,\}} = \{\,1, 2, 3, 4\,\}$, let $C_{\{\,x,y\,\}} = \{\,(\,1, 1\,), (\,-2, 4\,)\,\}$, and let $C = \left\{\,C_{\{\,x\,\}}, C_{\{\,y\,\}}, C_{\{\,x,y\,\}}\,\right\}$. Finally, let $\mathcal{C} = (\,X, D, C\,)$ be a CSP. The constraint $C_{\{\,x\,\}}$ corresponds to the variety $V_{\{\,x\,\}} = \{\,-2, -1, 0, 1\,\} \times k$, i.e. it is the set of $(\,x, y\,)$-tuples where $x \in \{\,-2, -1, 0, 1\,\}$ and $y \in k$. It follows directly from the relationship between unions of varieties and the variety of the intersection of their ideals that:

$$\begin{aligned}
C_{\{\,x\,\}} &= \{\,-2, -1, 0, 1\,\} \\
&= \mathrm{V}\left(\langle\, x + 2\,\rangle\right) \cup \mathrm{V}\left(\langle\, x + 1\,\rangle\right) \cup \mathrm{V}\left(\langle\, x\,\rangle\right) \cup \mathrm{V}\left(\langle\, x - 1\,\rangle\right) \\
&= \mathrm{V}\left(\langle\, x + 2\,\rangle \cap \langle\, x + 1\,\rangle \cap \langle\, x\,\rangle \cap \langle\, x - 1\,\rangle\right) \\
&= \mathrm{V}\left(\langle\,(x + 2)(x + 1)(x - 0)(x - 1)\,\rangle\right).
\end{aligned}$$

Therefore,

$$V_{\{x\}} = \mathrm{V}\left(\left\langle G_{\{x\}} \right\rangle_R\right) \subset k^2,$$

where $G_{\{x\}}$ is given by

$$G_{\{x\}} = \{ (x+2)(x+1)(x-0)(x-1) \} .$$

Similarly,

$$V_{\{y\}} = \mathrm{V}\left(\left\langle G_{\{y\}} \right\rangle_R\right) \subset k^2,$$

where $G_{\{y\}}$ is given by

$$G_{\{y\}} = \{ (y-1)(y-2)(y-3)(y-4) \} .$$

Using the same technique we can compute a generating system for the ideal of $V_{\{x,y\}}$ as follows:

$$\begin{aligned} V_{\{x,y\}} &= \{ (1,1), (-2,4) \} \\ &= \mathrm{V}\left(\langle x-1, y-1 \rangle_R \cap \langle x+2, y-4 \rangle_R\right) . \end{aligned}$$

We shall use the algorithms proposed in Section 3.4.4 and Section 3.4.5 to compute the intersection of $\langle x-1, y-1 \rangle_R$ and $\langle x+2, y-4 \rangle_R$. The computation of the intersection proceeds as follows. Let z_1 and z_2 be two new variables and let $\prec$ be the lexicographical term order such that $x \prec y \prec z_1 \prec z_2$. It is recalled from Section 3.4.5 that if I and J are two ideals of $k[x,y]$ then

$$I \cap J = k[x,y] \cap (\langle 1 - z_1 - z_2 \rangle + z_1 I + z_2 J).$$

The reduced Gröbner basis of

$$\langle 1 - z_1 - z_2 \rangle + z_1 \langle x-1, y-1 \rangle + z_2 \langle x+2, y-4 \rangle$$

with respect to $\prec$ is given by

$$\left\{ x^2 + x - 2, y + x - 2, z_1 - x/3 - 2/3, z_2 + x/3 - 1/3 \right\} .$$

Therefore,

$$\langle x-1, y-1 \rangle \cap \langle x-2, y+4 \rangle = \left\langle x^2 + x - 2, y + x - 2 \right\rangle .$$

This allows us to conclude that

$$V_{\{x,y\}} = \mathrm{V}\left(\left\langle G_{\{x,y\}} \right\rangle_R\right), \tag{4.4}$$

where $G_{\{x,y\}}$ is given by

$$\left\{ x^2 + x - 2, y + x - 2 \right\} .$$

Equation (4.2) states that the solutions $\mathcal{S}$ of $\mathcal{C}$ are given by the intersection of the varieties of the constraints in C. This is equivalent to Equation (4.3) which states that $\mathcal{S}$ is equal to the variety of the sum of the ideals of the varieties of the constraints in C. Let J be the sum of the ideals of the varieties of the constraints of C, i.e. $J = \sum_{C_S \in C} \mathrm{I}(V_S)$. Furthermore, let $\prec$ be the lexicographical term order, such that $x \prec y$. The Gröbner basis of J with respect to $\prec$ is given by:

$$\{ x^2 + x - 2, y + x - 2 \} .$$

It is the same as the generating system of the ideal of $V_{\{ x,y \}}$ from Equation (4.4). It follows immediately that $\mathcal{S} = C_{\{ x,y \}}$ and that $\mathcal{S}$ are the solutions of $\mathcal{C}$.

Notice that the R-module of the ideal generated by the Gröbner basis contains $\langle G_{\{ x \}} \rangle_R$, $\langle G_{\{ y \}} \rangle_R$, and $\langle G_{\{ x,y \}} \rangle_R$. It contains $\langle G_{\{ x \}} \rangle_R$ because $x^2 + x - 2 = (x+2)(x-1)$ divides $(x+2)(x+1)(x-0)(x-1)$. It contains $\langle G_{\{ y \}} \rangle_R$ because

$$\begin{aligned} \langle x^2 + x - 2, y + x - 2 \rangle &= \langle (2-y)^2 + (2-y) - 2, y + x - 2 \rangle \\ &= \langle y^2 - 5y + 4, y + x - 2 \rangle \\ &= \langle (y-4)(y-1), y + x - 2 \rangle \end{aligned}$$

and because $(y-4)(y-1)$ divides $(y-1)(y-2)(y-3)(y-4)$. It is left as an exercise to the reader to prove that $\langle G_{\{ x,y \}} \rangle_R$ is also contained by the R-module of the ideal which is generated by the Gröbner basis.

In the previous paragraphs we have demonstrated how to translate a CSP into a generating system of an ideal whose variety is equal to the solutions of the CSP. The following proposition suggests an algorithm to get back from the generating system of an ideal to a CSP whose solutions are equal to the common zeros of the ideal.

Proposition 4.4. *Let $X = \{ x_1, \ldots, x_n \}$ be a finite set of variables such that $D(x_i)$ has a finite cardinality for each $x_i \in X$. Let $F \subset k[X]$ be a finite set of polynomials. Then there exists an algorithm to compute $\mathrm{V}(F) \cap \times_{x_i \in X} D(x_i)$.*

Proof. First observe that $(v_1, \ldots, v_n) \in \mathrm{V}(F) \cap \times_{x_i \in X} D(x_i)$ if and only if $(v_1, \ldots, v_n) \in \times_{x_i \in X} D(x_i)$ and each of the polynomials in F vanishes at $(v_1, \ldots, v_n)$. Next observe that $\times_{x_i \in X} D(x_i)$ has a finite cardinality. The problem of finding the algorithm has been reduced to the enumeration of the members of $\times_{x_i \in X} D(x_i)$ and a finite sequence of tests to see if these members are the zeros of the members of a finite set of polynomials. □

4.5 An Algorithm for CSPs in Directionally Solved Form

This section describes a transformation technique from any CSP with extensional constraints to an equivalent CSP which is in directionally solved form with respect to some ordering on the variables. The resulting CSP corresponds to a reduced Gröbner basis with respect to a lexicographical term order.

In the following, let n be a positive integer, let $X = \{ x_1, \ldots, x_n \}$, let (X, D, C) be a finite CSP, let $\prec$ be the lexicographical term order such that $x_1 \prec \cdots \prec x_n$, and let $\mathcal{S}$ denote the solution set of the CSP. It is recalled from the previous section that every constraint C_S

corresponds to some variety $V_S \in k^n$. The set of solutions $\mathcal{S}$ of the CSP can be described as the variety of the sum of the ideals of the varieties V_S, i.e.

$$\mathcal{S} = \mathrm{V}\left(\sum_{C_S \in C} \mathrm{I}(V_S)\right).$$

The transformation is given by:

1. (a) For each $C_S \in C$ compute a generating system $B_S \subset k[x_1, \ldots, x_n]$ for $\mathrm{I}(V_S)$. After this step we have:

$$\mathcal{S} = \mathrm{V}\left(\sum_{C_S \in C} \langle\, B_S \,\rangle\right).$$

   (b) Let $B_X = \bigcup_{C_S \in C} B_S$. After this step we have:

$$\mathcal{S} = \mathrm{V}\left(\langle\, B_X \,\rangle\right).$$

2. (a) Compute the reduced Gröbner basis G_X of $\langle\, B_X \,\rangle$ with respect to $\prec$. We now have:

$$\mathcal{S} = \mathrm{V}\left(\langle\, G_X \,\rangle\right).$$

3. (a) For each polynomial g occurring in G_X, let S_g denote its variables. Compute the maximal (with respect to inclusion) subset B'_{S_g} of polynomials in G_X the variables of which are given by S_g. After this step we have:

$$\mathcal{S} = \mathrm{V}\left(\sum_{g \in G_X} \left\langle\, B'_{S_g} \,\right\rangle\right).$$

   (b) If $G_X = \{\,1\,\}$ then set $C'_{\{\,x_1\,\}} = \emptyset$ and C' to $\left\{\, C'_{\{\,x_1\,\}} \,\right\}$. Otherwise, for each B'_{S_g} computed in the previous step compute:

$$C'_{S_g} = \mathrm{V}\left(B'_{S_g}\right) \cap \bigtimes{}_{x_i \in S_g} D(x_i),$$

   where $\times$ is the Cartesian product operator. Set $C' = \left\{\, C'_{S_g} \;:\; g \in G_X \,\right\}$. Finally, we have

$$\mathcal{S} = \mathrm{V}\left(\sum_{C'_{S_g} \in C'} \langle\, {C'}_{S_g} \,\rangle\right).$$

The resulting CSP corresponds to C'.

The bases B_S in step 1.a can be computed by intersecting ideals. The constraints C'_{S_g} can be computed by the algorithm suggested by Proposition 4.4. If $W \neq \emptyset$ is a variety with a finite cardinality, then $I = \langle W \rangle$ is a zero-dimensional radical ideal. Theorem 3.14 (Extension Theorem) and Theorem 3.42 (Triangular Form Theorem) guarantee that extending non-empty partial solutions of elimination ideals of I must succeed. Similarly, the existence of a unary constraint for the smallest variable is guaranteed. Therefore, the CSP is in directionally solved form with respect to $\prec$.

In this paragraph we shall describe how to compute CSPs which are in globally solved form. The change is very simple. Replace Step (2) by: "Compute a universal Gröbner basis of $\langle B_X \rangle$." Here, a *universal Gröbner basis* of an ideal I is a set which is a Gröbner basis of I with respect to any term order. The interested reader is referred to [Becker and Weispfenning, 1993, pp. 514–515] for a short introduction to universal Gröbner bases and to [Mora and Robbiano, 1988] for more detailed information.

Gröbner bases are difficult to compute. Given a generating system F of a zero-dimensional ideal it requires (worst case) $\mathbf{O}\left(d^{\mathbf{O}(n)}\right)$ time to compute the Gröbner basis of the ideal generated by F, where d is the maximum total degree of a monomial in F and n the number of variables. However, the following observations may be made:

- The techniques presented work in any ring $k[X]$ if k is an algebraically closed field. Changing from an algebraically closed field to a finite field will not affect any of the results if the field is sufficiently large to encode the members of the largest domain. The reasons for this are two-fold. First, it can be shown that finite fields are perfect (See [Becker and Weispfenning, 1993, Corollary 7.3]). Therefore, Lemma 3.20 remains valid. The second reason is as follows. Our application of theorems (including the Counting Theorem) are specialised for the case where the field of computation is algebraically closed. For the case where the field of computation is finite our results still hold because all our ideals are radical by construction and our operations (intersection and addition of ideals) do not introduce zeros "outside" the field.

  If p is a prime then $\mathbb{F}_p = \mathbb{Z}/\langle p \rangle$ is a finite field containing p members [Cox *et al.*, 1997, page 359]. The method remains valid for the choice of $\mathbb{F}_p[X]$ if p is a small prime greater than or equal to the maximum domain size. This will avoid large (intermediate) coefficients and should speed up the computation significantly.

- Computing CSPs in directionally solved form corresponds to finding all solutions of a CSP which is also a difficult problem. As a matter of fact, the problem of finding all solutions of the CSP has a worst-case time-complexity $\mathbf{O}(d^n)$, where d is the largest domain size.

  It is not difficult to show that if the maximum domain size is d and if ideal intersection is used to compute the generating systems of the varieties of the constraints then the maximum degree of the polynomials in the generating systems of the bases will be d as well. This does not exclude the possibility that the bounds $\mathbf{O}\left(d^{\mathbf{O}(n)}\right)$ and $\mathbf{O}(d^n)$ coincide for the algorithm presented in this section.

- We could use Proposition 3.32 to compute the bases B_S in step 1 (a) and 1 (b). This should make it easier to compute the generating systems of the ideals of the constraints and (perhaps) the Gröbner basis to be computed in Step 2.

For example, let $C_{\{x,y\}} = \{ (0,0), (1,0), (0,1) \}$, let $G_1 = \{ x, y \}$, let $G_2 = \{ x-1, 0 \}$, let $G_3 = \{ x, y-1 \}$, and let I be the ideal of $C_{\{x,y\}}$. Then

$$\begin{aligned} I &= \langle G_1 \rangle \cap \langle G_2 \rangle \cap \langle G_3 \rangle \\ &= \langle x, y \rangle \cap \langle x-1, y \rangle \cap \langle x, y-1 \rangle \\ &= (\langle x \rangle + \langle y \rangle) \cap (\langle x-1 \rangle + \langle y \rangle) \cap (\langle x \rangle + \langle y-1 \rangle) \\ &= (\langle x \rangle \cap \langle x-1 \rangle + \langle y \rangle) \cap (\langle x \rangle + \langle y-1 \rangle) \\ &= \langle x \rangle \cap \langle x-1 \rangle + \langle x \rangle \cap \langle y \rangle + \langle y \rangle \cap \langle y-1 \rangle \\ &= \langle x(x-1) \rangle + \langle xy \rangle + \langle y(y-1) \rangle \\ &= \langle x(x-1), xy, y(y-1) \rangle . \end{aligned}$$

Let $G = \{ x(x-1), xy, y(y-1) \}$. G_1, G_2 and G_3 are universal Gröbner bases and so is G. The structure of G is very similar to the structure of G_1, G_2 and G_3. Unfortunately, in general the construction does not always lead to such nice bases and it can lead to sets which are not Gröbner bases with respect to any term order.

Note that the construction is very similar to algorithms for the transformation of a formula $\mathcal{F}_1$ in disjunctive normal form to a formula $\mathcal{F}_2$ which is equivalent to $\mathcal{F}_1$ and is in conjunctive normal form. For example, every member (v_i, w_j) of $C_{\{x,y\}}$ corresponds to a (maximal) conjunction in $\mathcal{F}_1$ of the form:

$$V_i \wedge W_j,$$

and vice versa. For each member of the generating system of the form:

$$(x - v_{i_1}) \cdots (x - v_{i_m})(y - w_{j_1}) \cdots (y - w_{j_n})$$

there is a (maximal) disjunction of the form:

$$V_{i_1} \vee \cdots \vee V_{i_m} \vee W_{j_1} \vee \cdots \vee W_{j_n}$$

in $\mathcal{F}_2$ and vice versa.

The advantage of the construction is that it avoids the need for the algorithm for ideal intersection as described in Section 3.4.5. Thus it eliminates the need to compute the lexicographical (read difficult) Gröbner bases which are required for the variable elimination part of that intersection algorithm.

4.6 Example Applications

This section presents two examples where CSPs are transformed to their equivalents in directionally solved form using the technique described in the previous section.

Example 4.5 (Traffic Lights). The following constraints model a set of German traffic lights and are based on [Hower, 1995].

$$C_{\{v_i,p_i,v_{i+1 \bmod 4},p_{i+1 \bmod 4}\}} = \{(r,r,g,g),(r_y,r,y,r),(g,g,r,r),(y,r,r_y,r)\};$$
$$C_{\{v_i\}} = \{r,g,r_y,y\};$$
$$C_{\{p_i\}} = \{r,g\},$$

for $i \in \{0,1,2,3\}$. The eight variables are given by p_0, p_1, p_2, p_3, v_0, v_1, v_2, and v_3. The variables p_i correspond to pedestrian lights. The remaining variables are vehicle lights. There are four 4-ary constraints $C_{\{v_0,p_0,v_1,p_1\}}$, $C_{\{v_1,p_1,v_2,p_2\}}$, $C_{\{v_2,p_2,v_3,p_3\}}$, $C_{\{v_3,p_3,v_0,p_0\}}$, and eight unary constraints corresponding to a domain of each of the variables. The macro-structure of the CSP is depicted in Figure 4.1. Every variable x is represented by the circle containing x. The 4-ary

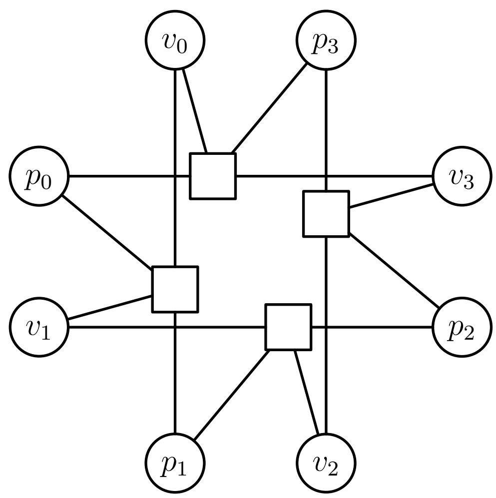

Figure 4.1: Macro-structure of original CSP.

constraint C_S is represented by the square which is connected to the variables in S by straight lines. Assume $g = 0, r_y = 1, y = 2, r = 3$. Computing the generating systems for the constraints with the algorithm as described in Section 3.4.5 results in the following systems:

$$\begin{aligned}
&B_{\{v_i,p_i,v_{i+1 \bmod 4},p_{i+1 \bmod 4}\}} \\
&\quad = \{ v_{i+1 \bmod 4}^4 - 6v_{i+1 \bmod 4}^3 + 11v_{i+1 \bmod 4}^2 - 6v_{i+1 \bmod 4} \\
&\qquad , v_i + v_{i+1 \bmod 4} - 3 \\
&\qquad , p_{i+1 \bmod 4} - v_{i+1 \bmod 4}^3 + 6v_{i+1 \bmod 4}^2 - 11v_{i+1 \bmod 4} \\
&\qquad , p_i + v_{i+1 \bmod 4}^3 - 3v_{i+1 \bmod 4}^2 + 2v_{i+1 \bmod 4} - 6 \\
&\qquad \} \\
&B_{\{v_i\}} = \{ v_i^4 - 6v_i^3 + 11v_i^2 - 6v_i \}; \\
&B_{\{p_i\}} = \{ p_i^2 - 3p_i \},
\end{aligned}$$

for $i \in \{0,1,2,3\}$.

The union B_X of these generating systems consists of 12 binomials and 8 monomials. Let $\prec$ be the lexicographical term order given by $v_3 \prec v_2 \prec v_1 \prec v_0 \prec p_3 \prec p_2 \prec p_1 \prec p_0$. The reduced Gröbner basis G_X of $\langle B_X \rangle$ with respect to $\prec$ is given by:

$$
\begin{array}{l}
\{\, v_3^4 - 6v_3^3 + 11v_3^2 - 6v_3 \\
,\ v_2 + v_3 - 3 \\
,\ v_1 - v_3 \\
,\ v_0 + v_3 - 3 \\
,\ 2p_3 - v_3^3 + 6v_3^2 - 11v_3 \\
,\ 2p_2 + v_3^3 - 3v_3^2 + 2v_3 - 6 \\
,\ 2p_1 - v_3^3 + 6v_3^2 - 11v_3 \\
,\ 2p_0 + v_3^3 - 3v_3^2 + 2v_3 - 6 \\
\}.
\end{array}
$$

The reduced Gröbner basis has revealed a structure which was implicit in the original CSP. The basis does not equal $\{\, 1 \,\}$. By Hilbert's Weak Nullstellensatz (Theorem 3.7) the CSP is satisfiable.

It is recalled that Proposition 3.22 ensures that the construction of the generating bases of the constraints using ideal intersection results in generating systems of radical ideals. Proposition 3.22 guarantees that the sum of zero-dimensional radical ideals is either inconsistent or zero-dimensional and radical. It is because of the latter that the Counting Theorem (Theorem 3.46) can be applied to count the number of common zeros of the members of the sum. The theorem guarantees that the number of zeros of a zero-dimensional radical ideal is equal to the number of reduced terms of its ideal, where the reduced terms are those monomials that are not divisible by any of the leading terms of the reduced Gröbner basis of that ideal. The reduced terms of $\langle G_X \rangle$ with respect to $\prec$ are given by $\{\, 1, v_3, v_3^2, v_3^3 \,\}$. There are four reduced terms and Theorem 3.46 guarantees that there are exactly four solutions to the CSP.

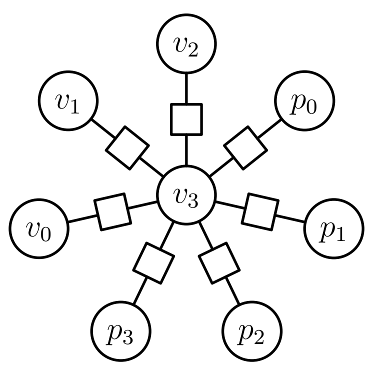

Figure 4.2: Macro-structure of CSP in solved form.

The zeros of each polynomial in the basis correspond to one of the following constraints of the

CSP in directionally solved form with $\prec$ the macro-structure of which is depicted in Figure 4.2:

$$
\begin{aligned}
C'_{\{v_3\}} &= \{g, r_y, y, r\};\\
C'_{\{v_2,v_3\}} &= \{(r,g),(y,r_y),(r_y,y),(g,r)\};\\
C'_{\{v_1,v_3\}} &= \{(g,g),(r_y,r_y),(y,y),(r,r)\};\\
C'_{\{v_0,v_3\}} &= \{(r,g),(y,r_y),(r_y,y),(g,r)\};\\
C'_{\{p_3,v_3\}} &= \{(g,g),(r,r_y),(r,y),(r,r)\};\\
C'_{\{p_2,v_3\}} &= \{(r,g),(r,r_y),(r,y),(g,r)\};\\
C'_{\{p_1,v_3\}} &= \{(g,g),(r,r_y),(r,y),(r,r)\};\\
C'_{\{p_0,v_3\}} &= \{(r,g),(r,r_y),(r,y),(g,r)\}.
\end{aligned}
$$

Example 4.6 (Five Queens Problem). The following constraints model a 5-queens problem, i.e. the problem of positioning five queens on a five by five chessboard such that none of the queens attacks another queen. It is assumed that the i-th queen q_i is placed in the i-th column of the board.

$$
\begin{aligned}
C_{\{q_i\}} &= \{0,1,2,3,4\}, \quad \text{for } i \in \{0,1,2,3,4\};\\
C_{\{q_i,q_j\}} &= \left\{(r_i,r_j) \in \{0,1,2,3,4\}^2 : r_i \neq r_j \wedge (r_i - i)^2 \neq (r_j - j)^2\right\},\\
&\quad \text{for } i \neq j, \text{ and } i,j \in \{0,1,2,3,4\}.
\end{aligned}
$$

The generating bases for the constraints are given by the equations:

$$
B_{\{q_i\}} = \left\{ \prod_{0 \leq j \leq 4} (q_i - j) \right\}, \quad \text{for } i \in \{0,1,2,3,4\}
$$

and

$$
B_{\{q_i,q_j\}} = \left\{ \prod_{\substack{1 \leq k \leq 4 \\ k \neq j-i}} ((q_i - q_j)^2 - k^2) \right\}, \quad \text{for } 0 = i < j \leq 4.
$$

Note that this time the construction of the generating bases is slightly different from the previous example. Strictly speaking, the generating bases of the bivariate ideals are too loose, i.e. they have too many zeros (some of them are “outside” the board). The ideals generated by the univariate polynomials overcome this because they will “remove” these superfluous zeros. Furthermore, these univariate bases ensure that the generating system will be that of a radical ideal.

Let $X = \bigcup_{i=0}^{4} \{q_i\}$ and B_X the union of the generating bases and $\prec$ the lexicographical order where $q_4 \prec q_3 \prec q_2 \prec q_1 \prec q_0$. The reduced Gröbner basis G_X of $\langle B_X \rangle$ with respect to

$\prec$ is given by:

$$
\begin{aligned}
&\{q_4^5 - 10q_4^4 + 35q_4^3 - 50q_4^2 + 24q_4 \\
&,\ 6q_3^2 - 5q_3q_4^3 + 30q_3q_4^2 - 37q_3q_4 - 30q_3 - 10q_4^4 \\
&\quad + 90q_4^3 - 254q_4^2 + 210q_4 + 36 \\
&,\ 24q_2 + 25q_3q_4^4 - 200q_3q_4^3 + 515q_3q_4^2 - 460q_3q_4 \\
&\quad + 72q_3 - 50q_4^4 + 440q_4^3 - 1270q_4^2 + 1264q_4 - 240 \\
&,\ 24q_1 - 25q_3q_4^4 + 200q_3q_4^3 - 515q_3q_4^2 + 460q_3q_4 \\
&\quad - 72q_3 + 50q_4^4 - 420q_4^3 + 1150q_4^2 - 1092q_4 + 120 \\
&,\ 6q_0 + 6q_3 - 5q_4^3 + 30q_4^2 - 37q_4 - 30 \\
&\}.
\end{aligned}
$$

The basis shows that after having positioned q_4 and q_3 at valid locations there is only one possibility to position the remaining three queens. This is demonstrated by the fact that each of the last three polynomials in the basis is *linear* in the variable which is its leading term. The number of reduced terms of the reduced Gröbner basis is $5 \times 2 \times 1 \times 1 \times 1 = 10$.[2] Theorem 3.46 (Counting Theorem) guarantees that there are exactly ten solutions to the 5-queens problem. The CSP in directionally solved form which is equivalent to $(\, X, D, C\,)$ is given by $(\, X, D, C'\,)$, where

$$
\begin{aligned}
C' &= \{\, C'_{\{\,q_4\,\}}, C'_{\{\,q_3,q_4\,\}}, C'_{\{\,q_2,q_3,q_4\,\}}, C'_{\{\,q_1,q_3,q_4\,\}}, C'_{\{\,q_0,q_3,q_4\,\}} \,\}\,; \\
C'_{\{\,q_4\,\}} &= \{\, 0, 1, 2, 3, 4 \,\}\,; \\
C'_{\{\,q_3,q_4\,\}} &= \{\, (\,2,0\,), (\,3,0\,), (\,3,1\,), (\,4,1\,), (\,0,2\,), \\
&\qquad (\,4,2\,), (\,0,3\,), (\,1,3\,), (\,1,4\,), (\,2,4\,) \,\}; \\
C'_{\{\,q_2,q_3,q_4\,\}} &= \{\, (\,4,2,0\,), (\,1,3,0\,), (\,0,3,1\,), (\,2,4,1\,), (\,3,0,2\,), \\
&\qquad (\,1,4,2\,), (\,2,0,3\,), (\,4,1,3\,), (\,3,1,4\,), (\,0,2,4\,) \,\}; \\
C'_{\{\,q_1,q_3,q_4\,\}} &= \{\, (\,1,2,0\,), (\,4,3,0\,), (\,2,3,1\,), (\,0,4,1\,), (\,1,0,2\,), \\
&\qquad (\,3,4,2\,), (\,4,0,3\,), (\,2,1,3\,), (\,0,1,4\,), (\,3,2,4\,) \,\}; \\
C'_{\{\,q_0,q_3,q_4\,\}} &= \{\, (\,3,2,0\,), (\,2,3,0\,), (\,4,3,1\,), (\,3,4,1\,), (\,4,0,2\,), \\
&\qquad (\,0,4,2\,), (\,1,0,3\,), (\,0,1,3\,), (\,2,1,4\,), (\,1,2,4\,) \,\}.
\end{aligned}
$$

Note that originally there were ten binary constraints, whereas the CSP in directionally solved form contains one binary constraint and three ternary constraints.

4.7 Concluding Remarks

In this chapter, we have studied a new technique for the transformation of extensional CSPs to equivalent CSPs in directionally solved form with respect to a certain variable ordering. The resulting CSPs correspond to reduced Gröbner bases for lexicographical term orders.

If the CSP in directionally solved form is satisfiable, a backtrack-free search for the first solution exists. Furthermore all solutions can be found without encountering dead-ends.

[2] The reduced terms are $\{\, t_1 t_2 \,:\, (\,t_1, t_2\,) \in \{\, q_4^4, q_4^3, q_4^2, q_4, 1 \,\} \times \{\, q_3, 1 \,\} \,\}$.

Most of the time needed in the transformation process is spent on the computation of a Gröbner basis of a zero-dimensional ideal. The general problem of computing Gröbner bases is very difficult. Computing such bases for zero-dimensional ideals is much easier in practice. Suggestions have been presented on how to improve the algorithm.

Chapter 5

The Geometry of Constraints

5.1 Introduction

This chapter presents tools to analyse and disect constraints. The tools are the building blocks for a new *generalised backtracking algorithm* which is a generalisation of the well known chronological backtracking algorithm. Generalised backtracking is sound and complete.

The motivation for the generalised backtracking algorithm is as follows. It has been observed in the mathematical community that a solution strategy for systems of multivariate polynomial equations where Gröbner bases with respect to total degree orders are factorised and the induced problems are solved is to be preferred to a strategy where lexicographical Gröbner bases (elimination ideals/strict "lexicographical" rules) dictate the order in which equations should be used to decompose the problem [Boege *et al.*, 1986; Czapor, 1989; Melenk, 1990; 1993; Gräbe, 1994; Pesch, 1996]. The reasons are two-fold. Firstly, the Gröbner bases with respect to total degree orders are (normally) easier to compute. Secondly, the polynomials occurring in the total degree bases (normally) have a lower degree thereby leading to factors of lower degree. Gräbe furthermore observes that problems coming from real life often fulfill the condition of being factorisable [Gräbe, 1994]. The strategy used by the chronological backtracking algorithm is similar to the "lexicographical" approach mentioned before. The generalised backtracking algorithm on the other hand is motivated by similar observations as the "total degree" approach. To be more specific, the generalised backtracking algorithm is not restricted to the use of unary constraints (the domains of the variables) alone to decompose problems.

Both the chronological backtracking algorithm and the generalised backtracking algorithm traverse search trees. The chronological backtracker decomposes problems at each internal node of the search tree by considering a unary constraint (the domain of a variable). For each member in the unary constraint it creates a sub-problem. The number of sub-problems that have to be considered is equal to the cardinality of the unary constraint. In terms of tree traversals the unary constraint determines the (local) number of branches the sub-trees of which the chronological backtracker has to traverse. This number is called the *local branching factor*. As already indicated, the generalised backtracking algorithm can use *any* kind of constraint to obtain a problem decomposition. It will be demonstrated that this will never result in a higher local branching factor but may result in a lower local branching factor in return for a marginal increase in the space complexity.

The chronological backtracking algorithm has received much attention from many researchers. Variants of the algorithm range from a vanilla version [Golomb and Baumert, 1965], to forward checking [Haralick and Elliott, 1980] and MAC [Sabin and Freuder, 1994], and to backjumping [Gaschnig, 1978], conflict directed backjumping [Prosser, 1993], and dynamic backtracking [Ginsberg and McAllester, 1994]. For detailed treatments of and surveys of backtracking the reader may wish to confer [Kondrak and van Beek, 1995; 1997; Nadel, 1989; Dechter and Frost, 1999]. The reader may wish to consult [Ginsberg, 1993; Tsang, 1993] for an introductory treatment of backtracking.

It is a well established fact that in order to keep (backtrack) search efficient it is imperative that the branching factors of the nodes near the root of the search tree be kept as small as possible. The contribution of generalised backtracking is that it is the first attempt to keep the branching factor of the search tree small by analysing the structure of *any* kind of constraint and by using an alternative (exhaustive) way to enumerate the members of the constraint.

The remainder of this chapter is as follows. Section 5.2 introduces the notions of *covers* and *partitions* of constraints. This is followed by Section 5.3 which introduces the notion of a *linear* constraint and shows how linear constraints can be used to simplify CSPs. Arguments are presented that this simplification corresponds to a "localised" breadth-first search. Special kinds of partitions of constraints are discussed in Section 5.4. The generalised backtracking algorithm and the experimental results are presented in Section 5.5. A summary is presented in Section 5.6.

5.2 Covers and Partitions of Constraints

This section presents methods to decompose any CSP into several CSPs the solutions of which are pairwise disjoint and the union of the solutions of which is equal to the solutions of the original CSP. It is shown that certain kinds of decompositions are essentially the same as the decompositions that are (implicitly) computed by the chronological backtracking algorithm.

First, the notions of a *cover* of a constraint, that of a *partition* of a constraint, and that of a *maximal partition* of a constraint are presented. This is followed by a proposition which demonstrates the applicability of these notions to the decomposition of CSPs. Finally, an example is presented where the proposition is applied to a maximal partition of a unary constraint to decompose a CSP, thereby explaining how chronological backtracking works.

In the following, 2^S denotes the *power set* of S, i.e. the set of all subsets of S.

Definition 5.1 (Cover). Let S be a set. A set $\kappa \subseteq 2^S$ is called a *cover* of S if $S = \cup_{c\in\kappa} c$. The set containing all covers of S is denoted $K(S)$, i.e. $K(S) = \left\{ \kappa \subseteq 2^S \,:\, S = \cup_{c\in\kappa} c \right\}$.

Example 5.2 (Cover). The set $\{ \{ 0, 1 \}, \{ 1, 2 \} \}$ is a cover of $\{ 0, 1, 2 \}$.

Definition 5.3 (Partition). Let S be a set. A set $\pi \in K(S)$ is called a *partition* of S if $(\forall s_1, s_2 \in \pi)(s_1 \cap s_2 = \emptyset \iff s_1 \neq s_2)$. The set of all partitions of S is denoted $\Pi(S)$.

Partitions are covers whose members are pairwise disjoint.

Example 5.4 (Partition). The set $\{\,\{\,0,1\,\}\,,\{\,2\,\}\,\}$ is a partition of $\{\,0,1,2\,\}$.

The *maximal partition* of a set S is the set $\{\,\{\,s\,\}\,:\,s \in S\,\}$.

It is recalled that the variety of the intersection of two ideals is equal to the union of their varieties (Theorem 3.26), i.e. if I and J are ideals of some polynomial ring $k[X]$ then the following must hold:

$$\mathrm{V}\left(I \cap J\right) = \mathrm{V}\left(I\right) \cup \mathrm{V}\left(J\right).$$

It is also recalled that the variety of the sum of two ideals and the intersection of their varieties are the same (Theorem 3.28), i.e. if I and J are ideals of some polynomial ring $k[X]$ then the following must hold:

$$\mathrm{V}\left(I + J\right) = \mathrm{V}\left(I\right) \cap \mathrm{V}\left(J\right).$$

Finally, it is recalled that there is a relationship between varieties and constraints. The relationship is that a constraint $C_S \subset k^n$ is the variety of an ideal of $k[S]$.

Let $\mathcal{C} = (\,X, D, C\,)$ be any CSP, let C_T be any member of C, and let κ be any cover of C_T. The following proposition states that on the one hand the solutions of $\mathcal{C}$ and on the other the union of the solutions of the CSPs which are created by replacing C_T in C by the members of κ are the same.

Proposition 5.5 (Covers of Constraints). *Let X be a non-empty set of variables, let $R = k[X]$, and let $D(x) \subset k$, for all $x \in X$. Furthermore, let $(\,X, D, C\,)$ be any* CSP*, let $T \subseteq X$, let $C_T \in C$, and let $C' = C \setminus \{\,C_T\,\}$. If $\kappa \in K(C_T)$ is a cover of C_T then the following holds:*

$$\mathrm{V}\left(\sum_{V \in C} \mathrm{I}_R\left(V\right)\right) = \bigcup_{c \in \kappa} \mathrm{V}\left(\mathrm{I}_R\left(c\right) + \sum_{W \in C'} \mathrm{I}_R\left(W\right)\right).$$

Proof. It holds that:

$$\begin{aligned}\mathrm{V}\left(\mathrm{I}_R\left(C_T\right)\right) &= \mathrm{V}\left(\mathrm{I}_R\left(\cup_{c \in \kappa} c\right)\right)\\ &= \mathrm{V}\left(\cap_{c \in \kappa} \mathrm{I}_R\left(c\right)\right)\\ &= \bigcup_{c \in \kappa} \mathrm{V}\left(\mathrm{I}_R\left(c\right)\right).\end{aligned} \tag{5.1}$$

It is also true that:

$$\begin{aligned}\mathrm{V}\left(\sum_{V \in C} \mathrm{I}_R\left(V\right)\right) &= \mathrm{V}\left(\mathrm{I}_R\left(C_T\right) + \sum_{V \in C'} \mathrm{I}_R\left(V\right)\right)\\ &= \mathrm{V}\left(\mathrm{I}_R\left(C_T\right)\right) \cap \mathrm{V}\left(\sum_{V \in C'} \mathrm{I}_R\left(V\right)\right).\end{aligned}$$

Using Equation (5.1) this is tantamount to:

$$\begin{aligned}\mathrm{V}\left(\sum_{V \in C} \mathrm{I}_R\left(V\right)\right) &= (\bigcup_{c \in \kappa} \mathrm{V}\left(\mathrm{I}_R\left(c\right)\right)) \cap \mathrm{V}\left(\sum_{V \in C'} \mathrm{I}_R\left(V\right)\right)\\ &= \bigcup_{c \in \kappa} \mathrm{V}\left(\mathrm{I}_R\left(c\right) + \sum_{V \in C'} \mathrm{I}_R\left(V\right)\right),\end{aligned}$$

which completes the proof. □

Notice that partitions are covers. Therefore, Proposition 5.5 also applies to partitions and maximal partitions.

Proposition 5.5 allows for the decomposition of a CSP into a collection of CSPs. The collection represents the CSP in the sense that the union of the solutions of the members of that collection is equal to the solutions of that CSP.

The following demonstrates how Proposition 5.5 can be used to explain how the chronological backtracking algorithm works.

Example 5.6 (Chronological Backtracking). Let $\mathcal{C} = (\, X, D, C \,)$ be the CSP, where

$$\begin{aligned} X &= \{\, x, y \,\}\,; \\ C &= \left\{\, C_{\{\,x\,\}}, C_{\{\,y\,\}}, C_{\{\,x,y\,\}} \,\right\}; \\ D(x) &= \{\, 1, 2 \,\}\,; \\ D(y) &= \{\, 1, 2, 3 \,\}\,; \\ C_{\{\,x\,\}} &= \{\, 1, 2 \,\}\,; \\ C_{\{\,y\,\}} &= \{\, 1, 2, 3 \,\}\,; \\ C_{\{\,x,y\,\}} &= \{\, (\,1,1\,), (\,1,2\,), (\,2,3\,) \,\}\,. \end{aligned}$$

The solution set of $\mathcal{C}$ is $C_{\{\,x,y\,\}}$. To backtrack with x as the current variable corresponds to the application of Proposition 5.5 to the CSP for the cover (maximal partition, really) $\pi = \left\{\, C'_{\{\,x\,\}}, C''_{\{\,x\,\}} \,\right\}$, where $C'_{\{\,x\,\}} = \{\, 1 \,\}$ and $C''_{\{\,x\,\}} = \{\, 2 \,\}$.

The application of Proposition 5.5 to π allows for the decomposition of the constraint $C_{\{\,x\,\}}$ into the two constraints $C'_{\{\,x\,\}}$ and $C''_{\{\,x\,\}}$. The two constraints can be used to dissect $\mathcal{C}$ into the two CSPs $\mathcal{C}'$ and $\mathcal{C}''$, where

$$\begin{aligned} \mathcal{C}' &= (\, X, D, \{\, C'_{\{\,x\,\}}, C_{\{\,y\,\}}, C_{\{\,x,y\,\}} \,\}\,); \\ \mathcal{C}'' &= (\, X, D, \{\, C''_{\{\,x\,\}}, C_{\{\,y\,\}}, C_{\{\,x,y\,\}} \,\}\,). \end{aligned}$$

The solutions of $\mathcal{C}'$ are given by $\{\, (\,1,1\,), (\,1,2\,) \,\}$ and the solutions of $\mathcal{C}''$ are given by $\{\, (\,2,3\,) \,\}$. The union of the solutions of $\mathcal{C}'$ and $\mathcal{C}''$ is equal to the solution set of $\mathcal{C}$. If the "standard" lexicographical heuristics are used then a chronological depth-first backtracking algorithm will first solve $\mathcal{C}'$ and then $\mathcal{C}''$.

5.3 Linear Constraints

This section will discuss how to use certain properties of constraints to simplify binary CSPs. In particular it will be shown that what will be called *linear* constraints (many-to-one-relations)[1] can be used to simplify binary CSPs. The simplifications consist of a transformation of a binary

[1] A suitable name for linear constraints could also have been *functional* constraints had it not been for the fact that there is a "name clash" for such constraints. For example, [Van Hentenryck *et al.*, 1992] and [David, 1995]

CSP to a binary CSP where a variable has been eliminated (modulo renaming). The sizes of the domains in the resulting CSP will be less than or equal to the sizes of the domains in the original CSP. The number of binary constraints in the resulting CSP will be less than the number of binary constraints in the original CSP. It will be argued that such transformation can be regarded as a "localised" breadth-first search of depth two.

The remainder of this section is as follows. First, definitions will be provided of the *degree* of a set of variables in a constraint and that of a *linear constraint*. Next, examples will be provided of the application of linear constraints to the simplification of CSPs. Finally, a summary will be presented.

Let $k = \mathbb{C}$. The degree of a variable of a polynomial and the total degree of a polynomial are closely related to the number of zeros of that polynomial in k. For example, the degree of a variable in a polynomial is an upper bound of the maximal number of assignments to the variable for which the polynomial will vanish given fixed assignments to the remaining variables in the polynomial.

Example 5.7 (Polynomial Degree (1)). Let $\mathcal{P} = x^2 - 3x + 2 = (x-1)(x-2)$. The degree of x in $\mathcal{P}$ is 2 and $\mathcal{P}$ has exactly 2 zeros for x in $\mathcal{P}$.

Example 5.8 (Polynomial Degree (2)). Let $\mathcal{P} = x^2 - y$.

- The degree of x in $\mathcal{P}$ is 2. If we substitute any value from k for y in $\mathcal{P}$ then there are either 1 or 2 zeros in k for x in $\mathcal{P}$.
- The degree of y in $\mathcal{P}$ is 1. If we substitute any value from k for x in $\mathcal{P}$ then there is 1 zero for y in $\mathcal{P}$.

The ideal-variety correspondence allows us to translate polynomials to constraints and vice versa. This suggests that constraints also have "degrees" and "total degrees." The following is an attempt to generalise these notions of degree and total degree to that of the degree of a set of variables in a constraint. Notions similar to that of the degree of a constraint do not seem to have appeared before in artificial intelligence. The number of substitutions of values for a variable in a polynomial corresponds to a branching factor of a constraint in a search tree. As will be shown later, the degree of a set of variables in a constraint relates these variables to the branching factor. Constraints with low degrees correspond to low branching factors in search.

Definition 5.9 (Degree of Constraint). Let S and T be non-empty sets of variables such that $S \subseteq T$. Furthermore, let $R = k[T]$, let C_T be a constraint whose cardinality is finite, and let $\deg(\cdot, \cdot)$ be the function depicted in Figure 5.1. The number $\deg(C_T, S)$ is called the *degree* of S in C_T. The degree of $\{ x \}$ in C_T will also be called the *degree* of x in C_T or the *x-degree* of C_T.

both use functional constraints with a different meaning. Functional constraints in the context of [Van Hentenryck *et al.*, 1992] are what will be called *bi-linear* constraints here further on. They correspond to one-to-one relations. Functional constraints in the context of [David, 1995] correspond to the notion of what will be called linear binary constraints in this work.

```
function deg(C_T, S) :
var d, m, I, Js, Ps, Vs;
begin
    if C_T = ∅ then
        return 0;
    else if |S| = 1 then begin
        let T \ S = { x_1, ..., x_m };
        I ≔ I_R (C_T);
        Js ≔ { ⟨ x_1 − v_1, ..., x_m − v_m ⟩_R : ( v_1, ..., v_m ) ∈ k^m };
        Vs ≔ { V (I + J) : J ∈ Js };
        d ≔ max({ |V| : V ∈ Vs });
        return d;
    end
    else begin
        Ps ≔ { π ∈ Π(C_T) : (∀c ∈ π)(∃x ∈ S)(deg(c, { x }) = 1) };
        d ≔ min({ |π| : π ∈ Ps });
        return d;
    end;
end;
```

Figure 5.1: Degree function.

If $|S| = 1$ then the degree of S in C_T is the maximum number of solutions for the variable in S that are "allowed" by C_T given fixed assignments to the variables in $T \setminus S$.

A constraint C_T is called *linear* in $x \in T$ if the degree of x in C_T is one. C_T is called *quadratic* in $x \in T$ if the degree of x in C_T is two, and so on. A binary constraint $C_{\{x,y\}}$ is called *bi-linear* if it is linear in both x and y. A constraint C_T is called *sub-linear* (in $S \subseteq T$) if $C_T = \emptyset$. Finally, a constraint C_T will be called *linear* if it is linear in some variable in T.

Example 5.10 (Degree). Consider the binary constraint $C_{\{x,y\}}$ given by:

$$C_{\{x,y\}} = \{ (0,0), (0,1), (0,2), (0,3), (1,0), (2,0) \}.$$

The constraint $C_{\{x,y\}}$ is not linear in x. For example, there are three tuples in $C_{\{x,y\}}$ whose second members are equal. Therefore, the degree of x in $C_{\{x,y\}}$ is at least three. Similarly, $C_{\{x,y\}}$ is not linear in y either. It is left to the reader to verify that the degree of x in $C_{\{x,y\}}$ is three and that the degree of y in $C_{\{x,y\}}$ is four.

Note that $d_{xy} = \deg(C_{\{x,y\}}, \{x, y\}) > 1$ because $C_{\{x,y\}}$ is non-empty and is neither linear in x nor linear in y. However, it can be shown that $d_{xy} = 2$. For example, consider the following two constraints:

$$C'_{\{x,y\}} = \{ (0,1), (0,2), (0,3) \};$$
$$C''_{\{x,y\}} = \{ (0,0), (1,0), (2,0) \}.$$

Both $C'_{\{x,y\}}$ and $C''_{\{x,y\}}$ are linear. The former is linear in x and the latter is linear in y. The set $\pi = \left\{ C'_{\{x,y\}}, C''_{\{x,y\}} \right\}$ is a partition of $C_{\{x,y\}}$. It follows from the definition of the degree of $\{x, y\}$ in $C_{\{x,y\}}$ that $d_{xy} \leq |\pi| = 2$. As observed before, $d_{xy} > 1$. Clearly, d_{xy} is two.

Linear constraints can be used to simplify binary CSPs. If an arc-consistent constraint $C_{\{x,y\}}$ is linear then the variables x and y can be *amalgamated* into a "super-variable" which represents the values in the Cartesian product of the domains of x and y that are in $C_{\{x,y\}}$. The cardinality of the domain of the super-variable is equal to $\max(|D(x)|, |D(y)|)$ and the number of constraints in the resulting CSP will be less than the number of constraints of the original CSP. The transformation will leave all constraints of the form $C_{\{w\}}$ or $C_{\{w,z\}}$ intact, for w and $z \notin \{x, y\}$.

Thus, linear binary constraints allow for the elimination of a variable without causing an increase in the domain sizes of the variables or the number of constraints. The remainder of this section provides concrete examples about the flavour of linear constraints and how to exploit their properties.

Example 5.11 (Singleton Domains). During backtrack search it often occurs that the domain of a future variable reduces to a singleton set. Let x be such variable.

If the problem is binary and if the problem is arc-consistent then x can be removed. Should there be solutions then the projections of these solutions onto the domain of x will be the value in its domains.

If C_T is a constraint which involves x and if the domain of x is a singleton then C_T is linear or sub-linear in x. C_T is linear or sub-linear in x because any assignment to the variables in $T \setminus \{x\}$

which satisfies the projection of C_T onto $T \setminus \{ x \}$ can be extended to at most one assignment to the variables in T such that this extended assignment satisfies C_T. If the projection of C_T onto the domain of x is non-empty then the constraint C_T can be contracted to a constraint on $T \setminus \{ x \}$ without "losing" any solutions; the solutions for x can always be recovered.

The *reason* why the solutions can be recovered is that T is linear in x. Therefore, there is a function from the variables in $T \setminus \{ x \}$ to x. If C_T is binary, then the contraction of C_T entails the creation of a unary constraint on the remaining variable, say y, in $T \setminus \{ x \}$. If the problem is arc-consistent then the contraction of C_T is the same as $D(y)$ and it can be ignored. In binary CSPs that are arc-consistent, variables whose domains are singletons can therefore be eliminated.

In the previous example it was argued that a variable x whose domain is a singleton can be removed from binary arc-consistent CSPs because there is a function from the variables in $T \setminus \{ x \}$ to x and that this mapping could be used to recover the value of x. This is *exactly* the same reason as the one upon which MAC (a backtracker which maintains arc-consistency [Sabin and Freuder, 1994]) relies, namely that after the assignment of a value to the current variable and after arc-consistency processing the current variable can be removed from a problem if it is arc-consistent because there is a function from the future variables to the current variable. Some people may argue that x can be removed because its *only* value has been "saved" and can therefore be recovered. Other people may argue that after the assignment to x any arc-consistent constraint between x and another variable has "become" universal and can therefore be removed. However, the concept of x being "dependent on" a function is more general because—as the following example will demonstrate—it allows for the simultaneous recovery of *several different* values of x as opposed to only one.

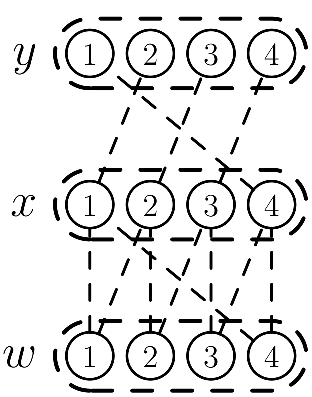

Figure 5.2: Micro-structure before amalgamation.

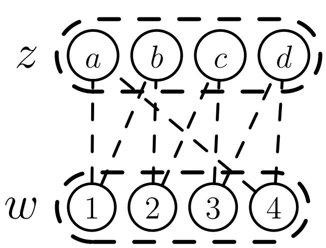

Figure 5.3: Micro-structure after amalgamation.

Example 5.12 (Amalgamation of Nodes (1)). Consider the constraint satisfaction problem $\mathcal{C}$ whose micro-structure is depicted in Figure 5.2. All constraints are binary. The constraint $C_{\{ w,x \}}$ is not linear. The remaining constraint $C_{\{ x,y \}}$ is bi-linear.

Consider the sub-problem consisting of the two variables x and y, their domains, and the constraint $C_{\{ x,y \}}$. As it turns out the sub-problem has exactly four solutions. The nodes x and y

can be transformed into a new node z whose domain contains four values a, b, c and d without increasing the maximum domain size. The transformation is such that these four values represent the four solutions of the sub-problem.

Figure 5.3 depicts the micro-structure of the same CSP where x and y have been "amalgamated" into one fresh variable z. The value a in the domain of z represents the tuple $(\,x, y\,) = (\,1, 2\,)$, b corresponds to $(\,x, y\,) = (\,2, 3\,)$, c corresponds to $(\,x, y\,) = (\,3, 4\,)$, and d corresponds to $(\,x, y\,) = (\,4, 1\,)$. The problem is satisfiable if and only if the original problem is satisfiable, and its solutions are in one-to-one correspondence with the solutions of its original problem. The structure of the new problem is simpler than that of the original problem.

Transformations, like the one from the CSP whose micro-structure is depicted in Figure 5.2 to the CSP whose micro-structure is depicted in Figure 5.3 may also be regarded as the elimination of a variable which linearly depends on a linear constraint (modulo renaming).

For binary CSPs the worst-case time-complexity for the detection of *all* linear constraints is in $\mathbf{O}\,(ed^2)$. This is exactly the time that is required to make a CSP arc-consistent—an "overhead" which is considered to be well spent by researchers in the constraint satisfaction area. It is not difficult to see how to incorporate part of the work for the detection of linear constraints into existing arc-consistency algorithms.

If the domain sizes are large then most binary constraints are not linear and this can be found out without much overhead. The reason why this does not require much overhead is that it is not difficult (on average) to detect that there are at least two tuples in a binary constraint whose first members are equal and to find two tuples in a binary constraint whose second members are equal. However, when domain sizes become small a relatively large proportion of all the possible binary constraints are linear. To detect that a constraint is linear is relatively easy if the sizes of the domains are small. As argued, every linear binary constraint allows for the cheap elimination of a variable from the CSP.

An application of linear constraints which is different from search is to settings where humans are aided by constraint based decision-support-systems. Humans are easily baffled by many variables and many constraints. On the other hand, they seem to understand linear constraints well. They also seem to understand the kind of transformation which corresponds to the amalgamation of nodes. The application of linear constraints to the automatic transformation of a CSP to a CSP whose structure is easier to understand and in one-to-one correspondence to the original CSP seems to be very appropriate in such settings.

The following example demonstrates that, in the presence of constraint propagation, to amalgamate two variables that are involved in a linear binary constraint does not only allow for the elimination of variables but may sometimes allow for the elimination of values.

Example 5.13 (Amalgamation of Nodes (2)). Consider the CSP whose micro-structure is depicted in Figure 5.4. The CSP consists of four variables w, x, y and z, their domains, and four binary constraints $C_{\{\,w,z\,\}}$, $C_{\{\,x,z\,\}}$, $C_{\{\,x,y\,\}}$, and $C_{\{\,y,z\,\}}$. The CSP is arc-consistent.

The binary constraints $C_{\{\,w,z\,\}}$, $C_{\{\,x,z\,\}}$, $C_{\{\,x,y\,\}}$, and $C_{\{\,y,z\,\}}$ which were mentioned before are explicit. Besides these explicit constraints there are also implicit constraints. These constraints are determined by projections, (natural) joins, and intersections of constraints. For example, there is an implicit constraint $C_{\{\,x,y,z\,\}}$ between x, y, and z. $C_{\{\,x,y,z\,\}} = \{\,(\,2, 1, 1\,)\,\}$. The constraint

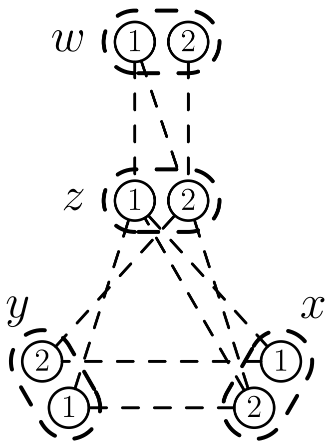

Figure 5.4: Micro-structure before collapsing x and y.

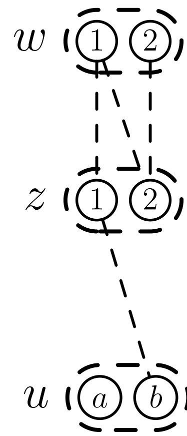

Figure 5.5: Micro-structure after collapsing x and y.

between x, y, and z may also be considered as a constraint between x and y on the one hand and z on the other. This constraint is given by $C_{\{(x,y),z\}} = \{((2,1),1)\}$. The members of $C_{\{(x,y),z\}}$ are in one-to-one correspondence to the solutions of the sub-problem involving the variables x, y, and z, their domains, and the constraints $C_{\{x,y\}}$, $C_{\{x,z\}}$, and $C_{\{y,z\}}$.

$C_{\{x,y\}}$ is linear. Therefore, the nodes x and y can be amalgamated into a node u whose domain contains one value for each of the tuples that are "allowed" by $C_{\{x,y\}}$ without increasing the size of the domains. Renaming (x,y) to u, $(1,2)$ to a, and $(2,1)$ to b results in a CSP the solutions of which are in one-to-one correspondence with the solutions of the original CSP. In particular, $(w,z,u) = (1,1,b) \iff (w,x,y,z) = (1,2,1,1)$.

The micro-structure of the resulting CSP is depicted in Figure 5.5. Note that the CSP is not arc-consistent. The values 2 in the domain of z and a in the domain of u have lost support as a "result" of the constraint $C_{\{(x,y),z\}}$ which was implicit between x, y, and z. It is straightforward to make the CSP arc-consistent again.

To conclude this section it should be observed that the amalgamation of two variables x and y corresponds to what may be regarded as a localised breadth-first search of depth two. To see why this is true observe that the domain of the amalgamation of two variables contains the representatives of the values in the constraint between the variables. This set is equal to the set containing the allowed assignments of the nodes of the search tree at depth two which uses an ordering where x and y are the first variables. The advantage of amalgamation is that no decision has to be made yet about which value to assign to which variable, it simplifies the problem, and it allows for cheap constraint propagation. The advantage of not making a decision about which variable has to become the next current variable is that to postpone this decision may avoid assignments leading to traversals of sub-trees that are infeasibly large.

5.4 Linear Partitions

The previous sections have demonstrated the usefulness of partitions of constraints and linear constraints. In this section we shall study a special kind of partition called *linear* partitions and a function to compute such partitions. The function is non-trivial in the sense that the cardinality of its result is "low." We shall see that linear partitions and the transformation to amalgamate nodes can be used to enumerate the nodes in the search tree of constraints more efficiently than chronological backtracking.

In the following let $\mathrm{proj}_{x_{i_j}}(\cdot)$ be the *projection function* defined as follows:

$$\mathrm{proj}_{x_{i_j}}((\, v_{i_1}, \ldots, v_{i_m}\,)) = \begin{cases} v_{i_j} & \text{if } 1 \leq j \leq m; \\ \perp & \text{otherwise.} \end{cases}$$

Definition 5.14 (Layer). Let S be a non-empty set of variables, let $x \in S$, and let C_S be a non-empty constraint. Furthermore, let $C_{\{\,x\,\}} = \{\, \mathrm{proj}_x(t) \,:\, t \in C_S \,\}$. A set $S_x \subseteq C_S$ is called an x-*layer* of C_S if $|S_x| = |C_{\{\,x\,\}}|$ and $\{\, \mathrm{proj}_x(t) \,:\, t \in S_x \,\} = C_{\{\,x\,\}}$.

Example 5.15 (Layer). Let $C_{\{\,x,y\,\}} = \{\, (\,0,0\,), (\,0,1\,), (\,1,2\,) \,\}$. There are two x-layers of $C_{\{\,x,y\,\}}$. They are given by $\{\, (\,0,0\,), (\,1,2\,) \,\}$ and by $\{\, (\,0,1\,), (\,1,2\,) \,\}$. The only y-layer of $C_{\{\,x,y\,\}}$ is given by $C_{\{\,x,y\,\}}$ itself.

Layers of binary constraints—as the following lemma demonstrates—are linear.

Lemma 5.16 (Linearity of Layers of Binary Constraints). *Let $C_{\{\,x,y\,\}}$ be a non-empty binary constraint, let $z \in \{\, x, y \,\}$ and let $z' = x + y - z$. If S_z is a z-layer of $C_{\{\,x,y\,\}}$ then S_z is linear in z'.*

Proof. If S_z is not linear in z' then there must be at least two tuples, say t_1 and t_2 in S_z, such that $\mathrm{proj}_z(t_1) = \mathrm{proj}_z(t_2)$. This cannot be true because $|S_z| = |\{\, \mathrm{proj}_z(t) \,:\, t \in C_{\{\,x,y\,\}} \,\}|$ and $\{\, \mathrm{proj}_z(t) \,:\, t \in S_x \,\} = \{\, \mathrm{proj}_z(t) \,:\, t \in C_{\{\,x,y\,\}} \,\}$. □

Lemma 5.17 (Monotonicity). *Let $C_{\{\,x,y\,\}}$ be a binary constraint, let $z \in \{\, x, y \,\}$, and let S_z be a z-layer of $C_{\{\,x,y\,\}}$. Furthermore, let $d_w = \deg(C_{\{\,x,y\,\}}, \{\, w \,\})$, and let $d'_w = \deg(C_{\{\,x,y\,\}} \setminus S_z, w)$, for $w \in \{\, x, y \,\}$. Then $\min(d'_x, d'_y) < \min(d_x, d_y)$.*

Proof. Trivial. □

Definition 5.18 (Linear Partition). A *linear partition* is a partition whose members are linear.

The following defines a function to transform a binary constraint to a linear partition of that constraint.

Proposition 5.19 (Linear Partition of Binary Constraint). *Let $C_{\{\,x,y\,\}}$ be a non-empty finite constraint. Furthermore, let $d_x = \deg(C_{\{\,x,y\,\}}, \{\, x \,\})$, and let $d_y = \deg(C_{\{\,x,y\,\}}, \{\, y \,\})$. Finally, let $P_l(\cdot)$ be the function defined in Figure 5.6, then $P_l(C_{\{\,x,y\,\}})$ is a linear partition of $C_{\{\,x,y\,\}}$. Furthermore, $|P_l(C_{\{\,x,y\,\}})| \leq \min(d_x, d_y)$.*

```
function P_l(C_i) :
    var B_i, C_{i+1}, R_i, z;
begin
    if C_i = ∅ then
        return ∅;
    else begin
        if deg(C_i, { x }) > deg(C_i, { y }) then
            z ≔ x;
        else if deg(C_i, { x }) < deg(C_i, { y }) then
            z ≔ y;
        else
            z ≔ any member from { x, y };
        B_i ≔ any z-layer of C_i;
        C_{i+1} ≔ C_i \ B_i;
        R_i ≔ { B_i } ∪ P_l(C_{i+1});
        return R_i;
    end;
end;
```

Figure 5.6: Partition function

Proof. Let $C_1 = C_{\{x,y\}}$. To prove that the proposition is correct it has to be demonstrated that $P_l(C_1)$ terminates, that $P_l(C_1)$ is a partition of C_1, that the members of $P_l(C_1)$ are linear, and that the cardinality of $P_l(C_1)$ does not exceed $\min(d_x, d_y)$.

termination Assume that $P_l(C_1)$ does not terminate. By assumption $|C_1|$ is finite. It follows from the non-termination of $P_l(C_1)$ and its termination criterion that $C_i \supset \emptyset$ for $i \in \mathbb{N} \setminus \{ 0 \}$. This together with the definition of B_i allows us to infer that $\emptyset \subset B_i \subseteq C_i$ must hold. Therefore, $C_{i+1} = C_i \setminus B_i \subset C_i$ and it follows that the sequence

$$C_1 \supset C_2 \supset C_3 \supset \cdots$$

is infinite. This contradicts the premise that $|C_1|$ is finite.

partition property Let $C_1 = C_{\{x,y\}}$ and let $d = |P_l(C_1)|$. To prove that $P_l(C_1)$ is a partition of C_1 we must prove that $C_1 = \cup_{c \in P_l(C_1)} c$ and that $(\forall C_S, C_T \in P_l(C_1))(C_S \neq C_T \iff \emptyset = C_S \cap C_T)$.

First note that $P_l(C_i) = \cup_{i=1}^{d} \{ B_i \}$. Next note that $C_{i+1} \cup B_i = C_i$, for $i = d, d-1, \ldots, 1$. Clearly, $P_l(C_1)$ is a cover of C_1. To see why the members of $P_l(C_1)$ are pairwise disjoint, observe that $B_j \subseteq C_{i+1} = C_i \setminus B_i$, for $1 \leq i < j \leq d$.

linearity property By Lemma 5.16, B_i is linear, for $1 \leq i \leq d$.

cardinality property Use Lemma 5.17 and induction on $\min(d_x, d_y)$. □

We are almost ready to demonstrate the application of linear partitions. Before doing so we need to define the notion of a *generalised branching factor*.

Definition 5.20 (Generalised Branching Factor). The *generalised branching factor* of a linear partition of a constraint is given by the cardinality of that partition.

Note that maximal partitions of unary constraints are linear. Therefore, the notions of generalised branching factor and that of the branching factor coincide for maximal partitions of unary constraints.

The application of linear partitions will become apparent in the next example. Before we go on to that example, it should be pointed out that the minimum of the degrees of the variables that are involved in an arc-consistent binary constraint, cannot exceed the minimum of their domain sizes. This is formulated as the following proposition.

Proposition 5.21. *Let $C_{\{x,y\}}$ be a non-empty constraint, let $d_z = \deg(C_{\{x,y\}}, \{ z \})$, and let $D(z) = \left\{ \operatorname{proj}_z(t) \,:\, t \in C_{\{x,y\}} \right\}$, for $z \in \{ x, y \}$, then*

$$\min(d_x, d_y) \leq \min(|D(x)|, |D(y)|).$$

Proof. $|D(y)| \geq d_x$ and $|D(x)| \geq d_y$. □

In the previous section we have seen that backtracking uses linear partitions of unary constraints to enumerate the members of the domain of the current variable. We have also seen that linear binary constraints can be used to amalgamate two variables. We have argued that this may be viewed as variable elimination (modulo renaming). By combining linear constraints and amalgamation, we can obtain lower (generalised) branching factors.

Example 5.22 (Generalised Backtracking). Consider the constraint $C_{\{x,y\}}$ whose micro-structure is depicted at the top of Figure 5.7. The constraint is cubic in x and in y. The two constraints

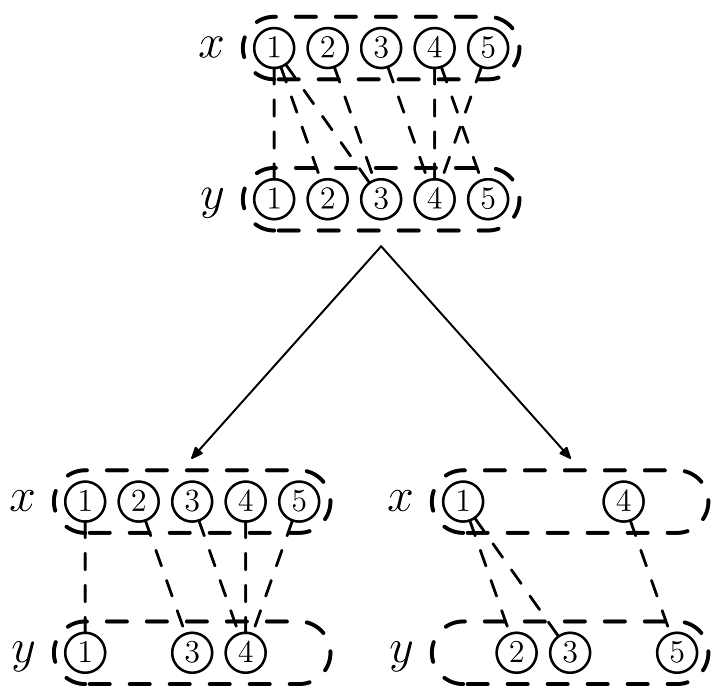

Figure 5.7: Linear partition.

whose micro-structures are depicted at the bottom of Figure 5.7 form the partition $\pi = \{ \mathcal{C}_1, \mathcal{C}_2 \}$ of $C_{\{x,y\}}$, where

$$\mathcal{C}_1 = \{ (1,1), (2,3), (3,4), (4,4), (5,4) \};$$
$$\mathcal{C}_2 = \{ (1,2), (1,3), (4,5) \}.$$

$\mathcal{C}_1$ is linear in y, whereas $\mathcal{C}_2$ is linear in x. The partition was computed using $P_l(\cdot)$ by always selecting the lexicographically smallest z-layer to compute the sets B_i.

Note that the generalised branching factor of π (the cardinality of π) is 2. This is strictly less than the x-degree and the y-degree of the original constraint. This demonstrates that the inequality in Proposition 5.19 may be strict.

The in-order search trees for the chronological backtracking algorithm for the variable orderings $x \prec y$ and $y \prec x$ are depicted in Figures 5.8 and 5.9. The subscripts of the nodes and leaves

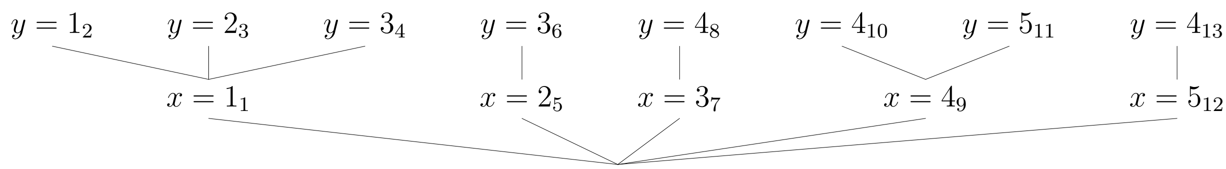

Figure 5.8: In-order search tree. Variable ordering $x \prec y$. Branching factor is 5. Number of visited nodes is 13.

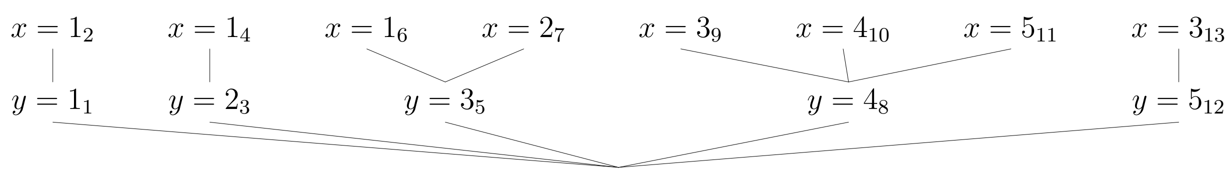

Figure 5.9: In-order search tree. Variable ordering $y \prec x$. Branching factor is 5. Number of visited nodes is 13.

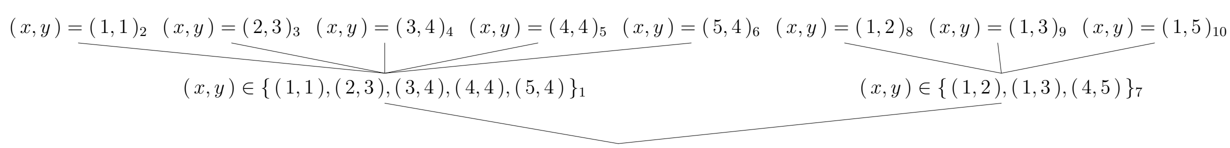

Figure 5.10: Generalised search tree. Generalised Branching Factor is 2. Number of Visited Nodes is 10.

of the trees represent the visiting order. Both trees have 13 visited nodes and have a branching factor of 5 at the root of the tree. The number of leaves of the tree is equal to the cardinality of the constraint $C_{\{x,y\}}$.

The *generalised search tree* corresponding to the partition π is depicted in Figure 5.10. The nodes and leaves of this tree are visited by a *generalised backtracking algorithm*. At the root of the tree there are two branches—one for each member of π. Each of the members of π is linear. As argued before, linear binary partitions of arc-consistent constraints correspond to the domain of a variable. As also argued before, a linear arc-consistent binary constraint can be used to eliminate a variable (modulo renaming) by amalgamating the variables that are involved.

1. The number of leaves of the generalised search tree is equal to the number of leaves of the in-order search trees. There is one leaf for each member of the constraint.

2. The linearity of π ensures that the maximum domain size does not increase.

3. The generalised branching factor at the root does not exceed the minimum domain size (the branching factors of the in-order search trees) of x and y and is usually less than it.

4. The linearity of the partitions ensures that the depth of the generalised search tree is the same as that of the in-order search trees.

5. The number of visited nodes of the generalised backtracking tree is less than the number of nodes of each of the chronological backtracking trees. This is a consequence of 1, 2, and 3, and the fact that (for this example) the generalised branching factor is strictly less than the minimum domain size.

The most important effect of the generalised backtracking approach is that it has decreased the generalised branching factor.

As argued before, chronological backtracking uses linear partitions of unary constraints to decompose problems. The best generalised branching factor that can be obtained by the chronological backtracking algorithm is equal to the minimum domain size of the variables in a problem. The generalised branching factor of binary constraints is usually less than the minimum domain size of the variables that are involved.

Generalised backtracking works because the degrees of constraints determine the generalised branching factor. If the degree of a variable x in a binary constraint is d_x then x can be eliminated (modulo renaming) at the cost of a branching factor of d_x or less. We have already observed that d_x never exceeds the cardinality of the domain of x and may be less than it. If d_x is less than the domain size of x then a smaller branching factor can be achieved than with the traditional backtracking approach. Since the domain sizes do not increase and since the height of the search tree remains the same this results in strictly fewer visited nodes.

Figures 5.7–5.10 suggest that every leaf in the tree (read the representatives of the members of $C_{\{x,y\}}$) can be visited. This is not true in general. There are at least two reasons. The first reason is that in general there may be more variables in a problem and the variables which will be the current variable at depth two from the root of the tree may be different. The second reason is that branches may become dead-ends as a result of the use of constraint propagation techniques.

5.5 Experimental Results

This section presents some experimental results obtained by the application of the generalised backtracking algorithm to some examples from the literature. First we shall discuss some implementation issues. Next we shall describe the problems, present the results, and discuss improvements. Finally, we shall discuss future work.

5.5.1 Implementation Issues

The algorithm which was used for the experiment described in this section was implemented in Haskell. The following are the steps carried out by the algorithm.

- The algorithm maintains arc-consistency. Backtracking occurs as soon as an arc-inconsistency occurs.

- The algorithm removes every variable which is not involved in any constraint.

- The algorithm uses a heuristic to select a binary constraint $C_{\{x,y\}}$ whose generalised branching factor is likely to be low. In the process of finding $C_{\{x,y\}}$, all universal constraints will be removed. The algorithm uses the domain sizes of the variables to get an impression of an upper bound on the generalised branching factor. $C_{\{x,y\}}$ is selected from the constraints whose variables have the smallest domains. Note that a minimum domain size heuristic would have selected the next current variable to be a variable whose domain size was equal to $\min(|D(x)|, |D(y)|)$.

  The algorithm will inspect $C_{\{x,y\}}$ to see if it is linear.

  - If $C_{\{x,y\}}$ is linear then the algorithm will amalgamate x and y. Should this result in an inconsistency then backtracking occurs. Otherwise, the algorithm solves the remaining problem.
  - If $C_{\{x,y\}}$ is not linear then the algorithm uses the function $P_l(\cdot)$ defined in Proposition 5.19 to compute a linear partition π of that constraint. For each of the members $C'_{\{x,y\}}$ of π the algorithm replaces $C_{\{x,y\}}$ by $C'_{\{x,y\}}$, amalgamates x and y, and if this did not result in an inconsistency, solves the remaining problem.

As with most other algorithms, heuristics were used to decide tie-breaks. These tie-break deciding heuristics did not make use of special properties of the problem that were solved.

The implementation was not aimed at efficiency in the sense of reducing the total number of consistency-checks. Instead, the aim was to reduce the generalised branching factors.

5.5.2 Some Results

This section describes the results of applications of the generalised backtracking algorithm to two large problems known from the literature. The objective of the experiment was to determine the generalised branching factors during the different stages of the problem and compare this with the minimum domain size of the variables at these stages. For a chronological backtracking algorithm which maintains arc-consistency the minimum domain size of the variables is a lower bound on its generalised branching factor. The problems which were used are the Radio Link Frequency Assignment Problems (RLFAPs) #3 and #4 [CELAR, 1994]. These problems are originally optimisation problems. However, they have been used here as exemplification problems.

Problem	Variables	Constraints			
		Total Number	Universal	Linear	Partitioned
RFLAP #3	400	2760	80	199	173
RFLAP #4	680	3967	99	300	181

Table 5.1: Problem overview.

Some basic properties of RLFAP #3 and #4 are listed in Table 5.1. The column "Variables" lists the number of variables for the problems. The column "Total Number" in the "Constraints" column lists the number of binary constraints of the problems. The column "Linear" in the "Constraints" column lists the number of linear constraints that were detected during search. It is recalled that such constraints were used for the elimination of a variable (modulo renaming). The column "Partitioned" in the "Constraints" column lists the number of (non-linear) constraints which were partitioned during search. Note that this number represents the number of decision points which were encountered during search. For neither of the problems was backtracking required.

In the following, let π_i be the i-th linear partition (the i-th decision point) which was computed by the algorithm. Let O_i be the original branching factor at the stage in the backtracking process when π_i was computed. It is recalled that O_i is the minimum domain size of all remaining variables in the problem—the best branching factor which can be obtained by a chronological backtracking algorithm which maintains arc-consistency. Finally, let G_i be the generalised branching factor of π_i.

Figure 5.11 depicts the original branching factor O_i and the generalised branching factor G_i for RLFAP #3. The solid line depicts O_i as a function of i. The dashed line depicts G_i as a function of of i. Figure 5.11 depicts the ratio G_i/O_i of the generalised and original branching factors as a function of i.

Figure 5.13 depicts the original and generalised branching factors for RLFAP #4. Figure 5.14 depicts the ratio of the generalised and original branching factors for RLFAP #4.

A first observation is that the generalised branching factor does not exceed the original branching factor and can be considerably less than it (especially for RLFAP #3). The ratio between the generalised and the original branching factors can be as much as 0.4–0.5 even at nodes where there are many branches. This may be an indication that a proper implementation could also save consistency-checks. A second observation is that (especially for RLFAP #4) the ratio between the original and the generalised branching factor is almost 1 at different times during search. This may be an indication that the problems are relatively loose at those times.

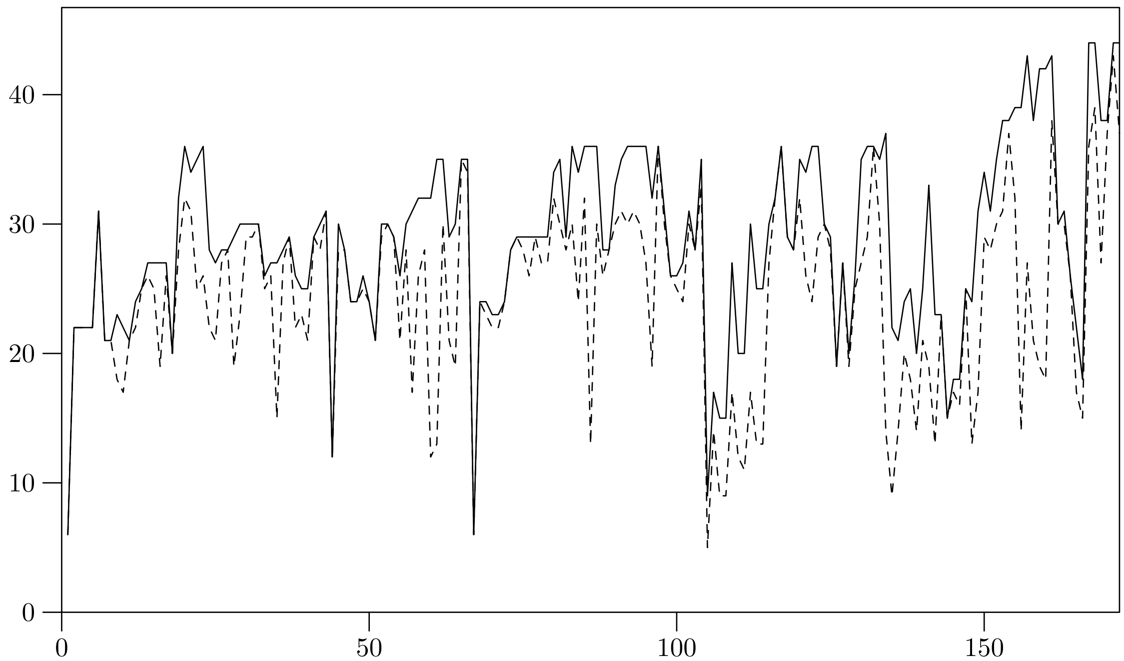

Figure 5.11: Branching factors for RLFA problem #3. Solid: O_i. Dashed: G_i.

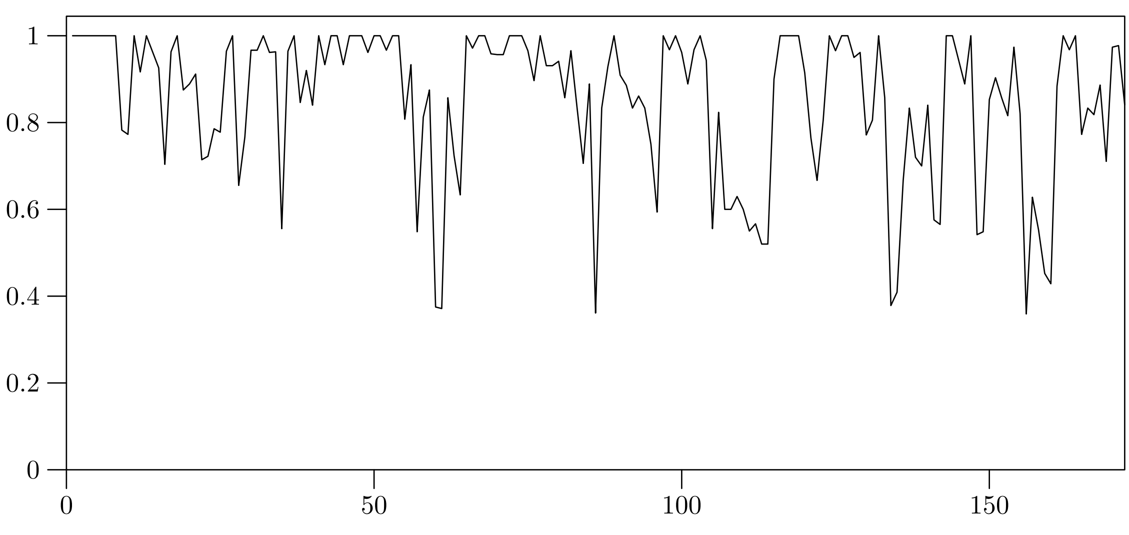

Figure 5.12: Ratio G_i/O_i of branching factors for RLFA problem #3.

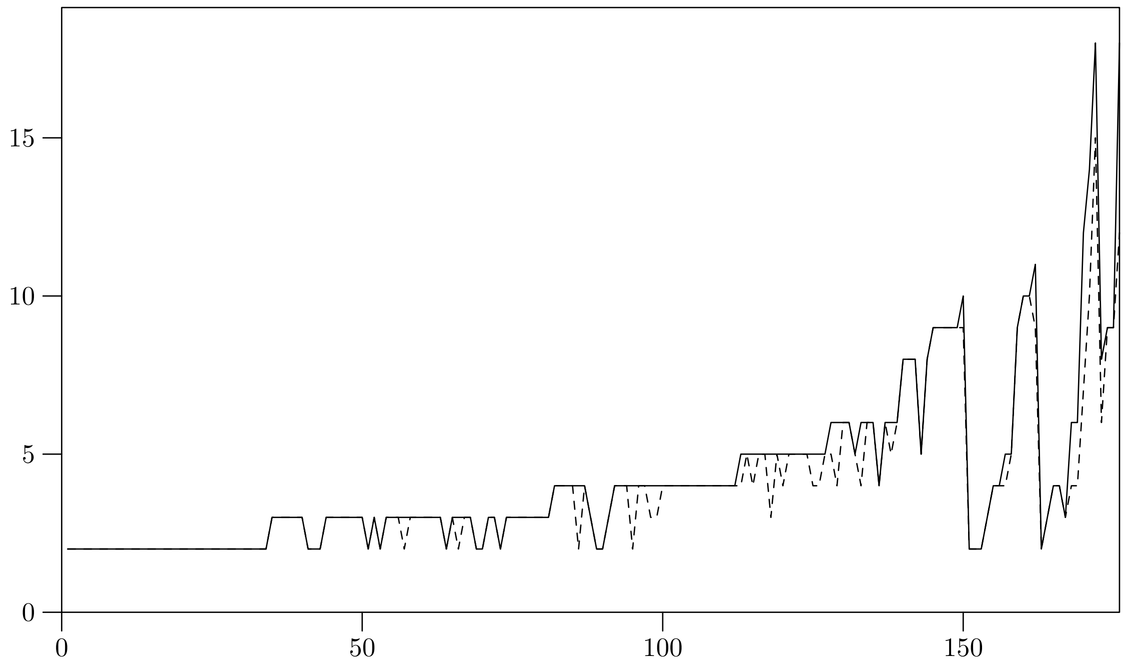

Figure 5.13: Branching factors for RLFA problem #4. Solid: O_i. Dashed: G_i.

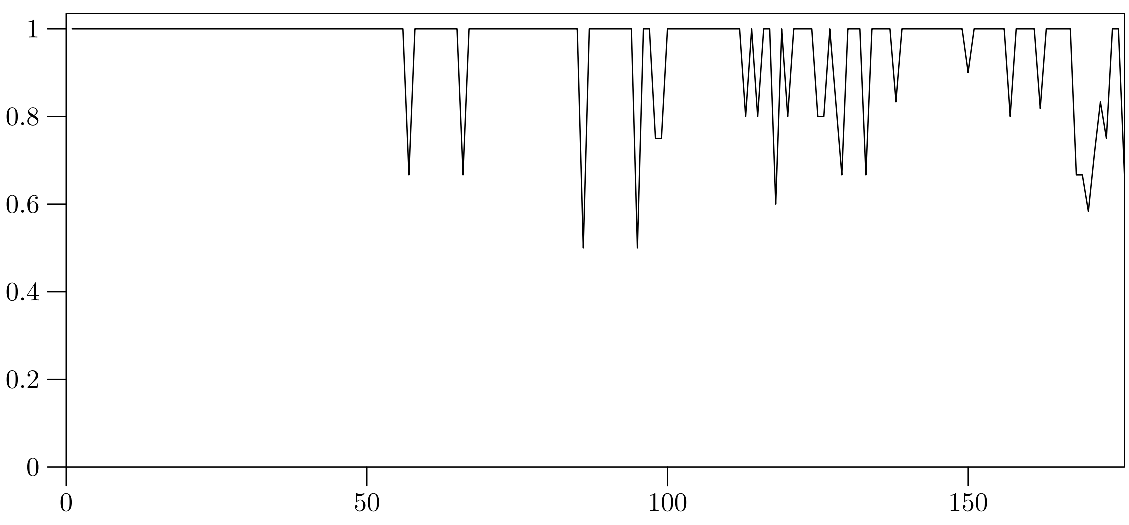

Figure 5.14: Ratio G_i/O_i of branching factors for RLFA problem #4.

5.5.3 Future Work

In this section we shall investigate possibilities for proper implementations of the generalised backtracking algorithm.

After arc-consistency has been obtained, it is important to find a binary constraint with a low generalised branching factor. To find such constraints may require many consistency-checks. However, a lot of the work to detect such constraints can be combined with the work to maintain arc-consistency. The main reason for this is that an upper bound of the generalised branching factor of a constraint $C_{\{x,y\}}$ is $\min(d_x, d_y)$, where d_x is the x-degree of $C_{\{x,y\}}$ and d_y is the y-degree of $C_{\{x,y\}}$. In the following, let z be any member of $\{x, y\}$, let $z' = x + y - z$, and let v be any member of $D(z)$. Furthermore,

- Let $S_z^+(v)$ be the values in $D(z')$ that are known to support v.
- Let $S_z^-(v)$ be the values in $D(z')$ that are known not to support v.
- Let $u(v, z) = |D(z')| - |S_z^-(v)|$, and let $U(z) = \max(\{ u(v, z) : v \in D(z) \})$. Then $U(z)$ is an upper bound for the z-degree of $C_{\{x,y\}}$
- Let $l(v, z) = |S_z^+(v)|$, and let $L(z) = \max(\{ l(v, z) : v \in D(z) \})$. Then $L(z)$ is a lower bound for the z-degree of $C_{\{x,y\}}$.

As part of their effort to make problems arc-consistent, arc-consistency algorithms can keep track the consistency-checks they carried out. For each constraint $C_{\{x,y\}}$ these consistency-checks can be used to get an idea about the x-degree d_x of $C_{\{x,y\}}$ and the y-degree d_y of $C_{\{x,y\}}$ because $L(x) \leq d_x \leq U(x)$ and $L(y) \leq d_y \leq U(y)$. If $P_l(\cdot)$ is used to compute a linear partition of $C_{\{x,y\}}$ then the minimum branching factor of $P_l(C_{\{x,y\}})$ is at most $\min(d_x, d_y)$.

5.6 Summary

In this chapter, techniques have been presented to use the geometry of constraints to study their properties, to transform CSPs, and to solve them. It has been shown that geometric properties of constraints can be used to reason about and simplify CSPs.

The notion of the degree of a set of variables in a constraint has been introduced. This has led to the notion of a linear constraint and it has been shown how linear constraints can be used for the simplification of CSPs by amalgamating the variables involved in a linear binary constraint. This amalgamation operation corresponds to a variable elimination (modulo renaming). Arguments have been presented that for binary CSPs the average costs for the detection of linear constraints is low if arc-consistency is maintained.

It has been shown that the essence of chronological backtracking is that it uses linear partitions of unary constraints (the domains of the variables) to decompose a CSP into a set of CSPs the solutions of which are disjoint and whose union is equal to the solutions of the original CSP. The cardinality of a linear partition is called the generalised branching factor of that partition. The optimal generalised branching factor for a chronological backtracking algorithm which maintains arc-consistency is equal to the minimum domain size.

A generalisation of the chronological backtracking algorithm has been presented. This algorithm, called generalised backtracking, is not restricted to the use of linear partitions of unary constraints to decompose CSPs but can use any kind of constraint for this purpose. A function $P_l(\cdot)$ has been presented to compute linear partitions of binary constraints. The cardinalities of these partitions are small. If $C_{\{x,y\}}$ is a constraint such that the size of the domain of x or y is equal to the minimum domain size then the generalised branching factor of $P_l(C_{\{x,y\}})$ is never larger then the minimum domain size but may be smaller.

A few results have been presented of applications of a toy implementation of the generalised backtracking algorithm. The results are promising in the sense that they demonstrated that significant reductions of the generalised branching factor can be obtained. It may be possible that, with proper adjustments, the algorithm may become an improvement on the standard backtracking algorithm in the sense that it will also save consistency-checks. Suggestions have been presented on how to properly implement the algorithm. However, future research has to demonstrate whether generalised backtracking can be implemented efficiently and proper experiments have to be set up to compare chronological and generalised backtracking. Of course, these experiments should be complemented by a theoretical investigation.

Chapter 6

The AC-3_b Arc-Consistency Algorithm

6.1 Introduction

Arc-consistency algorithms are widely used to reduce the search-space of CSPs (Constraint Satisfaction Problems). Arc-consistency algorithms require *support-checks* (also known as consistency-checks in the constraint literature) to find out about the properties of CSPs. They use *arc-heuristics* and *domain-heuristics* to select their next support-check. Arc-heuristics operate at *arc-level* and select the constraint that will be used for the next check. Domain-heuristics operate at *domain-level*. Given a constraint, they decide which values will be used for the next check. Certain kinds of arc-consistency algorithms use heuristics which are—in essence—a combination of arc-heuristics and domain-heuristics.

In this chapter we will present a domain-heuristic which uses the notion of a *double-support check* to improve the average performance of arc-consistency algorithms. The improvement is that, where possible, support-checks are used to find supports for *two* values, one in the domain of each variable, which were not previously known to be supported. It is motivated by the insight that *in order to minimise the number of support-checks it is necessary to maximise the number of uncertainties which are resolved per check.* We used this idea to improve AC-3 and DEE to obtain a new general purpose arc-consistency algorithm called AC-3_b. We will present experimental results of a comparison of DEE, AC-3, AC-7, and AC-3_b. Our experimental results seem to indicate that AC-3_b always performs better than DEE and usually performs better than both AC-3 and AC-7 for the set of test-problems under consideration. Our average time-complexity results to be presented in Chapter 7 support these results. Together, these results seem to suggest that the double-support heuristic can be used to improve arc-consistency algorithms beyond the current state-of-the-art.

It is well known from the literature that arc-heuristics can influence the performance of arc-consistency algorithms. To the best of our knowledge, ours is the first domain-heuristic to improve the performance of arc-consistency algorithms.

The rest of this chapter is organised as follows. In Section 6.2 we will briefly recall some of the concepts from the constraint satisfaction literature that will be used in this chapter. We will discuss related work in Section 6.3. In Section 6.4 we will present the notion of a double-support check. We will describe AC-3_b in Section 6.5. Our experimental results will be presented and discussed in Section 6.6. Finally, in Section 6.7, we will present our conclusions and discuss further research.

6.2 Constraint Satisfaction Theory

In this section we will briefly recall the notions from the constraint satisfaction literature that will be used in this chapter. In Section 6.2.1 we will recall some basic constraint satisfaction terminology. In Section 6.2.2 we will lay out the main concepts of arc-consistency.

6.2.1 Constraint Satisfaction

Remember that a CSP is a triple (X, D, C), where X is a set containing the variables of the CSP, D is a function which maps each of the variables of the CSP to its domain, and C is a set containing the constraints of the CSP. We will denote the domain of variable x by $D(x)$. In this chapter we will only consider *binary* CSPs, i.e. we will only consider CSPs where the arity of the constraints is at most 2.

A *binary constraint* $C_{\{x,y\}} \subseteq D(x) \times D(y)$ between two variables x and y is a set of pairs. The pairs in the constraint represent the only combinations of values the variables can take. $C_{\{x,y\}}$ allows for x to take the value v and y to take the value w if and only if $(v, w) \in C_{\{x,y\}}$. Likewise, a *unary constraint* $C_{\{x\}}$ is a subset of the domain of x. A member of a constraint is said to *satisfy* the constraint.

A CSP is called *node-consistent* if and only if, for each variable x, either $C_{\{x\}} \notin C$ or each value in its domain satisfies $C_{\{x\}}$. Without loss of generality we will only consider node-consistent CSPs. Furthermore, we will assume that $(\forall x \in X)(D(x) \neq \emptyset)$. A test of the form $v \in C_{\{x\}}$ or $(v, w) \in C_{\{x,y\}}$ is called a *support-check* (normally referred to as *consistency-check* in the constraint literature).

Associated with a binary CSP is its directed constraint graph with nodes corresponding to the variables, and arcs corresponding to the constraints in the CSP. For every unary constraint $C_{\{x\}}$ an arc (x, x) exists. For every binary constraint $C_{\{x,y\}}$, two directed arcs (x, y) and (y, x) exist. These arcs correspond to the "directed" relations R_{xy} and R_{yx}, where $R_{xy} = C_{\{x,y\}}$ and $R_{yx} = \{ (w, v) : (v, w) \in R_{xy} \}$. The set containing all directed relations is called R. More formally, $R = \cup_{C_{\{x,y\}} \in C} \{ R_{xy}, R_{yx} \}$. Two distinct variables x and y in a CSP are called *neighbours* if $C_{\{x,y\}} \in C$. A CSP is called *connected* if its constraint graph is connected.

An algorithm is called *bi-directional* if it exploits the general property of binary relations that $(v, w) \in R_{xy}$ if and only if $(w, v) \in R_{yx}$ for any $v \in D(x)$, any $w \in D(y)$ and any $R_{xy} \in R$ [Bessière *et al.*, 1995].

6.2.2 Arc-Consistency

Let x and y be variables, let $v \in D(x)$, and let $w \in D(y)$; then $y = w$ *supports* $x = v$ if $(v, w) \in R_{xy}$. In addition $x = v$ is said to be *supported* by y if there is some $w \in D(y)$ such that $y = w$ supports $x = v$.

Given the notion of support, a connected CSP is called *arc-consistent* if and only if every value in the domain of every variable is supported by all the neighbours of that variable.

A CSP is called *inconsistent* if it has no solutions. Arc-consistency algorithms repeatedly remove all unsupportable values from the domains of variables, or decide that a CSP is inconsistent by finding that some variable has no supported values in its domain.

Arc-consistency algorithms require support-checks to find out about the properties of CSPs. They use *arc-heuristics* and *domain-heuristics* to select their next support-check. Arc-heuristics operate at *arc-level* and select the constraint that will be used for the next check. Domain-heuristics operate at *domain-level*. Given a constraint, they decide which values will be used for the next check. Certain kinds of arc-consistency algorithms use heuristics which are—in essence—a combination of arc-heuristics and domain-heuristics.

6.3 Related Work

In this section we will briefly discuss some related work on *general purpose* arc-consistency algorithms. Here, by general purpose algorithm, is meant an algorithm which can be applied to any binary CSP.

One of the earliest algorithms is AC-3 [Mackworth, 1977]. It has a worst-case time-complexity of $\mathbf{O}(ed^3)$ and a space-complexity of $\mathbf{O}(e + nd)$ [Mackworth and Freuder, 1985; 1993]. As usual n denotes the number of variables in the CSP, e denotes the number of binary-constraints, and d denotes the maximum domain-size.

Gaschnig introduced a related algorithm called DEE [Gaschnig, 1978]. DEE differs from AC-3 in that, in essence, whereas AC-3 processes only one arc $(\, x, y \,)$ at a time, DEE processes both $(\, x, y \,)$ and $(\, y, x \,)$ at the same time.

A bi-directional arc-consistency algorithm called AC-7 has been presented in [Bessière *et al.*, 1995]. AC-7 never repeats support-checks, has a $\mathbf{O}(ed)$ space-complexity, optimal $\mathbf{O}(ed^2)$ worst-case time-complexity, and never performs worse than AC-3 if AC-3 and AC-7 are both implemented with the usual lexicographic heuristics.

It is known from the literature that arc-heuristics can influence the performance of arc-consistency algorithms [Wallace and Freuder, 1992]. To the best of our knowledge, no reports have yet been presented that the proper use of domain-heuristics can improve the performance of arc-consistency algorithms.

6.4 Double-Support Checks

In this section we will introduce the notion of a *double-support check* and point out some of its properties. Consider the 2-variable CSP whose micro-structure is depicted in Figure 6.1. For this CSP we need at least four checks to find support for the four different values in the domain of y. These checks will also find us support for the values 1, 2, and 3 in the domain of x. Furthermore, we need at least four checks to decide that the value 4 has to be removed from the domain of x. Therefore, the total number of support-checks that we need to transform the CSP to its arc-consistent equivalent is eight or more. The minimum number of support-checks that are required is eight.

If a heuristic of lexicographically ordering the data-structures in AC-3, DEE, and AC-7 is assumed, then AC-7 would need 11 support-checks to make the CSP arc-consistent. The checks required by AC-7 are given by $(1,1) \in R_{xy}$, $(2,1) \in R_{xy}$, $(2,2) \in R_{xy}$, $(3,1) \in R_{xy}$, $(3,2) \in R_{xy}$, $(3,3) \in R_{xy}$, $(4,1) \in R_{xy}$, $(4,2) \in R_{xy}$, $(4,3) \in R_{xy}$, $(4,4) \in R_{xy}$, and finally, $(4,1) \in R_{yx}$. DEE would also need 11 support-checks to transform this CSP into its arc-consistent equivalent. For AC-3 this number would be 17. One of the reasons why AC-3 needs more support-checks than DEE and AC-7 is because AC-3 does not exploit the fact that relations are bi-directional. Bi-directionality is used by DEE, because while it is constructing a support for x, each value in $D(y)$ which is found to support a value in $D(x)$ is marked. It then tries to determine which values in $D(y)$ are supported by x but will not try to find a support for those values in $D(y)$ which are marked because they are already known to be supported. Bi-directionality is exploited by AC-7 because it never tests for $(w,v) \in R_{yx}$ if it already has checked $(v,w) \in C_{\{x,y\}}$, and vice versa.

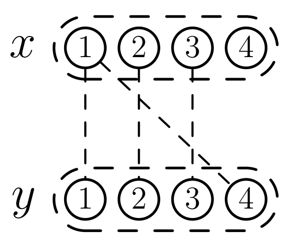

Figure 6.1: Micro-structure of 2-variable CSP.

But even for 2-variable CSPs and the lexicographical heuristics mentioned before, DEE and AC-7 do too much work. For example, after AC-7 has established that $y = 1$ supports $x = 1$, the first thing it will do to find a support for $x = 2$ is to try to find it with $y = 1$. As shown further on, it would be better to postpone the check $(2,1) \in R_{xy}$, because a support for $x = 2$ may be found elsewhere in $D(y)$, thus allowing for the possibility of *two* values to be added to those values which are known to be supported—as opposed to only one. The basic idea presented in this chapter is the insight that *in order to minimise the number of support-checks it is necessary to maximise the number of uncertainties that are resolved per check.*

The objective of arc-consistency processing is to resolve some uncertainty; it has to be known, for each $v \in D(x)$ and for each $w \in D(y)$, whether it is supported. Support-checks are performed to resolve these uncertainties. A *single-support check*, $(v,w) \in C_{\{x,y\}}$, is one in which, before the check is done, it is already known that either v or w are supported. A *double-support check*, $(v,w) \in C_{\{x,y\}}$, is one in which there is still, before the check, uncertainty about the support-status of both v and w. If a double-support check is successful, two uncertainties are resolved. If a single-support check is successful, only one uncertainty is resolved. A good arc-consistency algorithm, therefore, would always choose to do a double-support check in preference to a single-support check, because the former offers the potential higher payback.

At any stage in the process of making the 2-variable CSP arc-consistent:

- There is a set $S_x^+ \subseteq D(x)$ whose values are all known to be supported by y;
- There is a set $S_x^? = D(x) \setminus S_x^+$ whose values are unknown, as yet, to be supported by y.

The same holds if the roles for x and y are exchanged. In order to establish support for a value $v_x^? \in S_x^?$ it seems better to try to find a support among the values in $S_y^?$ first. The advantage of this is that for each $v_y^? \in S_y^?$ the check $(\, v_x^?, v_y^? \,) \in C_{\{\, x,y\, \}}$ is a double-support check and it is just as likely that any $v_y^? \in S_y^?$ supports $v_x^?$ as it is that any $v_y^+ \in S_y^+$ does. Only if no support can be found among the elements in $S_y^?$, should the elements v_y^+ in S_y^+ be used for single-support checks $(\, v_x^?, v_y^+ \,) \in C_{\{\, x,y\, \}}$. After it has been decided for each value in $D(x)$ whether it is supported or not, either $S_x^+ = \emptyset$ and the 2-variable CSP is inconsistent, or $S_x^+ \neq \emptyset$ and the CSP is satisfiable. In the latter case, the elements from $D(x)$ which are supported by y are given by S_x^+. The elements in $D(y)$ which are supported by x are given by the union of S_y^+ with the set of those elements of $S_y^?$ which further processing will show to be supported by some $v_x^+ \in S_x^+$.

If we apply the procedure as sketched before[1] to the CSP whose micro-structure is depicted in Figure 6.1 we would save support-checks when compared to DEE and AC-7. Instead of the 11 checks needed by AC-7, we would only need the following eight checks: $(\, 1, 1\,) \in C_{\{\, x,y\, \}}$, $(\, 2, 2\,) \in C_{\{\, x,y\, \}}$, $(\, 3, 3\,) \in C_{\{\, x,y\, \}}$, $(\, 4, 4\,) \in C_{\{\, x,y\, \}}$, $(\, 4, 1\,) \in C_{\{\, x,y\, \}}$, $(\, 4, 2\,) \in C_{\{\, x,y\, \}}$, $(\, 4, 3\,) \in C_{\{\, x,y\, \}}$, and finally, $(\, 1, 4\,) \in C_{\{\, x,y\, \}}$. Remember that as argued before eight is the absolute minimum number of checks that are needed to make the CSP arc-consistent.

It is not difficult to find an example where our approach would lead to more support-checks than required with AC-7, DEE or AC-3. For a random 2-variable CSP, however, the proposed method is more likely to lead to fewer support-checks. Proof of this will presented in the following chapter. The crucial insight is that *maximising the number of successful double-support checks is a prerequisite to minimising the total number of support-checks*. The experimental results in Section 6.6 seem to support the claim that a heuristic which aims at maximising the number of successful double-support checks is efficient.

6.5 The AC-3_b Algorithm

Motivated by the observations in Section 6.4, we present a new arc-consistency algorithm called AC-3_b. The algorithm is depicted in Figure 6.2 and Figure 6.3. The style of presentation of the algorithm was chosen to keep it consistent with the style normally found in the literature. The input to the AC-3_b algorithm consists of the directed constraint graph G of the CSP, the set D of the domains of the variables in the CSP and the constraints C of the CSP. Its output is either $(\, \mathrm{wipeout}, \emptyset\,)$ if the CSP is arc-inconsistent or $(\, \mathrm{consistent}, D'\,)$ otherwise, where D' is the arc-consistent equivalent of D.

[1] Again we assume lexicographical ordering.

```
Q ← { (x, y) ∈ G : x ≠ y };
D′ ← copy(D);
while Q ≠ ∅ do begin
    select and remove any (x, y) from Q;
    (S_x^+, S_y^+, S_y^?) ← partition(D′(x), D′(y), C_{x,y});
    if S_x^+ = ∅ then begin
        return (wipeout, ∅);
    end;
    if D′(x) \ S_x^+ ≠ ∅ then begin
        replace D′(x) in D′ by S_x^+;
        Q ← Q ∪ { (z, x) ∈ G : z ≠ x, z ≠ y };
    end
    if (y, x) ∈ Q then begin
        remove (y, x) from Q;
        S_y^+ ← S_y^+ ∪ { v_y ∈ S_y^? : (∃v_x ∈ S_x^+)((v_x, v_y) ∈ C_{x,y}) };
        if D′(y) \ S_y^+ ≠ ∅ then begin
            replace D′(y) in D′ by S_y^+;
            Q ← Q ∪ { (z, y) ∈ G : z ≠ x, z ≠ y };
        end
    end
end
return (consistent, D′);
```

Figure 6.2: The AC-3$_b$ algorithm.

As shown in Figure 6.2 AC-3 uses a function called *partition*. This function is shown in Figure 6.3. Its input consist of the domains $D(x)$ and $D(y)$ and the constraint $C_{\{x,y\}}$. Its output consists of a tuple $(S_x^+, S_y^+, S_y^?)$ such that $S_y^+ \subseteq D(y)$, $S_y^? = D(y) \setminus S_y^+$ and in addition:

$$\begin{aligned} S_x^+ &= \left\{ v_x \in D(x) : (\exists v_y \in S_y^+)((v_x, v_y) \in C_{\{x,y\}}) \right\} \wedge \\ S_y^+ &\subseteq \left\{ v_y \in D(y) : (\exists v_x \in S_x^+)((v_x, v_y) \in C_{\{x,y\}}) \right\}. \end{aligned}$$

These rules express the fact that S_x^+ is the set of all values in $D(x)$ which are supported by $D(y)$, that each of its members is supported by some value of S_y^+ as well and that S_y^+ does not contain values which are not supported by S_x^+.

$S_x^? \leftarrow D(x)$;
$S_x^+ \leftarrow \emptyset$;
$S_y^? \leftarrow D(y)$;
$S_y^+ \leftarrow \emptyset$;
while $S_x^? \neq \emptyset$ do begin
    select and remove any $v_x^?$ from $S_x^?$;
    if $\exists v_y^? \in S_y^?$ such that $(v_x^?, v_y^?) \in C_{\{x,y\}}$ then begin
        select and remove any such $v_y^?$ from $S_y^?$;
        $S_x^+ \leftarrow S_x^+ \cup \{ v_x^? \}$;
        $S_y^+ \leftarrow S_y^+ \cup \{ v_y^? \}$;
    end
    else if $\exists v_y^+ \in S_y^+$ such that $(v_x^?, v_y^+) \in C_{\{x,y\}}$ then begin
        $S_x^+ \leftarrow S_x^+ \cup \{ v_x^? \}$;
    end
end
return $(S_x^+, S_y^+, S_y^?)$;

Figure 6.3: The partition algorithm.

For the AC-3$_b$ algorithm it is assumed that the input CSP is already node-consistent. AC-3$_b$ is a refinement of the AC-3 algorithm as described in [Mackworth, 1977] and DEE as described in [Gaschnig, 1978]. The subscript b in the name stands for bi-directional; perhaps a d for double-support would have been more appropriate. Compared with AC-3 the refinement is that when arc (x, y) is being processed and the reverse arc (y, x) is also in the queue, then support-checks can be saved because only support for the elements in $S_y^?$ has to be found (as opposed to support for all the elements in $D(x)$ in the AC-3 algorithm). Compared with DEE the refinement consists of the double-support heuristic.

AC-3$_b$ inherits all its properties like $\mathbf{O}(ed^3)$ worst-case time-complexity and a $\mathbf{O}(e + nd)$ worst-case space-complexity from AC-3. The proof follows directly from the relationship between AC-3$_b$ and AC-3. Note that the space-complexity of AC-3$_b$ is $\mathbf{O}(e + nd)$ as opposed to $\mathbf{O}(ed)$ for AC-7. The space-complexity characteristics of AC-3$_b$ are better because e (the number of constraints) is quadratic in n (the number of variables).

6.6 Experimental Results

In this Section we present experimental results to enable comparisons between AC-3, DEE, AC-7, and AC-3$_b$. In Section 6.6.1 we will describe the experiments and the implementations of the algorithms. In Section 6.6.2 we will discuss the results. Throughout this section, $\#cc(\mathcal{A})$ will denote the (average) number of support-checks performed by arc-consistency algorithm $\mathcal{A}$.

6.6.1 Description of the Experiment

In order to compare the arc-consistency efficiency of AC-3, DEE, AC-7, and AC-3$_b$ we have generated 30,420 random connected CSPs. For each combination of (density, average tightness) in $\{\,(\,d/40, t/40\,) : t \in \{\,1, 2, \ldots, 39\,\}, d \in \{\,1, 2, \ldots, 39\,\}\,\}$, we generated twenty random connected CSPs. Here, the *density* of a connected binary CSP is defined to be $2 \times (e - n + 1)/(n^2 - 3n + 2)$, where $n > 2$ is the number of variables in the CSP and e is the number of edges in the constraint-graph [Sabin and Freuder, 1994]. The *tightness* of a constraint $C_{\{\,x,y\,\}}$ is defined to be $1 - |C_{\{\,x,y\,\}}|/(|D(x)| \times |D(y)|)$. The *average tightness* of a binary CSP is the average of the tightnesses of the binary constraints [Sabin and Freuder, 1994]. The number of variables per CSP was a random number from 15 to 25. The domain size of the variables was always equal to the number of variables in the problem. The task of the arc-consistency algorithms consisted of transforming each CSP into its arc-consistent equivalent or deciding that the CSP did not have an arc-inconsistent equivalent. The *lexicographical queue* heuristic (see [Wallace and Freuder, 1992] for a description) was used for adding elements to, and removing elements from, queues/streams in all the algorithms.

The DEE version used for the experimentation here, is an arc-queue based version of the one described in [Gaschnig, 1978]. This implementation allows for a good estimation of the efficiency of the double-support heuristic since DEE and AC-3$_b$ both process the same edges in the same order. The two algorithms only differ in their domain heuristic, i.e. they only differ in the way they establish support for the elements in the domains of the variables at both ends of an edge.

6.6.2 Discussion of Results

In this section we will present the results of our experimentation described in the previous section. We have depicted the average numbers of support-checks for AC-3, DEE, AC-7, and AC-3$_b$ for the random CSPs at each combination of density and tightness in Figures 6.4–6.7. The numbers of support-checks for each algorithm averaged over each problem are presented in Table 6.1.

Figures 6.8–6.11 depict the difference graphs for the average number of support-checks between AC-3 and DEE, between AC-3 and AC-3$_b$, between DEE and AC-3$_b$, and between AC-7 and AC-3$_b$. The jagged lines at the bottom of Figures 6.8, 6.9, and 6.11 are where the difference between the number of support-checks equals zero. Figure 6.12 depicts $1 - \#cc(\text{AC-3}_b)/\#cc(\text{DEE})$. Figure 6.13 depicts $1 - \#cc(\text{AC-3}_b)/\#cc(\text{AC-7})$.

Table 6.1 seems to suggest that the DEE approach is a waste. Despite the fact that DEE uses the property that constraints are bi-directional, it can not gain much from it. AC-3, for example,

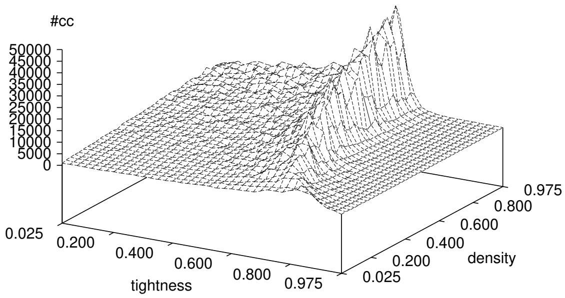

Figure 6.4: $\#cc$(AC-3).

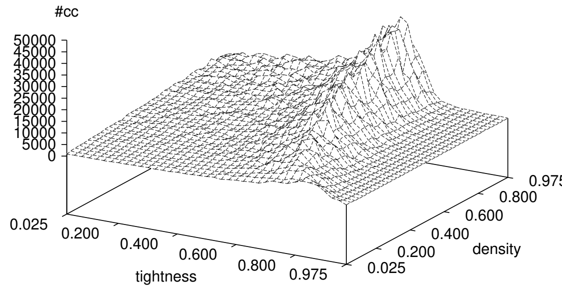

Figure 6.5: $\#cc$(DEE).

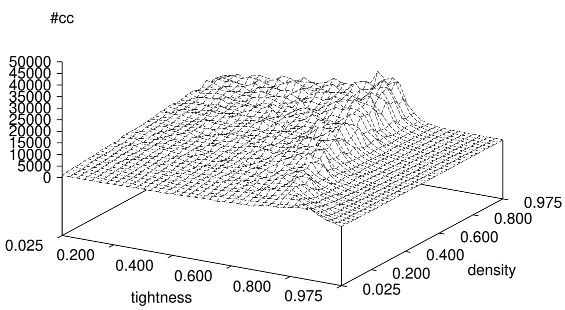

Figure 6.6: $\#cc$(AC-7).

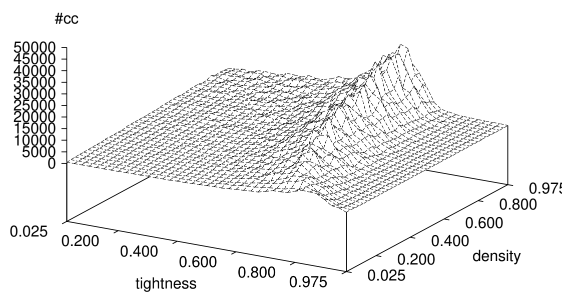

Figure 6.7: $\#cc$(AC-3_b).

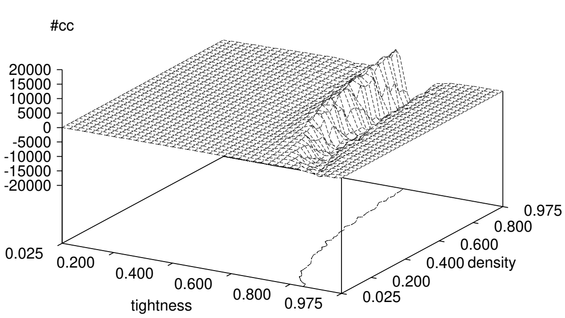

Figure 6.8: $\#cc$(AC-3) $-$ $\#cc$(DEE).

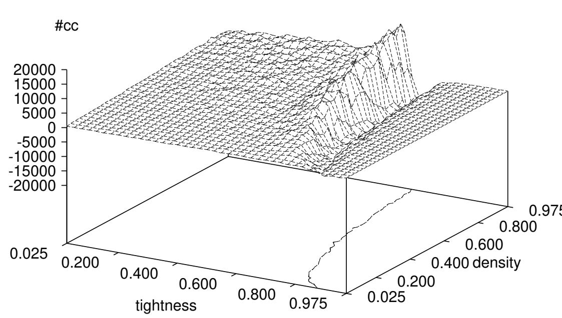

Figure 6.9: $\#cc$(AC-3) $-$ $\#cc$(AC-3_b).

	DEE	AC-3	AC-3$_b$	AC-7
$\#cc$	7311	7261	5077	5319

Table 6.1: Average number of support-checks.

does less work than DEE for certain problems because after AC-3 has processed arc (x, y) it does not always immediately process the reverse arc (y, x) if it is in the queue, whereas DEE always does. To postpone processing (y, x) can be good for two reasons. First, if the domain of x gets narrowed several times, the effect of adding the arc (y, x) to the queue several times, can be overcome by processing (y, x) only once. Second, establishing support for y by using values from $D(x)$ which will be removed from $D(x)$ later may waste support-checks. This may be illustrated by the following two possible events. In the first and most extreme case AC-3 would process another arc, say (x, z), and detect a wipe-out of $D(x)$. Had the arc (y, x) been processed before (x, z) then any support-check spent on this arc would have been wasted. In a less extreme case $D(x)$ could have been narrowed by processing other arcs to x. This may save work when (y, x) has to be processed because, in general, fewer support-checks have to be spent on each of the values in $D(y)$ when $D(x)$ gets smaller. Both effects become more pronounced when constraints become tighter. Only when constraints are loose will DEE outperform AC-3.

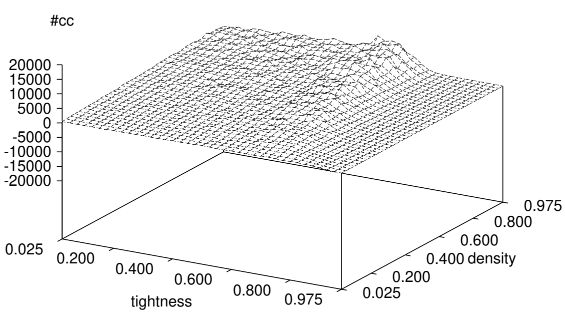

Figure 6.10: $\#cc(\text{DEE}) - \#cc(\text{AC-3}_b)$.

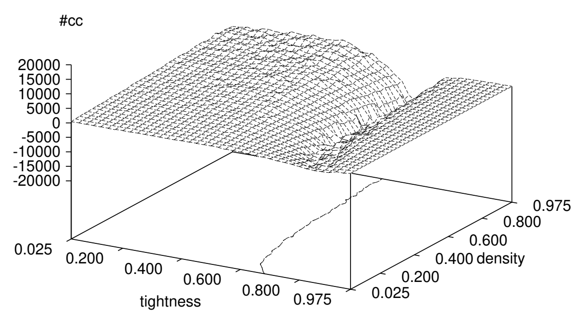

Figure 6.11: $\#cc(\text{AC-7}) - \#cc(\text{AC-3}_b)$.

AC-3$_b$ is always better than DEE. Figure 6.10 shows this—by the way, note that, those parts of the surface of the graph in Figure 6.10 which appear to be in the horizontal plane $\#cc = 0$ are, in fact, above this plane. This seems to suggest that our double-support heuristic is a good one. It is interesting to see that the ratio between the number of support-checks saved by the double-support heuristic and the total number of support-checks, is nearly constant for fixed tightness (see Figure 6.12).

AC-3$_b$ is better than AC-3 everywhere to the left of the phase transition region. AC-3 becomes better in the phase transition region, and stays better from there onwards. The reasons why AC-3 becomes better than AC-3$_b$ as the average tightness increases are the same as why AC-3 becomes better than DEE as average tightness increases.

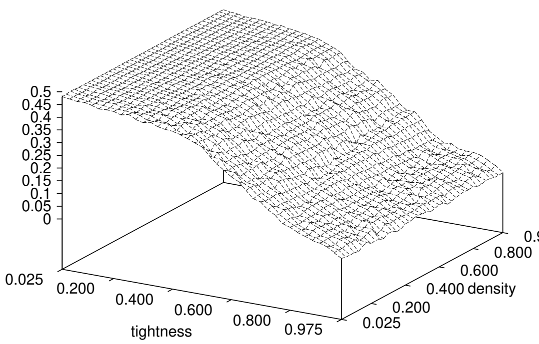

Figure 6.12: $1 - \frac{\#cc(\text{AC-3}_b)}{\#cc(\text{DEE})}$.

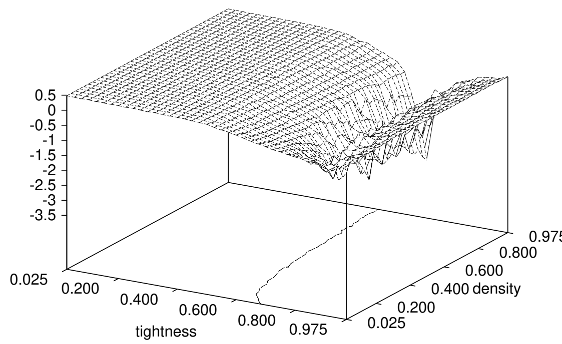

Figure 6.13: $1 - \frac{\#cc(\text{AC-3}_b)}{\#cc(\text{AC-7})}$.

Nevertheless, it seems that AC-3_b is a more efficient algorithm on average than AC-3. The disadvantage of always processing two arcs (the "DEE effect") is turned into an advantage by adopting the double-support heuristic. Possibilities seem to exist to improve AC-3_b and DEE. One possibility is to force the algorithms to degenerate to AC-3 (i.e. never to process a double arc) as soon as they know (or learn) they are processing tight constraints.

AC-3_b requires fewer support-checks than AC-7 in a larger area in the problem space (see Figure 6.11 and Figure 6.13) but as tightness increases AC-7 becomes better. In the low tightness area AC-3_b does better than AC-7 because most of its support-checks will lead to a double support. AC-7 accumulates knowledge about support-checks it has already carried out and never repeats one. Therefore, it has to outperform AC-3_b at some stage as tightness increases.

It may seem surprising that (in our setting) AC-3_b seems to perform better on the test problems than AC-7, despite the fact that AC-3_b has a worse time-complexity. However, this phenomenon actually also occurs elsewhere. For example, in the linear programming community the exponential simplex algorithm is still preferred over existing polynomial algorithms because it behaves better on average. AC-4 (another arc-consistency algorithm) has a better (worst-case) time complexity than AC-3. This did not stop people from using AC-3 because it was almost always better than AC-4 [Wallace, 1993].

6.7 Conclusions and Recommendations

In this chapter we have presented the notion of a double-support check and a domain-heuristic to maximise the number of successful double-support checks. We have used this domain-heuristic to obtain a general purpose arc-consistency algorithm called AC-3_b which is a cross-breed between DEE and AC-3. We have presented experimental results which seem to suggest that our domain-heuristic can improve the average performance of both DEE and AC-3. These results seem to be the first indication that domain-heuristics can improve the efficiency of arc-consistency algorithms. Our results also seem to indicate that, for the problems under consideration, AC-3$_b$ is more efficient in a large part of the tightness-density space than any existing arc-consistency algorithm including AC-7 with its usual lexicographical heuristics. We have suggested changes to improve AC-3$_b$ in the high tightness area. These changes would consist of letting the algorithm degenerate to AC-3 as soon as it would find out that it is processing tight constraints.

It seems that a double-support heuristic can be used to improve AC-7 as well. One of the changes to the algorithm should consist of the addition of a dynamic value ordering for the values in the domains of the variables. This ordering should partially depend on support-checks which were previously carried out, and should also consist of a tie-break ordering. Future research will have to learn what these proposed changes to these algorithms will mean in terms of average performance.

Chapter 7

Average Time-Complexity for Domain-Heuristics

7.1 Introduction

Arc-consistency algorithms are widely used to prune the search-space of CSPs. Arc-consistency algorithms require *support-checks* (also known as consistency-checks in the constraint literature) to find out about the properties of CSPs. They use *arc-heuristics* and *domain-heuristics* to select their next support-check. Arc-heuristics operate at *arc-level* and select the constraint that will be used for the next check. Domain-heuristics operate at *domain-level*. Given a constraint, they decide which values will be used for the next check. Certain kinds of arc-consistency algorithms use heuristics which are—in essence—a combination of arc-heuristics and domain-heuristics.

We will investigate the effect of domain-heuristics by studying the average time-complexity of two arc-consistency algorithms which use different domain-heuristics. We will assume that there are only two variables. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic based on the notion of a *double-support check*. Empirical evidence presented in the previous chapter suggests that the double-support heuristic is efficient.

We will define the algorithms $\mathcal{L}$ and $\mathcal{D}$ and present a detailed case-study for the case where the size of the domains of the variables is two. We will show that for the case-study $\mathcal{D}$ is superior on average. Three reasons will be pointed out why arc-consistency algorithms should give preference to double-support checks at domain-level.

We will carry out an exact average time-complexity analysis for the case where the domains are not restricted to have a size of two. Our analysis will provide solid mathematical evidence that $\mathcal{D}$ is the better algorithm on average.

We will derive relatively simple *exact* formulae for the average time-complexity of both algorithms and derive simple expressions for their upper and lower bounds. To be more specific, we will demonstrate that $\mathcal{L}$ requires about $2a + 2b - 2\log_2(a) - 0.665492$ checks for sufficiently large domain sizes a and b. We will also demonstrate that on average $\mathcal{D}$ requires a number of support-checks which is less than $2\max(a, b) + 2$ if $a + b \geq 14$. Our results demonstrate that $\mathcal{D}$

is the superior algorithm. Finally, we will provide the result that on average $\mathcal{D}$ requires strictly fewer than two checks more than any optimal algorithm if $a+b \geq 14$. This is the first such result ever to have been reported.

As part of our analysis we will compare the average time-complexity of the two algorithms under consideration and discuss the consequences of our simplifications about the number of variables in the CSP.

The relevance of this work is that the double-support heuristic can be incorporated into any existing arc-consistency algorithm. Our optimality result is informative about the possibilities and limitations of domain-heuristics.

The remainder of this chapter is organised as follows. In Section 7.2 we shall provide basic definitions and review constraint satisfaction. A formal definition of the lexicographical and double-support algorithms will be presented in Section 7.3. In that section we shall also carry out our case-study for the case where the size of the domains is two. We shall identify three reasons which, from an intuitive point of view, suggest that at domain-level arc-consistency algorithms should give preference to double-support checks. In Section 7.4 we shall carry out our average time-complexity analysis for the lexicographical algorithm. In Section 7.5 we shall do the same for the double-support algorithm. We shall compare the results of our average time-complexity analysis in Section 7.6. Our conclusions will be presented in Section 7.7.

7.2 Constraint Satisfaction

This section provides our basic definitions and reviews constraint satisfaction. Its organisation is as follows. In Section 7.2.1 we shall provide our basic definitions. In Section 7.2.2 we shall review the related literature. As we already indicated, it is our intention to study arc-consistency algorithms for the case where there are only two variables in the CSP. In Section 7.2.3 we shall discuss the consequences of this simplification.

7.2.1 Basic Definitions

Remember that a *constraint satisfaction problem* (or CSP) is a tuple $(\, X, D, C\,)$, where X is a set containing the variables of the CSP, D is a function which maps each of the variables to its domain and, C is a set containing the constraints of the CSP.

In this chapter we will only consider constraints between two variables at a time. Let $(\, X, D, C\,)$ be a CSP, let α and β two variables in X, let $D(\alpha) = \{\, 1, \ldots, a\,\}$, and let $D(\beta) = \{\, 1, \ldots, b\,\}$. A constraint between α and β restricts the values they are allowed to take simultaneously. For the purpose of our analysis we will represent constraints as matrices and we will only consider constraints between two variables at a time. If M is the constraint between α and β then M is an a by b zero-one matrix, i.e. a matrix with a rows and b columns whose entries are either zero or one. The tuple $(\, i, j\,)$ in the Cartesian product of the domains of α and β is said to *satisfy* the constraint M if $M_{ij} = 1$, where M_{ij} is the j-th column of the i-th row of M. A value $i \in D(\alpha)$ is said to be *supported* by $j \in D(\beta)$ if $M_{ij} = 1$. Similarly, $j \in D(\beta)$ is said to be supported by $i \in D(\alpha)$ if $M_{ij} = 1$.

Definition 7.1 (Arc-Consistency). Let $(\, X, D, C\,)$ be a CSP. The CSP is called *arc-consistent* if $(\forall x \in X)(|D(x)| \neq \emptyset)$ and for each constraint $M \in C$ it holds that if M is between α and β then for every $i \in D(\alpha)$ there is a $j \in D(\beta)$ which supports i and vice versa.

If $(\, X, D, C\,)$ is a CSP then we will assume that it is such that $X = \{\, \alpha, \beta\,\}$ and $C = \{\, M\,\}$, where M is the constraint between α and β. Furthermore we will assume that $D(\alpha) = \{\, 1, \ldots, a\,\} \neq \emptyset$ and $D(\beta) = \{\, 1, \ldots, b\,\} \neq \emptyset$. In other words, we will only concern ourselves with CSPs where there are two variables, where the domains are non-empty, where there is one constraint, and where the constraint is between the two variables of the CSP. We shall discuss the consequences of our simplifications in Section 7.2.3.

We will denote the set of all a by b zero-one matrices by $\mathbb{M}^{ab}$. We will call matrices, rows of matrices and columns of matrices non-zero if they contain more than zero ones, and call them zero otherwise. Finally, we will assume that unless explicitly stated otherwise all matrices are a by b matrices.

Definition 7.2 (Row-Support). The *row-support* of an a by b matrix M is the set

$$\{\, i \in \{1, \ldots, a\} \,|\, (\exists j \in \{\, 1, \ldots, b\,\})(M_{ij} = 1)\,\}\,.$$

The row-support of a matrix is the set containing the indices of its non-zero rows.

Definition 7.3 (Column-Support). The *column-support* of an a by b matrix M is the set

$$\{\, j \in \{1, \ldots, b\} \,|\, (\exists i \in \{\, 1, \ldots, a\,\})(M_{ij} = 1)\,\}\,.$$

The column-support of a matrix is the set containing the indices of its non-zero columns.

Let M be an a by b matrix. We will say that row i *supports* column j if $M_{ij} = 1$. Similarly, column j is said to support row i if $M_{ij} = 1$.

From now on we will also speak of the *support* of a matrix. By this we will mean the tuple $(\, s_r, s_c\,)$, where s_r is the row-support and s_c is the column-support of that matrix.

An *arc-consistency algorithm* removes all the unsupported values from the domains of the variables of a CSP until this is no longer possible. For the case where there are two variables, arc-consistency algorithms compute the support of a matrix. A *support-check* is a test to find the value of an entry of a matrix. We will write $M_{ij}^{?}$ for the support-check to find the value of M_{ij}. An arc-consistency algorithm has to carry out a support-check $M_{ij}^{?}$ to find out about the value of M_{ij}. The time-complexity of arc-consistency algorithms is expressed in the number of support-checks they require to find the support of their arguments.

If we assume that support-checks are not duplicated then at most ab support-checks are needed by any arc-consistency algorithm. For a zero a by b matrix each of these ab checks is required. The worst case time-complexity is therefore exactly ab for any arc-consistency algorithm. In this chapter we are interested in the average time-complexity of arc-consistency algorithms.

If $\mathcal{A}$ is an arc-consistency algorithm and M an a by b matrix, then we shall write $\mathrm{checks}_{\mathcal{A}}(M)$ for the number of support-checks required by $\mathcal{A}$ to compute the support of M.

Definition 7.4 (Average Time-Complexity). Let $\mathcal{A}$ be an arc-consistency algorithm. The *average time-complexity* of $\mathcal{A}$ *over* $\mathbb{M}^{ab}$ is the function $\operatorname{avg}_{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, where

$$\operatorname{avg}_{\mathcal{A}}(a, b) = \sum_{M \in \mathbb{M}^{ab}} \operatorname{checks}_{\mathcal{A}}(M)/2^{ab}.$$

Definition 7.5 (Repetitive Arc-Consistency Algorithm). Let $\mathcal{A}$ be an arc-consistency algorithm, then $\mathcal{A}$ is called *repetitive* if it repeats support-checks and *non-repetitive* otherwise.

A support-check $M^{?}_{ij}$ is said to *succeed* if $M_{ij} = 1$ and said to *fail* otherwise. If a support-check succeeds it is called *successful* and *unsuccessful* otherwise.

Definition 7.6 (Trace). Let a and b be positive integers, let M an a by b zero-one matrix, and let $\mathcal{A}$ be an arc-consistency algorithm. The *trace* of M with respect to $\mathcal{A}$ is the sequence of the form

$$(\, i_1, j_1, M_{i_1 j_1}\,), (\, i_2, j_2, M_{i_2 j_2}\,), \ldots, (\, i_l, j_l, M_{i_l j_l}\,), \tag{7.1}$$

where $l = \operatorname{checks}_{\mathcal{A}}(M)$ and $M^{?}_{i_k j_k}$ is the k-th support-check carried out by $\mathcal{A}$, for $1 \leq k \leq l$. The *length* of the trace in Equation (7.1) is defined as l. Its k*-th member* is defined by $(\, i_k, j_k, M_{i_k j_k}\,)$, for $1 \leq k \leq l$.

An a by b matrix has ab entries. Therefore, the lengths of the traces of non-repetitive algorithms are less than or equal to ab. An interesting property of traces of non-repetitive algorithms is the one formulated as the following theorem.

Theorem 7.7 (Trace). *Let $\mathcal{A}$ be a non-repetitive arc-consistency algorithm, let a and b be positive integers, let M be an a by b zero-one matrix, and let t be the trace of M with respect to $\mathcal{A}$. If l is the length of t then the number of matrices whose trace is t is exactly 2^{ab-l}.*

Proof. Let the k-th member of t be $(\, i_k, j_k, M_{i_k j_k}\,)$, for $1 \leq k \leq l$. We are looking for $|S|$, where

$$S = \left\{\, M' \in \mathbb{M}^{ab} \;:\; (\forall k \in \{\, 1, \ldots, l \,\})(M_{i_k j_k} = M'_{i_k j_k}) \,\right\}.$$

S contains exactly 2^{ab-l} members. □

The theorem will turn out to be convenient later on because it will allow us to determine the "savings" of traces of non-repetitive arc-consistency algorithms without too much effort.

7.2.2 Related Literature

Arc-consistency algorithms have been studied for quite some time. In 1977, Mackworth pointed out reasons why problems that are not arc-consistent are more difficult to solve with techniques based on backtracking and presented three arc-consistency algorithms called AC-1, AC-2, and AC-3 [Mackworth, 1977]. AC-3 is the most efficient of the three and, in a joint paper with Freuder, worst-case time-complexity results for AC-3 are presented [Mackworth and

Freuder, 1985]. The lower bound they present is $\mathbf{\Omega}(ed^2)$ and their upper bound is $\mathbf{O}\,(ed^3)$, where e is the number of constraints and d is the maximum domain-size. Both bounds are linear in the number of constraints.

AC-3, as presented by Mackworth, is not an algorithm as such; it is a *class* of algorithms which have certain data-structures in common and treat them similarly. The most prominent data-structure used by AC-3 is a *queue* which initially contains each of the pairs $(\,\alpha, \beta\,)$ and $(\,\beta, \alpha\,)$ for which there exists a constraint between α and β. The basic machinery of AC-3 is such that *any* tuple can be removed from the queue. For a "real" implementation this means that certain heuristics determine the choice of the tuple that was removed from the queue. By selecting a member from the queue, these heuristics determine the constraint that will be used for the next support-checks. In this chapter, such heuristics will be called *arc-heuristics*.

If $(\,\alpha, \beta\,)$ is the tuple that was removed from the queue then every value in the domain of α which is not supported by some value in the domain of β is removed from the domain of α. If values are removed from the domain of α then pairs of the form $(\,\gamma, \alpha\,)$ are added to the queue for every constraint between γ and α in the CSP, except for the case where $\beta = \gamma$. The algorithm keeps on doing this until either the queue becomes empty in which case the CSP is arc-consistent or one of the domains becomes empty in which case support-checks can be saved because the CSP cannot be made arc-consistent.

Not only are there heuristics for AC-3 to remove members from the queue, but also there are heuristics which, given a constraint, select the values in the domains of the variables that will be used for the support-checks. Such heuristics we will call *domain-heuristics*.

Empirical results from Wallace and Freuder clearly indicate that arc-heuristics influence the average performance of arc-consistency algorithms [Wallace and Freuder, 1992]. Wallace presents empirical evidence that the average time required by AC-3 is better than the average time required by AC-4 [Wallace, 1993]. AC-4 is an arc-consistency which has an optimal $\mathbf{O}\,(ed^2)$ worst case time-complexity [Mohr and Henderson, 1986].

The major drawback of AC-3 is that it cannot remember the support-checks it has already carried out and—as a consequence—repeats some of them. Bessière, Freuder and Régin present another class of arc-consistency algorithms called AC-7 [Bessière *et al.*, 1995]. AC-7 is an instance of the AC-INFERENCE schema, where support-checks are saved by making inference. In the case of AC-7 inference is made at domain-level, where it is exploited that $M_{ij} = M_{ji}^T$, where $\cdot^T$ denotes transposition. AC-7 has an optimal upper bound of $\mathbf{O}\,(ed^2)$ for its worst case time-complexity and has been reported to behave well on average.

The most prominent data-structures of AC-7 are a *deletion-stream* and a *seek-support-stream*. The purpose of the deletion-stream is to propagate the consequences of the removal of a value from the domain of one of the variables. The seek-support-stream contains tuples of the form $(\,(\,\alpha, i\,), \beta\,)$, where α and β are variables and $i \in D(\alpha)$. A tuple $(\,(\,\alpha, i\,), \beta\,)$ in the seek-support-stream represents the fact that support for the value $i \in D(\alpha)$ has to be found with some value in $D(\beta)$. Heuristics to select members from the seek-support-stream have effects on the number of support-checks that are required and the order in which they are carried out.

AC-7's heuristics for the selection of tuples from its seek-support-stream are a combination of arc-heuristics and domain-heuristics. However, because AC-7 uses inference, not every removal of every tuple from the seek-support-stream will result in an actual support-check. Since a double-support heuristic is but a special example of a heuristic, the heuristics for a particular implementation of AC-7 could be such that the domain-level component could (partially) depend on a double-support heuristic.

In their paper, Bessière, Freuder and Régin present empirical results that the AC-7 approach is superior to the AC-3 approach. They present results of applications of MAC-3 and MAC-7 to real-world problems. Here MAC-i is a backtracking algorithm which uses AC-i to maintain arc-consistency during search [Sabin and Freuder, 1994]. Unfortunately, it is not reported which members of the classes they use for their comparison, i.e. it is not reported which heuristics they used for AC-3 and AC-7 and their experiments cannot be repeated to get the same results.

In the previous chapter we presented an empirical comparison between AC-7 and AC-3_b which is a cross-breed between Mackworth's AC-3 and Gaschnig's DEE [Gaschnig, 1978]. At the domain-level AC-3_b uses a double-support heuristic. AC-3_b has the same worst-case time-complexity as AC-3. In the experimental setting of Chapter 6 it turned out that AC-3_b was more efficient than AC-7 in certain parts of the tightness/density spectrum. The results are an indication that it is possible to use domain-heuristics to improve the performance of arc-consistency algorithms.

7.2.3 The General Problem

In this section we shall discuss the reasons for, and the consequences of, our decision to study only two-variable CSPs. Also we will make some general comments about the presentation of our algorithms further on in this chapter.

One problem with our choice is that we have eliminated the effects that arc-heuristics have on arc-consistency algorithms. Wallace and Freuder showed that arc-heuristics have effects on performance [Wallace and Freuder, 1992]. Our study does not properly take the effects of arc-heuristics into account. However, later in this section we will argue that a double-support heuristic should be used at domain-level.

Another problem with our simplification is that we cannot properly extrapolate average results for two-variable CSPs to the case where arc-consistency algorithms are used as part of MAC-algorithms. For example, in the case of a two-variable CSP, on average about one out of every two support-checks will succeed. This is not true in MAC-search because most time is spent at the leaves of the search-tree and most support-checks in that region will fail. A solution would be to refine our analysis to the case where different ratios of support-checks succeed.

We justify our decision to study two-variable CSPs by two reasons. Our first reason is that at the moment the general problem is too complicated. We have studied a simpler problem hoping that it would provide insight to the successfulness of the the double-support heuristic from the previous chapter.

Our second reason to justify our decision to study two-variable CSPs is that we argue that at domain-level a double-support heuristic is a good choice and that it can be studied independent from an arc-heuristic. We assume that support-checks are not repeated.

Our reasoning is as follows. To compute an arc-consistent CSP we have to find out for each value in the domain of each of the variables if it is supported. We can only decide if a value

$i \in D(\alpha)$ is inherently unsupportable if we find a constraint between α and another variable β and use checks for *each* of the values in $D(\beta)$ that were not known to support i. In other words, if i is inherently unsupportable then *any* domain-heuristic is a good choice. If i is inherently supportable then for each constraint between α and β we should not only find a support as soon as possible but also find a support with a value in $D(\beta)$ whose support-status was not yet known, i.e. we should use double-support checks involving i. After all, if a double-support check succeeds it will provide more information about which values are supported by the constraint. This information can be used for the purpose of making inference and for the purpose of "guiding" heuristics. In other words, regardless of the inherent supportability of i, a double-support heuristic is a good choice to complement any arc-heuristic. Observe that our reasoning was independent of the choice of the arc-heuristic that was used. We can probably study the double-support heuristic by studying it for the case where the CSP is a two-variable CSP.

7.3 Two Arc-Consistency Algorithms

In this section we shall introduce two arc-consistency algorithms and present a detailed case study where we shall compare the average time-complexity of the two algorithms for the case where the domain size of both variables is two. The two algorithms differ in their domain-heuristic. The algorithms under consideration are a *lexicographical algorithm* and a *double-support algorithm*. The lexicographical algorithm will be called $\mathcal{L}$. The double-support algorithm will be called $\mathcal{D}$. As part of our presentation, we shall point out three different reasons which, from an intuitive point of view, suggest that arc-consistency algorithms should give preference to double-support checks at domain-level. We shall see that for the problem under consideration $\mathcal{D}$ outperforms $\mathcal{L}$.

The remainder of this section is as follows. In Section 7.3.1 we shall define $\mathcal{L}$ and examine its average time-complexity for two by two matrices. In Section 7.3.2 we shall define $\mathcal{D}$ and examine its average time-complexity for two by two matrices. As part of the examination process we will point out three reasons which suggest that arc-consistency algorithms should give preference to double-support checks. Finally, in Section 7.3.3, we shall compare the two algorithms.

7.3.1 The Lexicographical Algorithm $\mathcal{L}$

In this section we shall define the *lexicographical arc-consistency algorithm* called $\mathcal{L}$ and discuss its application to two by two matrices. We shall first define $\mathcal{L}$ and then discuss the application. We will not be concerned about the data-structures used in implementations of $\mathcal{L}$. Instead, it is our intention to present algorithms such that their essence becomes clear. In our presentation we use an ALGOL-ish pseudo-language which comes with a "`forall` $v \in S$ `do statements od`" iteration-construct. The semantics of the construct are that for each $s \in S$ it assigns s to v and carries out `statements`. The order in which the members of S are assigned to v is the same as the lexicographical order on the members of S.

In the presentations of our algorithms we will distinguish between dereferencing a constraint and dereferencing other matrices. This is important because the number of times we derefer-

```
function ℒ(M, a, b) :
    /* initialisation */
    row-support = ∅;
    column-support = ∅;
    forall i ∈ { 1, . . . , a } do
        forall j ∈ { 1, . . . , b } do
            C_ij ≔unknown;
        od;
    od;
    forall i ∈ { 1, . . . , a } do
        /* try to establish support for i */
        j ≔ 0;
        while (j < b) and (i ∉ row-support) do
            /* find lexicographically smallest j that supports i */
            j ≔ j + 1;
            C_ij ≔ M_ij;
            if (C_ij = 1) then
                row-support ≔ row-support ∪ { i };
                column-support ≔ column-support ∪ { j };
            fi;
        od;
    od;
    forall j ∈ { 1, . . . , b } \ column-support do
        /* try to establish support for j */
        i ≔ 0;
        while (i < a) and (j ∉ column-support) do
            /* find lexicographically smallest i that supports j */
            i ≔ i + 1;
            if (C_ij = unknown) then
                if (M_ij = 1) then
                    column-support ≔ column-support ∪ { j };
                fi;
            fi;
        od;
    od;
    return ( row-support, column-support );
od;
```

Figure 7.1: The lexicographical algorithm $\mathcal{L}$.

ence a constraint determines the time-complexity of the algorithms, whereas dereferencing other matrices does not.

Definition 7.8 (Lexicographical Arc-Consistency Algorithm). Let a and b be positive integers, and let $M \in \mathbb{M}^{ab}$. The *lexicographical* arc-consistency algorithm is the algorithm $\mathcal{L}$ defined in Figure 7.1.

It is important to point out that we distinguish between support-checks and matrix look-ups. Only the checks M_{ij} contribute to the total number of support-checks, whereas the look-ups C_{ij} do not.

$\mathcal{L}$ does not repeat support-checks. $\mathcal{L}$ first tries to establish its row-support. It does this for each row in the lexicographical order on the rows. When it seeks support for row i it tries to find the lexicographical smallest column which supports i. After $\mathcal{L}$ has computed its row-support, it tries to find support for those columns whose support-status is not yet known. It does this in the lexicographical order on the columns. When $\mathcal{L}$ tries to find support for a column j, it tries to find it with the lexicographically smallest row that was not yet known to support j.

Example 7.9 ($\mathcal{L}$). Let $M \in \mathbb{M}^{33}$ be the matrix whose first row is $[\ 0 \quad 1 \quad 1\ ]$, whose second row is $[\ 0 \quad 0 \quad 0\ ]$, and whose last row is $[\ 1 \quad 1 \quad 0\ ]$. In order to find the support of M the following support-checks are carried out by $\mathcal{L}$ in their order of appearance: $M_{11}^?$, $M_{12}^?$, $M_{21}^?$, $M_{22}^?$, $M_{23}^?$, $M_{31}^?$, $M_{13}^?$. The trace of M with respect to $\mathcal{L}$ is given by $(\,1,1,0\,)$, $(\,1,2,1\,)$, $(\,2,1,0\,)$, $(\,2,2,0\,)$, $(\,2,3,0\,)$, $(\,3,1,1\,)$, $(\,1,3,1\,)$.

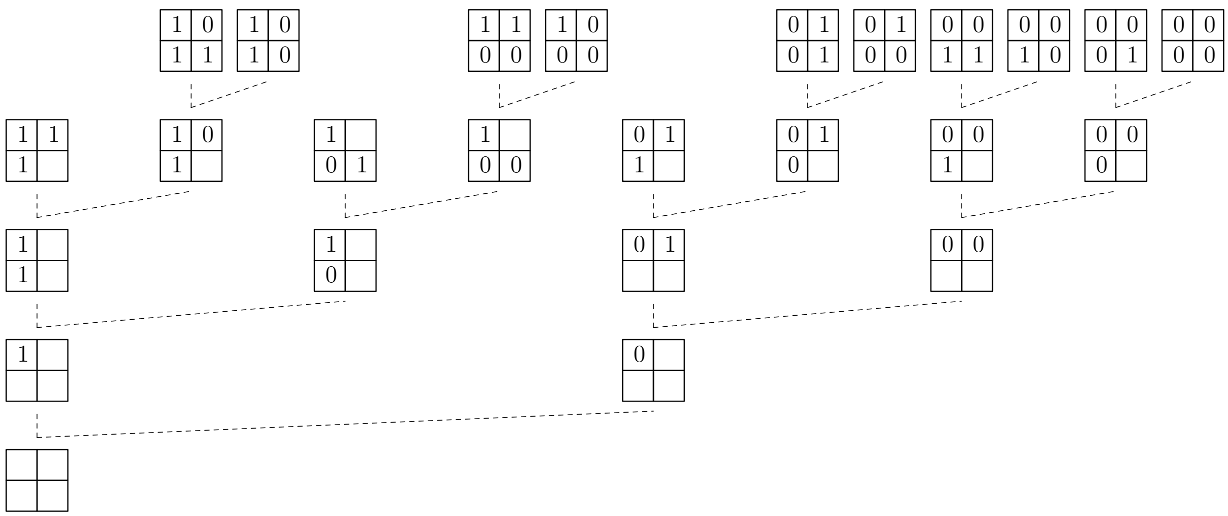

Figure 7.2: Traces of $\mathcal{L}$. Total number of support-checks is $16 \times 4 - 6 \times 1 = 58$.

Figure 7.2 is a graphical representation of all traces with respect to $\mathcal{L}$. Each different path from the root to a leaf corresponds to a different trace with respect to $\mathcal{L}$. Each trace of length l is represented in the tree by some unique path that connects the root and some leaf via $l-1$ internal nodes. The root of the tree is an artificial 0-th member of the traces. The nodes/leaves at distance l from the root correspond to the l-th members of the traces, for $0 \le l \le ab = 4$.

Nodes in the tree are decision points. They represent the support-checks which are carried out by $\mathcal{L}$. A branch of a node n that goes straight up represents the fact that a support-check, say $M_{ij}^?$, was successful. A branch to the right of that same node n represents the fact that the same $M_{ij}^?$ was unsuccessful. The successful and unsuccessful support-checks are also represented at node-level. The i-th row of the j-th column of a node does not contain a number if the check $M_{ij}^?$ has not been carried out. Otherwise, it contains the number M_{ij}. It is only by studying the nodes that it can be found out which support-checks have been carried out so far.

Example 7.10 (Trace). The path in Figure 7.2 from the root of the tree to the second leaf from the right represents the trace $t_1 = (\,1,1,0\,),(\,1,2,0\,)(\,2,1,0\,),(\,2,2,1\,)$. The path from the root to the leftmost leaf corresponds to the trace $t_2 = (\,1,1,1\,),(\,2,1,1\,),(\,2,2,1\,)$. There is only one two by two zero-one matrix whose trace is t_1. There are two two by two zero-one matrices whose trace is given by t_2.

Remember that we denote the number of rows by a and the number of columns by b. It is important to understand that there are $2^{ab} = 2^4 = 16$ different two by two zero-one matrices and that the traces of different matrices with respect to $\mathcal{L}$ can be the same. To determine the average time-complexity of $\mathcal{L}$ we have to add the lengths of the traces of each of the matrices and divide the result by 2^{ab}. Alternatively, we can compute the average number of support-checks if we subtract from ab the sum of the *average savings* of each the matrices, where the savings of a matrix are given by $ab - l$ and its average savings are given by $(ab - l)/2^{ab}$, where l is the length of the trace of the matrix with respect to $\mathcal{L}$. Similarly, we define the average savings of a trace as the sum of the average savings of all the matrices that have that trace.

It is interesting to notice that all traces of $\mathcal{L}$ have a length of at least three. Notice that, $\mathcal{L}$ is not capable to determine its support in fewer than three support-checks—not even if a matrix does not contain any zero at all. It is not difficult to see that $\mathcal{L}$ will always require at least $a+b-1$ support-checks.

$\mathcal{L}$ could only have terminated with two checks had both these checks been successful. If we focus on the strategy $\mathcal{L}$ uses for its second support-check for the case where its first support-check was successful we shall find the reason why it cannot accomplish its task in fewer than three checks. After $\mathcal{L}$ has successfully carried out its first check $M^?_{11}$ it needs to learn only *two* things. It needs to know if 2 is in the row-support and it needs to know if 2 is in the column-support. The next check of $\mathcal{L}$ is $M^?_{21}$. Unfortunately, this check can only be used to learn *one* thing. *Regardless of whether the check $M^?_{12}$ succeeds or fails*, another check *has* to be carried out.

If we consider the case where the check $M^?_{22}$ was carried out as the second support-check we shall find a more efficient way of establishing the support. The check $M^?_{22}$ offers the potential of learning about *two* new things. If the check is successful then it offers the knowledge that 2 is in the row-support and that 2 is in the column-support. Since this was all that had to be found out the check $M^?_{22}$ offers the potential of termination after two support-checks. What is more, one out of every two such checks will succeed. Only if the check $M^?_{22}$ fails do more checks have to be carried out. Had the check $M^?_{22}$ been used as the second support-check, checks could have been saved on average.

Remember that the same trace in the tree can correspond to different matrices. The Trace Theorem states that if l is the length of a trace then there are exactly 2^{ab-l} matrices which have the same trace. The shortest traces of $\mathcal{L}$ are of length $l_1 = 3$. $\mathcal{L}$ finds exactly $s_1 = 3$ traces whose lengths are l_1. The remaining l_2 traces all have length $l_2 = ab$. Therefore, $\mathcal{L}$ saves $(s_1 \times (ab - l_1) \times 2^{ab-l_1} + s_2 \times (ab - l_2) \times 2^{ab-l_2})/2^{ab} = (3 \times (4-3) \times 2^{4-3} + 0)/2^4 = 3 \times 1 \times 2^1/2^4 = 3/8$ support-checks on average. The strategy of $\mathcal{L}$ therefore requires an average number of support-checks of $ab - \frac{3}{8} = 4 - \frac{3}{8} = 3\frac{5}{8}$. In the next section we shall see that better strategies than that of $\mathcal{L}$ exist.

7.3.2 The Double-Support Algorithm $\mathcal{D}$

In this section we shall introduce a second arc-consistency algorithm and analyse its average time-complexity for the special case where the number of rows a and the number of columns b are both two. The algorithm will be called $\mathcal{D}$. It uses a heuristic to select its support-checks based on the notion of a *double-support check*.

The organisation of the remainder of this section is as follows. First, we shall formally define the notion of a double-support check and two other related notions. Next, we shall define $\mathcal{D}$ and analyse it for the two by two problem under consideration. As part of our analysis, we will identify three reasons which, from an intuitive point of view, suggest that arc-consistency algorithms should prefer double-support checks to other checks at domain-level.

Definition 7.11 (Zero-Support Check). Let M be an a by b matrix. A support-check $M^?_{ij}$ is called a *zero-support check* if, just before the check was carried out, the row-support status of i and the column-support status of j were known.

A zero-support check is a support-check from which nothing new can be learned about the support of a matrix. Good arc-consistency algorithms should therefore never carry out zero-support checks.

Definition 7.12 (Single-Support Check). Let M be an a by b matrix. A support-check $M^?_{ij}$ is called a *single-support check* if, just before the check was carried out, the row-support status of i was known and the column-support status of j was unknown, or vice versa.

A successful single-support check $M^?_{ij}$ leads to new knowledge about one thing. Either it leads to the knowledge that i is in the row-support of M where this was not known before the check was carried out, or it leads to the knowledge that j is in the column-support of M where this was not known before the check was carried out.

Definition 7.13 (Double-Support Check). Let M be an a by b matrix. A support-check $M^?_{ij}$ is called a *double-support check* if, just before the check was carried out, both the row-support status of of i and the column-support status of j were unknown.

A successful double-support check, say $M^?_{ij}$, leads to new knowledge about two things. It leads to the knowledge that i is in the row-support of M and that j is in the column-support of M where neither of these facts was known to be true just before the check was carried out.

Single-support checks, provided they are successful, lead to knowledge about one new thing at the price of one support-check. Double-support checks, provided they are successful, lead to knowledge about two new things at the price of one support-check. On average it is just as likely that a double-support check will succeed as it is that a single-support check will succeed—in both cases one out of two checks will succeed on average. The potential payoff of a double-support check is twice as large that that of a single-support check. This is our first indication that at domain-level arc-consistency algorithms should prefer double-support checks to single-support checks.

Our second indication that arc-consistency algorithms should prefer double-support checks to single-support checks is the insight that in order to minimise the total number of support-checks it is a necessary condition to maximise the number of successful double-support checks.

Later in this section we will point out a third indication—more compelling than the previous two—that arc-consistency algorithms should prefer double-support checks to single-support checks.

Definition 7.14 (Double-Support Arc-Consistency Algorithm). Let a and b be positive integers and $M \in \mathbb{M}^{ab}$. The *double-support* arc-consistency algorithm is the algorithm $\mathcal{D}$ defined in Figure 7.3.

The strategy used by $\mathcal{D}$ is a bit more complicated than that of $\mathcal{L}$. It will first try to use double-support checks to find support for its rows in the lexicographical order on the rows. It does this by finding for every row the lexicographically smallest column whose support-status is not yet known. When there are no more double-support checks left, $\mathcal{D}$ will use single-support checks to find support for those rows whose support-status is not yet known and then find support for those columns whose support status is still not yet known. When it seeks support for a row/column, it tries to find it with the lexicographically smallest column/row that is not yet known to support that row/column.

We have depicted the traces of $\mathcal{D}$ in Figure 7.4. It may not be immediately obvious, but the strategy of $\mathcal{D}$ is more efficient than that of $\mathcal{L}$. The reason for this is as follows. There are two traces whose length is shorter than $ab = 4$. There is one such trace whose length is $l_1 = 2$ and one such trace whose length is $l_2 = 3$. The remaining s_3 traces each have a length of $l_3 = ab$. Using the Trace Theorem we can use these findings to determine the number of support-checks that are saved on average. The average number of savings of $\mathcal{D}$ are given by $(s_1 \times (ab - l_1) \times 2^{ab-l_1} + s_2 \times (ab - l_2) \times 2^{ab-l_2} + s_3 \times (ab - l_3) \times 2^{ab-l_3})/2^{ab} = (2 \times 2^2 + 1 \times 2^1 + 0)/2^4 = 5/8$. This saves $1/4$ checks more on average than $\mathcal{L}$.

It is important to observe that l_1 has a length of only two and that it is the result of a sequence of two successful double-support checks. It is this trace which contributed the most to the savings. As a matter of fact, this trace by itself saved more than the *total* savings of $\mathcal{L}$.

The strategy used by $\mathcal{D}$ to prefer double-support checks to single-support checks leads to shorter traces. We can use the Trace Theorem to find that that the savings of a trace are of the form $(ab - l)2^{ab-l}$, where l is length of the trace. The double-support algorithm was able to produce a trace that was smaller than any of those produced by the lexicographical algorithm. To find this trace had a big impact on the total savings of $\mathcal{D}$. The reason why $\mathcal{D}$ was able to find the short trace was because it was the result of a sequence of successful double-support checks and its heuristic forces it to use as many double-support checks as it can. Traces which contain many successful double-support checks contribute much to the total average savings. This is our third and last indication that arc-consistency algorithms should prefer double-support checks over single-support checks.

7.3.3 A First Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we have compared the average time-complexity of the lexicographical algorithm $\mathcal{L}$ and the double-support algorithm $\mathcal{D}$ for the case where the size of the domains is two. We have found that the double-support algorithm was more efficient on average than the lexicographical algorithm for the problem under consideration.

We have been able to identify three reasons which, from an intuitive point of view, suggest that arc-consistency algorithms should prefer double-support checks to single-support checks. The first reason is that a double-support check has a pay-off which is twice as much. If a dou-

```
function D(M, a, b) :
    /* initialisation */
    row-support := ∅;
    column-support := ∅;
    forall i ∈ { 1, ..., a } do forall j ∈ { 1, ..., b } do C_ij :=unknown; od; od;
    forall i ∈ { 1, ..., a } do
        /* try to establish support for i using double-support checks */
        j := 0;
        while (j < b) and (i ∉ row-support) do
            /* try to find lexicographically smallest j that supports i */
            j := j + 1;
            if (j ∉ column-support) then
                C_ij := M_ij;
                if (C_ij = 1) then
                    row-support := row-support ∪ { i };
                    column-support := column-support ∪ { j };
                fi;
            fi;
        od;
    od;
    forall i ∈ { 1, ..., a } \ row-support do
        /* try to establish support for i using single-support checks */
        j := 0;
        while (j < b) and (i ∉ row-support) do
            /* try to find lexicographically smallest j that supports i */
            j := j + 1;
            if (C_ij = unknown) then
                C_ij := M_ij;
                if (C_ij = 1) then
                    row-support := row-support ∪ { i };
                fi;
            fi;
        od;
    od;
    forall j ∈ { 1, ..., b } \ column-support do
        /* try to establish support for j using single-support checks */
        i := 0;
        while (i < a) and (j ∉ column-support) do
            /* try to find lexicographically smallest i supporting j */
            i := i + 1;
            if (C_ij = unknown) then
                C_ij := M_ij;
                if (C_ij = 1) then
                    column-support := column-support ∪ { j };
                fi;
            fi;
        od;
    od;
    return ( row-support, column-support );
od;
```

Figure 7.3: The double-support algorithm $\mathcal{D}$.

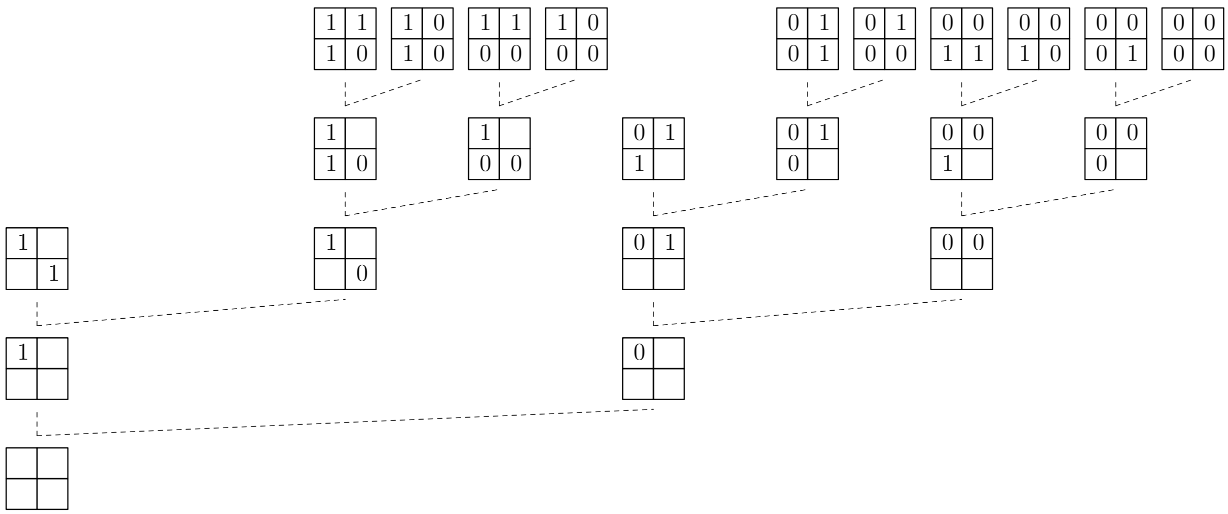

Figure 7.4: Traces of $\mathcal{D}$. Total number of support-checks is $16 \times 4 - 4 \times 2 - 2 \times 1 = 54$.

ble-support check is successful two things are learned in return for only one support-check, as opposed to only one new thing for a successful single-support check. The second reason is that it is a necessary condition to maximise the number of successful double-support checks in order to minimise the total number of support-checks. The third and last reason is that the average savings of a trace are of the form $(ab - l)2^{-l}$, where l is the length of the trace. The shorter the trace, the bigger the savings. Only traces that contain many successful double support-checks can become very small and thus lead to big savings. To find many such traces requires a heuristic which gives preference to double-support checks.

In the following sections we will provide solid mathematical evidence that the strategy used by $\mathcal{D}$ is superior to that used by $\mathcal{L}$.

7.4 Average Time-Complexity of $\mathcal{L}$

In this section we shall investigate the average time-complexity of the lexicographical algorithm. The organisation of this section is as follows. First, we shall define the notions of *left* and *right* support-checks. Next, we shall determine the average time-complexity of $\mathcal{L}$ by computing the average number of left and right support-checks of $\mathcal{L}$ and use them to establish an exact formula for the average time-complexity of $\mathcal{L}$. Finally, we shall determine simple upper and lower bounds for the average time-complexity of $\mathcal{L}$.

$\mathcal{L}$ establishes its support in two phases. In its first phase $\mathcal{L}$ tries to establish its row-support. In its second phase $\mathcal{L}$ carries out the remaining work to find the column-support. In the following, we will call the checks that are carried out in the first phase the *left support-checks*. The checks that are carried out in the second phase will be called the *right support-checks*.

The following lemma will be useful in our derivation of the average time-complexity for left support-checks further on.

Lemma 7.15. *Let b be a positive integer. Then*

$$\sum_{c=1}^{b} c2^c = 2 + (b-1)2^{b+1}.$$

Proof. Let $f(b) = \sum_{c=1}^{b} c2^c$, and let $g(b) = 2 + (b-1)2^{b+1}$. We have to prove that $f(b) = g(b)$ for all positive integers b. We have $f(1) = g(1) = 2$. For every positive b we have:

$$\begin{aligned} f(b+1) - f(b) &= (b+1)2^{b+1} \\ &= b2^{b+1} + 2^{b+1} \\ &= b2^{b+2} - (b-1)2^{b+1} \\ &= 2 + ((b+1)-1)2^{(b+1)+1} - (2 + (b-1)2^{b+1}) \\ &= g(b+1) - g(b). \end{aligned}$$

Since $f(1) = g(1)$ and $f(b+1) - f(b) = g(b+1) - g(b)$ for every $b > 1$ it follows that $f(b) = g(b)$ for all $b > 0$. □

The following relates b and the sum of the left support-checks of a row of length b.

Lemma 7.16 (Left Support-Checks for Single Rows). *Let b be a positive integer, let $M \in \mathbb{M}^{1b}$ be a 1 by b matrix, and let $L_{\mathcal{L}}^{M}$ be the number of support-checks required by $\mathcal{L}$ to determine the row-support of M. Then*

$$\sum_{M \in \mathbb{M}^{1b}} L_{\mathcal{L}}^{M} = 2^{b+1} - 2.$$

Proof. If a row has $c-1$ leading zeros followed by a one then c checks have to be carried out. There are 2^{b-c} rows that have $c-1$ leading zeros followed by a one. If a row is zero then b checks have to be carried out. Therefore, the number of left support-checks is $b + \sum_{c=1}^{b} c2^{b-c}$. The remainder of the proof consists of the simplification of this expression.

$$\begin{aligned} \sum_{M \in \mathbb{M}^{1b}} L_{\mathcal{L}}^{M} &= b + \sum_{c=1}^{b} c2^{b-c} \\ &= b + \sum_{c=0}^{b} (b-c)2^c \\ &= b + (\sum_{c=0}^{b} b2^c) - (\sum_{c=0}^{b} c2^c) \\ &= b + b(2^{b+1} - 1) - (\sum_{c=0}^{b} c2^c) \\ &= b2^{b+1} - (b2^{b+1} - 2^{b+1} + 2) \end{aligned}$$

The last equality holds because $\sum_{c=0}^{b} c2^c = \sum_{c=1}^{b} c2^c$, which is equal to $b2^{b+1} - 2^{b+1} + 2$ by Lemma 7.15. The proof ends by observing that

$$b2^b - (b2^{b+1} - 2^{b+1} + 2) = 2^{b+1} - 2. \qquad \square$$

The following relates a, b and the sum of the left support-checks of all a by b matrices.

Lemma 7.17 (Left Support-Checks by $\mathcal{L}$). *Let a and b be positive integers. Let $L_{\mathcal{L}}^{M}$ be the number of support-checks required by $\mathcal{L}$ to determine the row-support of M. Then*

$$\sum_{M \in \mathbb{M}^{ab}} L_{\mathcal{L}}^{M} = a(2 - 2^{1-b})2^{ab}.$$

Proof. We can use Lemma 7.16 to count the number of checks that have to be spent on a row at a fixed row r in a matrix. This number is given by $2^{b+1} - 2$. For each sequence of length b that consists of ones and zeroes there are exactly $2^{b(a-1)}$ different matrices where this sequence occurs in the row at the same fixed position r. Since there are a rows and since we can count the checks spent on each of the different rows independently, the total number of checks spent on a rows of length b is the same as a times the total number of checks spent on one row of length b. Therefore,

$$\begin{aligned}\sum_{M \in \mathbb{M}^{ab}} L_{\mathcal{L}}^{M} &= (2^{b+1} - 2)a2^{b(a-1)} \\ &= a(2 - 2^{1-b})2^{ab},\end{aligned}$$

which ends the proof. $\square$

The following lemma provides a lower bound for any arc-consistency algorithm.

Lemma 7.18 (Lower Bound). *Let a and b be positive integers, and let $\mathcal{A}$ be any arc-consistency algorithm then*

$$\max(a, b)(2 - 2^{1-\min(a,b)}) \leq \operatorname{avg}_{\mathcal{A}}(a, b).$$

Proof. Assume $b \leq a$. It is obvious that $\operatorname{avg}_{\mathcal{A}}(a, b)$ is at least as much as the average number of support-checks that have to be carried out to find the row-support. The average number of support-checks to find the row-support are at least $a(2 - 2^{1-b})$ (the number of left support-checks). The case for $a < b$ is analogous. $\square$

The following relates a, b and the sum of the right support-checks of all a by b matrices.

Lemma 7.19 (Right Support-Checks by $\mathcal{L}$). *Let a and b be positive integers, let $M \in \mathbb{M}^{ab}$ be an a by b matrix, and let $R_{\mathcal{L}}^{M}$ be the number of support-checks required by $\mathcal{L}$ to determine the remainder of the column-support of M after it has established the row-support of M. Then*

$$\sum_{M \in \mathbb{M}^{ab}} R_{\mathcal{L}}^{M} = 2^{ab}(2^{1-a}(1 - b) + 2\sum_{c=2}^{b}(1 - 2^{-c})^a).$$

Proof. A support-check should only be carried out if there are no rows $r < r'$ such that $M_{r'c} = 1$ and $M_{r'c'} = 0$ for $1 \leq c' < c$ and no rows of the form $r'' < r$ such that $M_{r''c} = 1$. Therefore,

$$\begin{aligned}
\sum_{M \in \mathbb{M}^{ab}} R_{\mathcal{L}}^{M} &= \sum_{c=2}^{b} \sum_{r=1}^{a} 2^{a(b-c)} 2^{(r-1)(c-1)} (2^c - 2)(2^c - 1)^{a-r} \\
&= \sum_{c=2}^{b} 2^{a(b-c)} (2^c - 2)(2^c - 1)^{a-1} \sum_{r=1}^{a} 2^{(r-1)(c-1)} (2^c - 1)^{1-r} \\
&= \sum_{c=2}^{b} 2^{a(b-c)+1} (2^{c-1} - 1)(2^c - 1)^{a-1} \sum_{r=0}^{a-1} (2^{c-1}/(2^c - 1))^r. \qquad (7.2)
\end{aligned}$$

Note that in the inner summation of Equation (7.2) the value of c is always greater than one. Therefore, $2^{c-1}/(2^c - 1) \neq 1$ for all the c that are under consideration. The inner summation turns out to be the sum of a geometric series and we can simplify it as follows:

$$\begin{aligned}
\sum_{r=0}^{a-1} (2^{c-1}/(2^c - 1))^r &= (1 - 2^{a(c-1)} (2^c - 1)^{-a})(1 - 2^{c-1} (2^c - 1)^{-1})^{-1} \\
&= (2^c - 1)(2^{c-1} - 1)^{-1} (1 - 2^{a(c-1)} (2^c - 1)^{-a})).
\end{aligned}$$

This allows us to continue to simplify Equation (7.2) as follows:

$$\begin{aligned}
&\sum_{c=2}^{b} 2^{a(b-c)+1} (2^{c-1} - 1)(2^c - 1)^{a-1} \sum_{r=0}^{a-1} (2^{c-1}/(2^c - 1))^r \\
&= \sum_{c=2}^{b} 2^{a(b-c)+1} (2^c - 1)^a (1 - 2^{a(c-1)} (2^c - 1)^{-a})) \\
&= 2^{ab} \sum_{c=2}^{b} (2(1 - 2^{-c})^a - 2^{1-a}) \\
&= 2^{ab} (2^{1-a}(1 - b) + 2 \sum_{c=2}^{b} (1 - 2^{-c})^a),
\end{aligned}$$

which concludes the proof of Lemma 7.19. □

We are finally in a position where we can determine the average time complexity of $\mathcal{L}$.

Theorem 7.20 (Average Time Complexity of $\mathcal{L}$). *Let a and b be positive integers. The average time complexity of $\mathcal{L}$ over $\mathbb{M}^{ab}$ is given by the function $\operatorname{avg}_{\mathcal{L}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, where*

$$\operatorname{avg}_{\mathcal{L}}(a, b) = a(2 - 2^{1-b}) + (1 - b)2^{1-a} + 2 \sum_{c=2}^{b} (1 - 2^{-c})^a. \qquad (7.3)$$

Proof. Divide the result of the addition of the left and right support-checks of $\mathcal{L}$ by 2^{ab}. □

We have verified Theorem 7.20 as follows. For each combination of a and b such that $1 \leq a, b \leq 5$ we have computed $\sum_{M \in \mathbb{M}^{ab}} \text{checks}_{\mathcal{L}}(M)$ by applying $\mathcal{L}$ to all a by b constraints and by keeping track of the total number of support-checks that were required for each combination of a and b. For each combination of a and b we have been able to verify that $\sum_{M \in \mathbb{M}^{ab}} \text{checks}_{\mathcal{L}}(M)$ was exactly $\sum_{M \in \mathbb{M}^{ab}}(L_{\mathcal{L}}^M + R_{\mathcal{L}}^M)$ and that $\text{avg}_{\mathcal{L}}(a, b)$ was exactly $\sum_{M \in \mathbb{M}^{ab}} \text{checks}_{\mathcal{L}}(M)/2^{ab}$. Our theoretical results for a by b constraints turned out to be exact, for $1 \leq a, b \leq 5$.

The following proposition will allow us to provide a neat bound for $\text{avg}_{\mathcal{L}}(a, b)$. The proposition is adapted from [Flajolet and Sedgewick, 1996, Proposition 7.9, p. 59].

Proposition 7.21 (Longest 1-Run). *Let* $\bar{L}_a = \sum_{c=0}^{\infty}(1 - (1 - 2^{-c-1})^a)$*, then*

$$\bar{L}_a = \log_2(a) + \frac{\gamma}{\log(2)} - \frac{1}{2} + \frac{1}{\log(2)} \sum_{c \in \mathbb{Z} \setminus \{ 0 \}} \Gamma\left(\frac{2ic\pi}{\log(2)}\right) e^{-2ic\pi \log_2(a)} + \mathbf{O}\left(\frac{1}{\sqrt{a}}\right).$$

Here, $\log_2(\cdot)$ *is the base-2 logarithm,* $\log(\cdot)$ *is the natural logarithm,* $\gamma \approx 0.577216$ *is Euler's gamma constant, and* $\Gamma(\cdot)$ *is the Gamma function. Thus,* $\bar{L}_a$ *is about* $\log_2(a) + 0.332746$ *as* a *becomes large.*

The following follows immediately from Proposition 7.21.

Corollary 7.22. *Let* $\bar{L}_a^b = \sum_{c=0}^{b}(1-(1-2^{-c})^a)$*, then* $\bar{L}_a^b = \bar{L}_a + 1$ *as* b *becomes large. Therefore,* $\bar{L}_a^b$ *is about* $\log_2(a) + 1.332746$ *as* a *and* b *become large.*

We are finally in a position to provide a "nice" expression for $\text{avg}_{\mathcal{L}}(a, b)$.

Corollary 7.23. *The average time-complexity of* $\mathcal{L}$ *is about* $2a + 2b - 2log_2(a) - 0.665492$.

Proof. By Theorem 7.20 we have

$$\begin{aligned}
\text{avg}_{\mathcal{L}}(a, b) &= a(2 - 2^{1-b}) + (1 - b)2^{1-a} + 2\sum_{c=2}^{b}(1 - 2^{-c})^a \\
&= a(2 - 2^{1-b}) - b2^{1-a} + 2\sum_{c=0}^{b}(1 - 2^{-c})^a \\
&= a(2 - 2^{1-b}) + b(2 - 2^{1-a}) + 2 - 2\sum_{c=0}^{b}(1 - (1 - 2^{-c})^a). \qquad (7.4)
\end{aligned}$$

Notice that if b becomes large the sum reduces to the sum $\bar{L}_a^b$ from Corollay 7.22. Therefore, $\text{avg}_{\mathcal{L}}(a, b)$ is about $2a + 2b - 2log_2(a) - 0.665492$. □

The bound from Corollay 7.23 is interesting because if a and b are of the same magnitude and become large then $\text{lwb}_{\mathcal{L}}(a, b)$ is "almost" of the form $2a + 2b - 2\log_2(a)$. This seems to suggest that $\mathcal{L}$ requires about two checks for each of the members in the domains of each of the variables. Given that on average every second entry of a matrix is a one, this seems to suggest that $\mathcal{L}$ cannot use checks that are used to learn about its row-support to also learn about its column-support and vice versa. Otherwise, the lower bound of $\mathcal{L}$ would not have included coefficients which are (approximately) 2 for *both* a *and* b. It is as if $\mathcal{L}$ carries out the checks to find its row-support on the one hand and the checks to find its column-support on the other almost completely independently from each other.

We can also explain the results we have obtained for the bound in Corollay 7.23 in the following way. $\mathcal{L}$ has to establish support for each of its a rows and b columns except for the l columns which were found to support a row when $\mathcal{L}$ was establishing its row-support. Therefore, $\mathcal{L}$ requires about $2a + 2(b - l)$ checks on average. To find l turns out to be easy. Assume that $a = 2^k$ for some integer $k > 1$. On average $a/2$ rows will be supported by the first column. From the remaining $a/2$ rows on average $a/4$ rows will be supported by the second column, ..., from the remaining 2 rows on average 1 will find support with the $\log_2(a)$-th column, i.e. $l \approx \log_2(a)$. If a does not have the special form 2^k then l will still be about $\log_2(a)$. This informal reasoning demonstrates that on average $\mathcal{L}$ will require about $2a + 2b - 2\log_2(a)$ support-checks and this is almost exactly what we found in Corollay 7.23.

7.5 Average Time-Complexity of $\mathcal{D}$

In this section we shall derive the average time-complexity of $\mathcal{D}$. It will turn out that this is a bit easier than the complexity analysis carried out in the previous section. As part of our analysis we will demonstrate that if $a + b \geq 14$ then $\mathcal{D}$ requires fewer than two checks more than any algorithm.

The organisation of this section is as follows. We shall first establish a recurrence equation for the average time-complexity of $\mathcal{D}$ and from it derive an upper and a lower bound for its average time-complexity.[1]

Theorem 7.24 (Average Time Complexity of $\mathcal{D}$). *The average time complexity of $\mathcal{D}$ over $\mathbb{M}^{ab}$ is given by* $\operatorname{avg}_{\mathcal{D}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, *where* $\operatorname{avg}_{\mathcal{D}}(a, 0) = 0$, $\operatorname{avg}_{\mathcal{D}}(0, b) = 0$ *and*

$$\begin{aligned}\operatorname{avg}_{\mathcal{D}}(a, b) = {} & 2 + (b - 2)2^{1-a} + (a - 2)2^{1-b} + 2^{2-a-b} - (a - 1)2^{1-2b} \\ & + 2^{-b}\operatorname{avg}_{\mathcal{D}}(a - 1, b) + (1 - 2^{-b})\operatorname{avg}_{\mathcal{D}}(a - 1, b - 1)\end{aligned}$$

if $a \neq 0$ *and* $b \neq 0$.

Proof. Let $\operatorname{tot}_{\mathcal{D}}(a, b) = \sum_{M \in \mathbb{M}^{ab}} \operatorname{checks}_{\mathcal{D}}(M)$. We shall first show how to obtain $\operatorname{avg}_{\mathcal{D}}(\cdot, \cdot)$ from $\operatorname{tot}_{\mathcal{D}}(\cdot, \cdot)$. Next we shall show how to obtain $\operatorname{tot}_{\mathcal{D}}(\cdot, \cdot)$.

The function $\operatorname{tot}_{\mathcal{D}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ is given by $\operatorname{tot}_{\mathcal{D}}(a, 0) = 0$, $\operatorname{tot}_{\mathcal{D}}(0, b) = 0$ and by

$$\begin{aligned}\operatorname{tot}_{\mathcal{D}}(a, b) &= (2^{b+1} - 2)2^{(a-1)b} + 2^{(a-1)(b-1)}((b - 2)2^b + 2) + (a - 1)(2^b - 1)2^{(a-2)b+1} \\ &\quad + \operatorname{tot}_{\mathcal{D}}(a - 1, b) + 2^{a-1}(2^b - 1)\operatorname{tot}_{\mathcal{D}}(a - 1, b - 1)\end{aligned}$$

[1] We have tried to apply a similar kind of analysis to $\mathcal{D}$ as we did to $\mathcal{L}$ but all attempts failed. As will turn out further on, the decision to use a recurrence equation has made the complexity analysis for $\mathcal{D}$ "easy." Perhaps a similar approach for $\mathcal{L}$ may result in "easy" proofs as well.

if $a > 0$ and $b > 0$. We can obtain $\operatorname{avg}_{\mathcal{D}}(s,t)$ from $\operatorname{tot}_{\mathcal{D}}(s,t)$ using the fact that $\operatorname{avg}_{\mathcal{D}}(s,t) = \operatorname{tot}_{\mathcal{D}}(s,t)/2^{st}$.

In the second and last part of the proof we have to demonstrate that the recurrence equation for $\operatorname{tot}_{\mathcal{D}}(\cdot,\cdot)$ is correct. The proof turns out to be easy when compared with the lexicographical case.

Note that $\operatorname{tot}_{\mathcal{D}}(1,b) = b2^b$ and $\operatorname{tot}_{\mathcal{D}}(a,1) = a2^a$ and for both cases, the recurrence equation is satisfied. Assume that both a and b are greater than one, then:

$$\begin{aligned}
&\operatorname{tot}_{\mathcal{D}}(a,b)\\
&= (2^{b+1}-2)2^{(a-1)b} + 2^{(a-1)(b-1)}((b-2)2^b+2) + (a-1)(2^b-1)2^{(a-2)b+1}\\
&\quad + \operatorname{tot}_{\mathcal{D}}(a-1,b) + 2^{a-1}(2^b-1)\operatorname{tot}_{\mathcal{D}}(a-1,b-1).
\end{aligned}$$

The following four points explain the purpose of each of the terms in the equation.

1. The term $(2^{b+1}-2)2^{(a-1)b}$ corresponds to the effort spent to find the smallest j such that $M_{1j} = 1$, where the effort is over all matrices in $\mathbb{M}^{ab}$. Lemma 7.16 states that the such effort spent on a row with b columns is $2^{b+1}-2$. The total number of checks is therefore given by $(2^{b+1}-2)2^{(a-1)b}$. The factor $2^{(a-1)b}$ accounts for the fact that the number of a by b matrices that have the same first row is $2^{(a-1)b}$.

2. If the first row was zero—there is one such case—then we have to compute $\operatorname{tot}_{\mathcal{D}}(a-1,b)$.

3. If the first row was non-zero—there are 2^b-1 such rows—we have to compute $\operatorname{tot}_{\mathcal{D}}(a-1,b-1)$. Let j be the smallest positive integer such that $M_{1j} = 1$. The factor 2^{a-1} accounts for the column below M_{1j} that is not checked in the recursive case.

4. "After" the recursive application the following two independent tasks have to be carried out.

   (a) For every column j for which no one could be found in the recursive case the check $M_{1j}^{?}$ has to be carried out. The following is an illustration of this case.

   $$\begin{bmatrix} * & \cdots & * & 1 & * & \cdots & * \\ & & & & 0 & & \\ & & & & \vdots & & \\ & & & & 0 & & \end{bmatrix}$$

   The term $2^{(a-1)(b-1)}((b-2)2^b+2)$ counts the number of these checks.

   (b) If the first row is non-zero and if j is the column for which it was found out that $M_{1j} = 1$, then for every row r for which no one could be found in the recursive case, it has to be checked if $M_{rj} = 1$. The following is an illustration of this case.

   $$\begin{bmatrix} 0 & \cdots & 0 & 1 & * & \cdots & * \\ & & & & & & \\ 0 & \cdots & 0 & * & 0 & \cdots & 0 \\ & & & & & & \end{bmatrix}$$

The term $(a-1)(2^b-1)2^{(a-2)b+1}$ counts these checks. The factor $(a-1)$ corresponds to the number of rows whose index is greater than one and the factor 2^b-1 is the number of non-zero rows of length b. The factor $2^{(a-2)b+1}$ is made up of the factor 2^1 which accounts for M_{rj} and the factor $2^{(a-2)b}$ which accounts for the M_{ij}, where $i \neq 1$ and $i \neq r$.

This ends the proof of Theorem 7.24. □

We have verified Theorem 7.24 as follows. For each combination of a and b such that $1 \leq a, b \leq 5$ we have computed $\sum_{M \in \mathbb{M}^{ab}} \mathrm{checks}_{\mathcal{D}}(M)$ by applying $\mathcal{D}$ to all a by b constraints and by keeping track of the total number of support-checks that were required for each combination of a and b. For each combination of a and b we have been able to verify that $\sum_{M \in \mathbb{M}^{ab}} \mathrm{checks}_{\mathcal{D}}(M)$ was exactly $\mathrm{tot}_{\mathcal{D}}(a,b)$. Our theoretical results for a by b constraints turned out to be exact, for $1 \leq a, b \leq 5$.

Theorem 7.25 (Upper Bound for Average Time-Complexity of $\mathcal{D}$). *Let a and b be positive integers such that $a+b \geq 14$. An upper bound for the average time-complexity of $\mathcal{D}$ over $\mathbb{M}^{ab}$ is given by* $\mathrm{upb}_{\mathcal{D}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, *where*

$$\begin{aligned} \mathrm{upb}_{\mathcal{D}}(a,b) &= 2\max(a,b)+2 \\ &\quad -(2\max(a,b)+\min(a,b))2^{-\min(a,b)} \\ &\quad -(2\min(a,b)+3\max(a,b))2^{-\max(a,b)}. \end{aligned} \tag{7.5}$$

Proof. We shall prove this by induction on $a+b$. Let $\mathcal{P}(i)$ be true if and only if $i \geq 14$ and for all positive integers a and b such that $i = a+b$ we have $\mathrm{avg}_{\mathcal{D}}(a,b) \leq \mathrm{upb}_{\mathcal{D}}(a,b)$.

We shall first verify the cases where $a=1$ or $b=1$ and then tackle the more general case. Let $b=1$, then

$$\begin{aligned} \mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{upb}_{\mathcal{D}}(a,b) &= \mathrm{avg}_{\mathcal{D}}(a,1) - \mathrm{upb}_{\mathcal{D}}(a,1) \\ &= a - (2a+2-(2a+1)/2-(2+3a)2^{-a}) \\ &= 2^{1-a} + 3a/2^{-a} - 3/2. \end{aligned}$$

This means that if a becomes greater than 2 then $\mathrm{avg}_{\mathcal{D}}(a,1) \leq \mathrm{upb}_{\mathcal{D}}(a,1)$. The case where $a=1$ is similar.

We have verified that for all integers $1 < a$ and $1 < b$ satisfying $14 \leq a+b \leq 15$ it is true that $\mathrm{avg}_{\mathcal{D}}(a,b) \leq \mathrm{upb}_{\mathcal{D}}(a,b)$. In other words, $\mathcal{P}(14)$ and $\mathcal{P}(15)$ are true.

Assume that $\mathcal{P}(i-2)$ and $\mathcal{P}(i-1)$ are true for some integer $i \geq 16$. We must prove that $\mathcal{P}(i)$ must hold as well. To do this we must prove that for all positive integers a and b such that $i = a+b$ we have $\mathrm{avg}_{\mathcal{D}}(a,b) \leq \mathrm{upb}_{\mathcal{D}}(a,b)$. We already know that if $a=1$ or $b=1$ then $\mathcal{P}(a+b)$ holds. Therefore, a proof for the case where $a>1$ and $b>1$ will suffice.

Let a, and b be any two integers such that $a>1$, $b>1$ and $i=a+b$. There are two cases:

either $1 < b < a$ or $1 < a \leq b$. Assume $1 < b < a$ then

$$\begin{aligned}
&\operatorname{avg}_{\mathcal{D}}(a,b) - \operatorname{upb}_{\mathcal{D}}(a,b) \\
&= 2 + 2^{2-a-b} + (b-2)2^{1-a} + (a-2)2^{1-b} - (a-1)2^{1-2b} \\
&\quad +2^{-b}\operatorname{avg}_{\mathcal{D}}(a-1,b) + (1-2^{-b})\operatorname{avg}_{\mathcal{D}}(a-1,b-1) - \operatorname{upb}_{\mathcal{D}}(a,b) \\
&\leq 2 + 2^{2-a-b} + (b-2)2^{1-a} + (a-2)2^{1-b} - (a-1)2^{1-2b} \\
&\quad +2^{-b}\operatorname{upb}_{\mathcal{D}}(a-1,b) + (1-2^{-b})\operatorname{upb}_{\mathcal{D}}(a-1,b-1) - \operatorname{upb}_{\mathcal{D}}(a,b) \\
&= (6-3a)2^{-a} + (6-3b)2^{-b} - (6-3b)2^{-2b}.
\end{aligned}$$

Since $b \geq 2$ it must hold that $(6-3b)2^{-b} \leq (6-3b)2^{-2b}$. This allows us to continue our simplification as follows:

$$\begin{aligned}
&(6-3a)2^{-a} + (6-3b)2^{-b} - (6-3b)2^{-2b} \\
&\leq (6-3a)2^{-a} + (6-3b)2^{-2b} - (6-3b)2^{-2b} \\
&= (6-3a)2^{-a} \\
&\leq 0,
\end{aligned}$$

where the last inequality follows from the fact that $a \geq 2$.

Assume that $1 < a \leq b$. We can use the same technique as for the case where $1 < b < a$ to derive the following:

$$\begin{aligned}
&\operatorname{avg}_{\mathcal{D}}(a,b) - \operatorname{upb}_{\mathcal{D}}(a,b) \\
&\leq 2 + 2^{2-a-b} + (b-2)2^{1-a} + (a-2)2^{1-b} - (a-1)2^{1-2b} \\
&\quad +2^{-b}\operatorname{upb}_{\mathcal{D}}(a-1,b) + (1-2^{-b})\operatorname{upb}_{\mathcal{D}}(a-1,b-1) - \operatorname{upb}_{\mathcal{D}}(a,b) \\
&= (6-3a)2^{-a} + (4-3b)2^{-b} + (3b-6)2^{-2b} \\
&\leq (6-3a)2^{-a} + (6-3b)2^{-b} + (3b-6)2^{-2b} \\
&\leq 0.
\end{aligned}$$

We have shown that for any integer $i \geq 16$ if $\mathcal{P}(i-2)$ and $\mathcal{P}(i-1)$ hold then $\operatorname{avg}_{\mathcal{D}}(a,b) \leq \operatorname{upb}_{\mathcal{D}}(a,b)$. In other words, for any integer $i \geq 16$ it is true that $\mathcal{P}(i-2) \wedge \mathcal{P}(i-1)$ implies $\mathcal{P}(i)$. We have verified that both $\mathcal{P}(14)$ and $\mathcal{P}(15)$ hold. Together this demonstrates that $\mathcal{P}(i)$ is true for every integer $i \geq 14$, which completes our proof. □

An important result that follows from Theorem 7.25 is that we can prove that $\mathcal{D}$ is efficient.

Theorem 7.26 (Efficiency). *Let $\mathcal{A}$ be any arc-consistency algorithm, and let $a + b \geq 14$, then*

$$\operatorname{avg}_{\mathcal{D}}(a,b) - \operatorname{avg}_{\mathcal{A}}(a,b) \leq 2 - \min(a,b)2^{-\min(a,b)} - (2\min(a,b) + 3\max(a,b))2^{-\max(a,b)}.$$

Proof. If $\operatorname{avg}_{\mathcal{D}}(a,b) < \operatorname{avg}_{\mathcal{A}}(a,b)$ then the theorem is obviously true. If $\operatorname{avg}_{\mathcal{A}}(a,b) \leq \operatorname{avg}_{\mathcal{D}}(a,b)$ then it follows from Lemma 7.18 that

$$\begin{aligned}
&\max(a,b)(2 - 2^{1-\min(a,b)}) \\
&\leq \operatorname{avg}_{\mathcal{A}}(a,b) \\
&\leq \operatorname{avg}_{\mathcal{D}}(a,b) \\
&\leq \operatorname{upb}_{\mathcal{D}}(a,b),
\end{aligned}$$

where $\mathrm{upb}_{\mathcal{D}}$ is the upper bound provided by Theorem 7.25. It then follows that

$$\begin{aligned}
&\mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{A}}(a,b) \\
&\leq \mathrm{upb}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{A}}(a,b) \\
&= 2 - \min(a,b)2^{-\min(a,b)} - (2\min(a,b) + 3\max(a,b))2^{-\max(a,b)},
\end{aligned}$$

which completes the proof. □

In other words, for every integer $i \geq 14$, for every positive integers a and b satisfying $i = a+b$, and for every arc-consistency algorithm $\mathcal{A}$ it is true that $\mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{A}}(a,b) < 2$. Note that if $\mathcal{A}$ is more efficient than $\mathcal{D}$ then $\mathrm{avg}_{\mathcal{A}}(a,b)/\mathrm{avg}_{\mathcal{D}}(a,b)$ goes to one as $a+b$ goes to infinity.

7.6 Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we shall compare the average number of support-checks required by $\mathcal{L}$ and $\mathcal{D}$. The comparison will consist of a theoretical evaluation of the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$ and of a comparison of the average number of support-checks for some special cases.

The remainder of this section is organised as follows. In Section 7.6.1 we shall compare the results obtained from the average time-complexity analysis of $\mathcal{L}$ and $\mathcal{D}$ from a theoretical point of view. In Section 7.6.2 we shall compare the results of the average time required by $\mathcal{L}$ and $\mathcal{D}$ for the problem classes $\mathbb{M}^{nn}$, for $1 \leq n \leq 20$.

7.6.1 A Theoretical Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we shall compare the results obtained in Section 7.4 and Section 7.5 from a theoretical point of view.

We already observed on Page 104 that the minimum number of support-checks required by $\mathcal{L}$ is $a+b-1$. In Section 7.5 we have derived an upper bound below $2\max(a,b)+2$ for the average number of support-checks required by $\mathcal{D}$, provided that $a+b \geq 14$. If $a+b \geq 14$ and $a = b$ then then the minimum number of support-checks required by $\mathcal{L}$ is almost the same as the average number of support-checks required by $\mathcal{D}$!

Our next observation sharpens the previous observation. It follows almost immediately from our average time-complexity analysis. It is the observation that $\mathcal{D}$ is a better algorithm than $\mathcal{L}$ because its upper bound is lower than the bound that we derived for $\mathcal{L}$ using Corollay 7.23. When a and b get large and are of the same magnitude then the difference is about $2\min(a,b)$ which is quite substantial.

We remarked that it was as if $\mathcal{L}$ carried out the checks to find its row-support on the one hand and the checks to find its column-support on the other completely independently of each other.

Our most important result is the observation that if $a+b \geq 14$ and if $\mathcal{A}$ is any arc-consistency algorithm then $\mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{A}}(a,b) < 2$. To the best of our knowledge, this is the first such result that has been obtained in the constraint literature.

7.6.2 A Comparison for Some Special Cases

In this section we shall compare the average time required by $\mathcal{L}$ and $\mathcal{D}$ for the first twenty cases where the number of rows and the number of columns are the same.

n	$\mathrm{avg}_{\mathcal{L}}$	$\mathrm{avg}_{\mathcal{D}}$	ratio	n	$\mathrm{avg}_{\mathcal{L}}$	$\mathrm{avg}_{\mathcal{D}}$	ratio
1	1.000	1.000	1.000	11	36.276	23.678	1.532
2	3.625	3.375	1.074	12	40.040	25.688	1.559
3	6.934	6.043	1.147	13	43.821	27.694	1.582
4	10.475	8.623	1.215	14	47.616	29.697	1.603
5	14.093	11.037	1.277	15	51.425	31.699	1.622
6	17.740	13.306	1.333	16	55.245	33.699	1.639
7	21.408	15.472	1.384	17	59.075	35.700	1.655
8	25.095	17.571	1.428	18	62.915	37.700	1.668
9	28.802	19.628	1.467	19	66.763	39.700	1.682
10	32.529	21.660	1.502	20	70.619	41.700	1.693

Table 7.1: Comparison of $\mathrm{avg}_{\mathcal{L}}(n,n)$ and $\mathrm{avg}_{\mathcal{D}}(n,n)$ for $n \in \{1,\ldots,20\}$.

Table 7.1 compares the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$ for each of the problem-classes $\mathbb{M}^{nn}$, where $1 \leq n \leq 20$. The columns n correspond to the class $\mathbb{M}^{nn}$. The columns $\mathrm{avg}_{\mathcal{L}}$ list the average number of support-checks required by $\mathcal{L}$. The columns $\mathrm{avg}_{\mathcal{D}}$ list the average number of support-checks required by $\mathcal{D}$. The columns ratio correspond to the ratio between $\mathrm{avg}_{\mathcal{L}}$ and $\mathrm{avg}_{\mathcal{D}}$. The data in Table 7.1 have been obtained with the use of Theorem 7.20 and Theorem 7.24.

It is important to state that the computations have *not* been carried out using floating-point numbers but with arbitrary-precision integers. This is necessary to avoid the loss of precision due to the enormous differences in the ratios between the absolute values of the numbers occurring in the formulae for the average number of support-checks required by $\mathcal{L}$ and $\mathcal{D}$.

The same data as presented in Table 7.1 are also presented in the form of a graph in Figure 7.5. The horizontal axis represents the size of the problem classes. A number n on this axis corresponds to the class of n by n matrices. The vertical axis represents the average number of support-checks required by both algorithms. The solid line in the graph represents the average number of support-checks spent by $\mathcal{L}(n,n)$. The dashed line in the graph represents the average number of support-checks spent by $\mathcal{D}(n,n)$.

The figure clearly demonstrates what was stated before as Theorem 7.24, namely that $\mathrm{avg}_{\mathcal{D}}$ is almost linear in the size of the problems. It furthermore demonstrates that already for small problem sizes $\mathcal{D}$ becomes significantly better than $\mathcal{L}$ and remains so.

In Figure 7.6 we have depicted the graph of $\mathrm{upb}_{\mathcal{D}}(n,n) - \mathrm{avg}_{\mathcal{D}}(n,n)$ for $n \in \{1,\ldots,20\}$. The position where the graph becomes positive is where $n = 7$, i.e. $\mathrm{upb}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{D}}(a,b)$ becomes positive when $a + b = 14$ which conforms with our analysis in the previous section. As the size of the problem increases the upper bound seems to remain about 0.25 above the average. This seems to suggest that it is still possible to improve upon the upper bound.

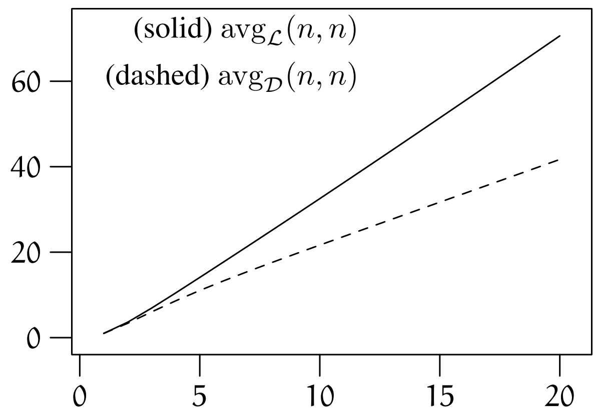

Figure 7.5: $\mathrm{avg}_{\mathcal{L}}(n,n)$ and $\mathrm{avg}_{\mathcal{D}}(n,n)$ for $n \in \{1,\ldots,20\}$.

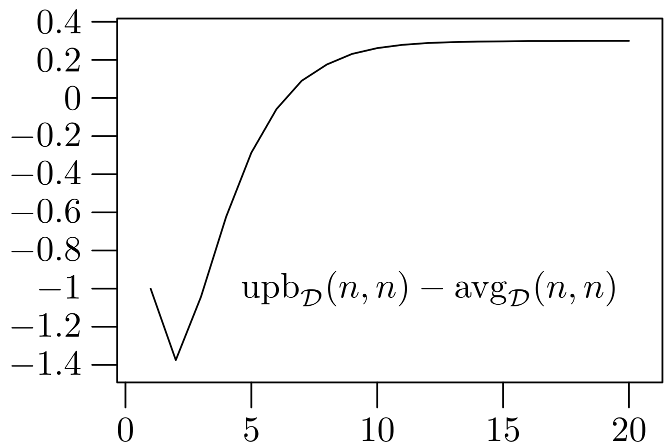

Figure 7.6: $\mathrm{upb}_{\mathcal{D}}(n,n) - \mathrm{avg}_{\mathcal{D}}(n,n)$ for $n \in \{1,\ldots,20\}$.

7.7 Conclusions and Recommendations

In this work we have studied two domain-heuristics for arc-consistency algorithms for the special case where there are two variables. We have defined two arc-consistency algorithms which differ only in the domain-heuristic they use. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic which gives preference to double-support checks. We have presented a detailed case-study of the algorithms $\mathcal{L}$ and $\mathcal{D}$ for the case where the size of the domains of the variables is two. Finally, we have carried out a careful average time-complexity analysis for $\mathcal{L}$ and $\mathcal{D}$.

We have defined the notion of a *trace* and have demonstrated the usefulness of this notion. In particular we have shown that the average savings of a trace are $(ab-l)2^{-l}$, where l is the length of the trace and a and b are the sizes of the domains of the variables.

As part of our detailed case-study we have presented three good reasons why arc-consistency algorithms should prefer double-support checks to other checks. The first reason is that a double-support check has a higher pay-off. If a double-support check is successful two things are learned in return for only one support-check as opposed to only one new thing for a successful single-support check. The second reason is that it is a necessary condition to maximise the number of successful double-support checks in order to minimise the total number of support-checks. The third and last reason is that the savings of a trace are of the form $(ab-l)2^{ab-l}$, where l is the length of the trace.

Our average time-complexity analysis has provided the bound of about $2a+2b-2\log_2(a)-0.665492$ checks for $\operatorname{avg}_{\mathcal{L}}(a,b)$ for sufficiently large a and b and an upper bound of $2\max(a,b)+2-(2\max(a,b)+\min(a,b))2^{-\min(a,b)}-(2\min(a,b)+3\max(a,b))2^{-\max(a,b)}$ for $\operatorname{avg}_{\mathcal{D}}(a,b)$, for $a+b\geq 14$.

Two results follow immediately from the lower bound for $\mathcal{L}$ and the upper bound for $\mathcal{D}$. Our first result is that it clearly shows that $\mathcal{D}$ is the better algorithm of the two. Our second result is the result that if $\mathcal{A}$ is any arc-consistency algorithm and if $a+b\geq 14$ then $\operatorname{avg}_{\mathcal{D}}(a,b)-\operatorname{avg}_{\mathcal{A}}(a,b)<2$. We have proved that $2\max(a,b)(1-2^{-\min(a,b)})$ is a lower bound for any arc-consistency algorithm. Together with our second result this allowed us to demonstrate that $\mathcal{D}$ is "optimal" in the sense that it is very close to that lower bound.

The work that was started here should be continued in the form of a refinement of our analysis for the case where only every m-th out of every n-th support-check succeeds. This will provide an indication of the usefulness of the two heuristics under consideration when they are used as part of a MAC-algorithm. Furthermore, we think that it should be worthwhile to tackle the more complicated problem of analysing the case where the constraints are not required to contain only two variables. Finally, we think that it should be interesting to implement an arc-consistency algorithm which does not repeat support-checks and which comes equipped with our double-support heuristic as its domain-heuristic.

Chapter 8

Conclusions and Recommendations

8.1 Introduction

In this thesis we have studied algorithms to solve problems occurring in the areas of Constraint Satisfaction and Gröbner Basis Theory. We have pointed out relationships between on the one hand notions in Geometry and Gröbner Basis Theory and on the other notions in Constraint Satisfaction Theory, we have studied domain-heuristics for arc-consistency algorithms, and we have generalised the chronological backtracking algorithm.

8.2 Gröbner Bases

8.2.1 Conclusion

It is a well known fact that varieties are in one-to-one correspondence with certain kinds of ideals. An important observation that we have made is that finite constraints are in one-to-one correspondence with certain kinds of varieties. Constraints are therefore in one-to-one correspondence with certain kinds of ideals.

The relationship between constraints and ideals, allows for the application of algorithms from Ideal Theory in general and Gröbner Basis Theory in particular to CSPs. To apply these algorithms we translate a CSP to a generating set of a polynomial ideal, apply the algorithm from ideal theory, and translate back to a CSP.

We have presented an algorithm to compute CSPs in directionally solved form. The general construction is as follows. Given an input constraint satisfaction problem $\mathcal{C} = (\, X, D, C\,)$ we compute a generating set of the unique radical ideal $I \subseteq k[X]$ whose variety is equal to the solution set of $\mathcal{C}$. Next, we compute the reduced Gröbner basis G of I with respect to a lexicographical term order $\prec$ and transform G to a constraint satisfaction problem $\mathcal{C}'$ which is equivalent to $\mathcal{C}$. If $G = \{\,1\,\}$ then the original CSP is unsatisfiable and we set $\mathcal{C}'$ to $(\, X, D, \left\{\, C'_{\{\,x\,\}} \,\right\}\,)$, where x is the least significant variable of X with respect to $\prec$ and $C'_{\{\,x\,\}} = \emptyset$. Otherwise, we set $\mathcal{C}'$ to the empty set and for every polynomial in G with variables $S \subseteq X$ we add a constraint C'_S to $\mathcal{C}'$

which is equal to the intersection of the Cartesian product of the domains of the variables in S and the common zeros of the polynomials in G whose variables are equal to S.

The algorithm heavily depends on properties of Gröbner bases of ideals with respect to lexicographical term orders. If $\prec$ is a lexicographical term order such that $x_i \prec x_j$ if and only if $i < j$, then a Gröbner basis G of an ideal $I \subseteq k[x_1, \ldots, x_n]$ with respect to $\prec$ contains a generating set of the elimination ideal $k[x_1, \ldots, x_m] \cap I$, for $1 \leq m \leq n$. If I is zero-dimensional then G contains a polynomial whose leading term with respect to $\prec$ is equal to $x_i^{\alpha_i}$, for $1 \leq i \leq n$ and $\alpha_i > 0$. This in its turn guarantees that all common zeros of $k[x_1, \ldots, x_{m-1}] \cap I$ can be extended to all common zeros of $k[x_1, \ldots, x_m] \cap I$, for $1 < m \leq 2$. The construction ensures that $\mathcal{C}$ and $\mathcal{C}'$ are equivalent. The properties of the Gröbner basis ensure that $\mathcal{C}'$ is in directionally solved form with respect to $\prec$.

With a minor change, the algorithm can also be used to compute CSPs which are solved with respect to all elimination orders.

8.2.2 Recommendations

The view of constraints as varieties allowed for the application of algorithms from Gröbner basis theory. The application of these algorithms is motivated by properties of elimination ideals $I \cap k[S]$ whose common zeros are the projections of the solutions of the input CSP onto the variables in S. This resulted in a better insight into the relationship between CSPs and these ideals.

It should be interesting to investigate such relationships further. In particular, it should be interesting to investigate the relationship between Gröbner bases G with respect to term orders which are not necessarily lexicographical and the varieties (constraints) of subsets of G. Perhaps, this may lead to more general consistency and search algorithms.

8.3 Arc-Consistency

8.3.1 Conclusion

We have studied existing and new arc-consistency algorithms for binary CSPs. We have—for the first time—presented experimental and theoretical results which clearly indicate that domain-heuristics can influence the performance of arc-consistency algorithms. We have developed a new domain-heuristic which has proved itself superior to other existing heuristics both in an experimental and in a theoretical setting. The domain-heuristic is a double-support heuristic. It seeks to maximise the number of values for which it can find support per support-check. We observed that it is necessary to maximise the number of values for which new support is found per check in order to minimise the total number of checks.

We have used the double-support heuristic to create a novel arc-consistency algorithm for binary CSPs. The algorithm is a cross-breed between AC-3 and DEE. It maintains the worst-case time-complexity and space-complexity of AC-3. Our experimental results suggest that despite the fact that the hybrid repeats support-checks it is more efficient than the current state-of-the-art algorithms which do not repeat support-checks. At the expense of a worse space-complexity and with minor changes the algorithm can be turned into an algorithm which remembers support-checks so as to avoid repeating them. With this improvement the algorithm should perform even better.

Arc-consistency algorithms which do not repeat support-checks all have the same worst-case time-complexity. It is only by studying the average time-complexity of such algorithms that the best such algorithm can be identified. Good arc-consistency algorithms have to be good on average.

Prompted by the success of the new double-support heuristic, we have studied its average time-complexity and that of a lexicographical domain-heuristic for the case where there are only two variables in the CSP. The main results of the time-complexity analysis are two-fold. The first result is that lexicographical heuristic is about two times less efficient on average than the double-support heuristic for sufficiently large domain sizes. A second result is that the double-support heuristic is nearly optimal for sufficiently large domain sizes; should better heuristics exist then they can only be "marginally" better. We have discussed the consequences of the decision to study 2-variable CSPs. Our informal discussion suggests that we can probably study domain-heuristics by studying 2-variable CSPs.

8.3.2 Recommendations

It should be interesting to implement an efficient arc-consistency algorithm which uses the double-support heuristic and does not repeat support-checks, and to compare it against AC-7. The average-case time-complexity analysis of the lexicographical and the double-support heuristic have provided insight into necessary properties of good arc-consistency algorithms for the case when there are two variables in the CSP. These results should be generalised for the case when there are more variables in the CSP. Many consistency algorithms use underlying lexicographical domain-heuristics. The notion of a double-support check for arc-consistency algorithms (2-consistency algorithms) may have a generalisation for k-consistency algorithms. It should be interesting to study such generalisations. If they exist, they may bring higher-order consistency algorithms into the realm of feasibility.

8.4 Generalised Backtracking

8.4.1 Conclusion

As laid out before, constraints are in essence varieties, i.e. constraints are geometrical objects which, in their turn, correspond to certain kinds of polynomial ideals. This suggests that constraints have "degrees" similar to the (total) degrees of polynomials. We have provided a definition of the degree of a set of variables in a constraint and provided an algorithm which uses this notion to partition constraints in a way which is reminiscent to factorisations of polynomials.

We have presented Proposition 5.5 which states that if C_S is a constraint, if κ is a cover of C_S, and if $\mathcal{C} = (\, X, D, C \,)$ is a CSP such that $C_S \in C$, then the solutions of $\mathcal{C}$ are equal to the union of the solutions of the members of $\{\, (X, D, \{\, c\,\} \cup (C \setminus \{\, C_S\,\}))\ :\ c \in \kappa\,\}$. Since partitions are covers, the proposition also applies to partitions.

We have shown that at the lowest level, backtracking implicitly uses the proposition for a unary constraint C_S (the domain of the current variable) and for the unique maximal partition of C_S. Partitions are called linear if their members are linear. Maximal partitions of unary constraints are linear. The (local) branching factor of chronological backtrack search is equal to the cardinality of the linear partition of a *unary* constraint (the domain of the current variable). We have generalised this notion of branching factor to that of a *generalised branching factor* (the cardinality) of a linear partition of *any* constraint.

We have used our insight into the properties of linear partitions to construct a generalisation of the chronological backtracking algorithm. It is a generalisation in the sense that it can use Proposition 5.5 for *any* kind of constraint C_S.

The use of a linear partition π_g of an arc-consistent binary constraint in generalised backtrack search is similar to the use of the linear partition π_c of a unary constraint in chronological backtrack search. In chronological backtrack search, the members of π_c are in one-to-one correspondence with the branches of the current node in the search tree and each member of π_c (each branch) allows for the elimination of the current variable. In generalised backtrack search, the members of π_g are in one-to-one correspondence with the branches of the current node in the search tree and each member of π_g (each branch in the search tree) allows for the elimination of a variable.

We have presented a function which maps a binary arc-consistent constraint $C_{\{\,x,y\,\}}$ to a linear partition π of $C_{\{\,x,y\,\}}$ such that the generalised branching factor of π does not exceed $\min(|D(x)|, |D(y)|)$ but may be smaller. With the use of this function, generalised backtracking can always obtain the same generalised backtracking factor as chronological backtracking and may obtain smaller generalised branching factors. Indeed, the application of the generalised backtracking algorithm to some large problems from the literature demonstrated that a significant reduction of the generalised branching factor can be obtained.

8.4.2 Recommendations

We have presented a few results of the application of a toy implementation of the generalised backtracking algorithm to some large problems from the literature. To investigate the usefulness of the generalised backtracking approach it is necessary to implement a version that attempts to minimise the number of support-checks. The performance of this algorithm should be compared empirically against other backtrack variants for a vast range of problems. The empirical investigation should be complemented by a theoretical evaluation of the efficiency of these algorithms. The empirical and theoretical comparisons should consist of comparisons of the average number of support-checks that are required for the different algorithms and comparisons of the average generalised branching factor.

Bibliography

[Adams and Loustaunau, 1994] W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. Graduate Studies in Mathematics. American Mathematical Society, 1994.

[Aiba and Hasegawa, 1992] A. Aiba and R. Hasegawa. Constraint logic programming system: CAL, GDCC and their constraint solvers. In Institute for New Generation Computer Technology (ICOT), editor, *Proceedings of the International Conference on Fifth Generation Computer Systems. Volume 1*, pages 113–131. IOS Press, 1992.

[Aiba *et al.*, 1988] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Proceedings International Conference on Fifth Generation Computer Systems, Tokyo, Japan, Dec 1988*, pages 263–76. Ohmsha Publishers, 1988.

[Apt, 1997] K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[Arnon and Mignotte, 1988] D.S. Arnon and M. Mignotte. On mechanical quantifier elimination for elementary algebra and geometry. *Journal of Symbolic Computation*, 5(1/2):237–260, 1988.

[Arnon, 1988] D.S. Arnon. Bibliography on algorithms in real algebra geometry. *Journal of Symbolic Computation*, 5(1&2):267–274, 1988.

[Baader and Siekmann, 1993] F. Baader and J.H. Siekmann. *Unification Theory*, pages 41–125. Oxford University Press, 1993.

[Becker and Weispfenning, 1993] T. Becker and V. Weispfenning. *Gröbner Bases* A Computational Approach to Commutative Algebra. Graduate Texts in Mathematics. Springer, 1993.

[Benhamou and Granvilliers, 1996] F. Benhamou and L. Granvilliers. Combining local consistency, symbolic rewriting and interval methods. In *Proceedings of the third International Conference on Artificial Intelligence and Symbolic Mathematical Computation (AISMC'96)*, Steyr, Austria, 1996. Springer-Verlag.

[Benhamou *et al.*, 1994] F. Benhamou, D. McAllister, and P. van Hentenryck. CLP(Intervals) revisited. In *International Symposium on Logic Programming*, pages 124–138. MIT Press, 1994.

[Benhamou, 1995] F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 1–21. Springer-Verlag, 1995.

[Bessière *et al.*, 1995] C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA.

[Bessière *et al.*, 1999] C. Bessière, E.G. Freuder, and J.-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[Boege *et al.*, 1986] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Groebner bases. *Journal of Symbolic Computation*, 2(1):83–98, 1986.

[Bowen and Bahler, 1991] J.A. Bowen and D. Bahler. Free logic in constraint processing. Technical report, Department of Computer Science, North Carolina State University, 1991.

[Bratko, 1986] I. Bratko. *PROLOG Programming for Artificial Intelligence*. Addison Wesley, 1986.

[Buchberger and Hong, 1991] B. Buchberger and H. Hong. Speeding-up quantifier elimination by Gröbner bases. Technical Report 91-06, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.

[Buchberger, 1985] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In R. Bose, editor, *Multidimensional Systems Theory*, Mathematics and Its Applications, chapter 6, pages 184–232. D. Reidel Publishing Company, 1985.

[Buchberger, 1987] B. Buchberger. History and basic features of the critical-pair/completion procedure. *Journal of Symbolic Computation*, 3(1&2):3–38, 1987.

[CELAR, 1994] CELAR. Radio link frequency assignment problem benchmark, `ftp://ftp.cs.city.ac.uk/pub/constraints/csp-benchmarks/celar`, 1994.

[Collins and Hong, 1991] G.E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991.

[Colmerauer, 1984] A. Colmerauer. Equations and inequations on finite and infinite trees. In Institute for New Generation Computer Technology, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[Colmerauer, 1987] A. Colmerauer. Introduction to Prolog III. In Commission of the European Communities, Directorate-General Telecommunications, Information Industries, and Innovations, editors, *Proceedings of the Fourth Annual ESPRIT Conference*, pages 611–629, 1987.

[Colmerauer, 1990] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[Cox *et al.*, 1996] D. Cox, J. Little, and J. O'Shea. *Ideals, Varieties, and Algorithms* An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer, 1996.

[Cox *et al.*, 1997] D. Cox, J. Little, and J. O'Shea. *Using Algebraic Geometry*. Springer, 1997. preprint.

[Czapor, 1989] S.R. Czapor. Solving algebraic equations: Combining Buchberger's algorithm with multivariate factorization. *Journal of Symbolic Computation*, 7(1):49–54, 1989.

[Dantzig, 1963] G.B. Dantzig. *Linear Programming and Extensions*. Princeton, New-Jersey, 1963.

[David, 1995] P. David. Using pivot consistency to decompose and solve functional csps. *Journal of Artificial Intelligence Research*, 2:447–474, May 1995. AI Access Foundation and Morgan Kaufmann Publishers.

[Dechter and Dechter, 1987] A. Dechter and R. Dechter. Removing redundancies in constraint networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)*, pages 105–109, 1987.

[Dechter and Frost, 1999] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems. Technical report, University of California, Irvine, 1999.

[Dechter and Meiri, 1989] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 271–277. Morgan Kaufmann, 1989.

[Dechter and Pearl, 1988a] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1988.

[Dechter and Pearl, 1988b] R. Dechter and J. Pearl. Tree-clustering schemes for constraint-processing. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88)*, pages 150–154, Saint Paul, Minnesota, USA, 1988. AAAI Press/MIT Press.

[Dechter and van Beek, 1995] R. Dechter and P. van Beek. Local and global relational consistency—summary of recent results. In U. Montanari and F. Rossi, editors, *PPCP*, number 976 in Lecture Notes in Computer Science, pages 240–257. Springer, 1995.

[Dechter and van Beek, 1997] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.

[Dechter, 1990a] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[Dechter, 1990b] R. Dechter. From local to global consistency. In P.F. Patel-Schneider, editor, *Proceedings of the Eight Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 231–237, 1990.

[Dechter, 1992] R. Dechter. Constraint networks; A survey. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276–285. Wiley, 1992.

[Dolzmann and Sturm, 1995] A. Dolzmann and T. Sturm. Simplification of quantifier-free formulas over ordered fields. Technical Report MIP-9517, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Dolzmann and Sturm, 1997a] A. Dolzmann and T. Sturm. Guarded expressions in practice. Technical Report MIP-9702, Fakultät für Mathematik und Informatik, Universität Passau, 1997.

[Dolzmann and Sturm, 1997b] A. Dolzmann and T. Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, 1997.

[Dolzmann *et al.*, 1996] A. Dolzmann, T. Sturm, and V. Weispfenning. A new approach for automatic theorem proving in real geometry. Technical Report MIP-9611, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[Flajolet and Sedgewick, 1996] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Mellin transform asymptotics. Technical Report Research Report 2956, INRIA, 1996.

[Freuder, 1978] E.G. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[Freuder, 1982] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[Freuder, 1985] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[Freuder, 1993] E.G. Freuder. Exploiting structure in constraint satisfaction problems. In B. Mayoh, editor, *Proceedings NATO ASI on Constraint Programing*, NATO Advanced Science Instirute Series, pages 51–73. Springer Verlag, 1993.

[Früwirth and Abdonnaher, 1997] T. Früwirth and S. Abdonnaher. *Constraint Programming* Grundlagen und Anwendungen. Springer, 1997.

[Gaschnig, 1978] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceeding of the Second Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.

[Ginsberg and McAllester, 1994] M.L. Ginsberg and D.A. McAllester. GSAT and dynamic backtracking. In A. Borning, editor, *Principles and Practice of Constraint Programming*, number 874 in Lecture Notes in Computer Science, pages 243–265. Springer-Verlag, Berlin/Heidelberg, 1994.

[Ginsberg, 1993] M.L. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Giovini *et al.*, 1991] A. Giovini, T. Mora, G. Niesi, L. Robbiano, and C. Traverso. "one sugar cube, please", or selection strategies in the Buchberger algorithm. In S. Watt, editor, *Proceedings ISSAC'91*, pages 49–54. ACM Press, New York, 1991.

[Golomb and Baumert, 1965] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, 1965.

[Gräbe, 1994] H.-G. Gräbe. On factorized Gröbner bases. Technical Report 6, Institut für Informatik, Universität Leipzig, Germany, 1994.

[Haralick and Elliott, 1980] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[Heintze *et al.*, 1992] N.C. Heintze, J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLPR programmer's manual version 1.2, 1992.

[Hong and Ratschan, 1995] H. Hong and S. Ratschan. RISC-CLP(Tree(Δ)) a constraint logig programming system with parametric domain. Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1995.

[Hong and Văsaru, 1996] H. Hong and D. Văsaru. Survey on nonlinear optimization. Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1996.

[Hong, 1991] H. Hong. Comparison of several decision algorithm for the existentional theory of the reals, 1991.

[Hong, 1992] H. Hong. Non-linear constraints solving over real numbers in constraint logic programming (introducing RISC-CLP). Technical Report 92-08, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[Hower, 1995] W. Hower. Constraint satisfaction — algorithms and complexity analysis. *Information Processing Letters*, 55(3):171–178, 1995.

[Jaakola, 1990] J. Jaakola. Modifying the simplex algorithm to a constraint solver. In P. Deransart and J. Maluszynski, editors, *Proceedings of the Second International Workshop on Programming Language Implemenation and Logic Programming*, pages 89–105, 1990.

[Jaffar and Lassez, 1986] J. Jaffar and J-L. Lassez. Constraint logic programming. Technical Report 74, Monash University, Department of Computer Science, Clayton, Victoria, Australia, 1986.

[Jaffar and Lassez, 1987a] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[Jaffar and Lassez, 1987b] J. Jaffar and J-L. Lassez. From unification to constraints. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the Sixth Conference on Logic Programming*, pages 1–18, 1987.

[Jaffar and Maher, 1994] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19 & 20:503–582, 1994.

[Jaffar *et al.*, 1993] J. Jaffar, M.J. Maher, P.J. Stuckey, and R.H.C. Yap. Projecting CLPR constraints. *New Generation Computing*, 11(3,4):449–469, 1993.

[Kapur, 1986] D. Kapur. Using Groebner bases to reason about geometry problems. *Journal of Symbolic Computation*, 2(4):399–408, 1986.

[Knuth and Bendix, 1970] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, U. K., 1970.

[Kondrak and van Beek, 1995] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 541–547, 1995.

[Kondrak and van Beek, 1997] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking methods. *Artificial Intelligence*, 89:365–387, 1997.

[Kutzler and Stifter, 1986] B. Kutzler and S. Stifter. On the application of Buchberger's algorithm to automated geometry theorem proving. *Journal of Symbolic Computation*, 2(4):389–397, 1986.

[Lassez *et al.*, 1988] J-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, number 306 in LNCS, pages 67–113. 1988.

[Lassez *et al.*, 1989] J-L. Lassez, K. McAloon, and T. Huynh. Simplification and elimination of redundant linear arithmetic constraints. In *Proceedings of the North American Conference on Logic Programming*, page 37ff., Cleveland, USA, 1989.

[Mackworth and Freuder, 1985] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–73, 1985.

[Mackworth and Freuder, 1993] A.K. Mackworth and E.C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59:57–62, 1993.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Marriot and Stuckey, 1998] K. Marriot and P.J. Stuckey. *Programming with Constraints,* An Introduction. MIT Press, 1998.

[Melenk, 1990] H. Melenk. Solving polynomial equation systems by Groebner methods. CWI Quarterly 3, 1990.

[Melenk, 1993] H. Melenk. Algebraic soltion of nonlinear equation systems in REDUCE. Technical report, Conrad-Zuse-Zentrum für Informationstechnik, Berlin, Germany, 1993.

[Mishra, 1993] B. Mishra. *Algorithmic Algebra*. Springer-Verlag, 1993.

[Mohr and Henderson, 1986] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Montanari and Rossi, 1988] U. Montanari and F. Rossi. Fundamental properties of networks of constraints: A new formulation. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, Springer Series in Symbolic Computation, pages 426–449, 1988.

[Montanari, 1974] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[Mora and Robbiano, 1988] T. Mora and L. Robbiano. The Gröbner fan of an ideal. *Journal of Symbolic Computation*, 6(2-3):183–208, 1988.

[Nadel, 1987] B.E. Nadel. The complexity of backtracking and forward checking: Search-order and instance specific results. Technical Report 88-002, Computer Science Dept., Wayne State University, Detroit, Michigan, U.S.A. , 1987.

[Nadel, 1989] B.E. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5(4):188–224, 1989.

[Nilsson and Maluszynski, 1989] U. Nilsson and J. Maluszynski. *Logic, Programming and PROLOG*. Wiley & Sons, 1989.

[Older and Benhamou, 1993] W. Older and B. Benhamou. Programming in CLP(BNR). In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 239–249, 1993.

[Pesch, 1996] M. Pesch. Factorizing Gröbner bases, 1996.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Renegar, 1992a] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part I: Introduction. Preliminaries. The geometry of semi-algebraic sets. The decision problem for the existential theory of the reals. *Journal of Symbolic Computation*, 13(3):255–299, 1992.

[Renegar, 1992b] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part II: The general decision problem. Preliminaries for quantifier elimination. *Journal of Symbolic Computation*, 13(3):301–327, 1992.

[Renegar, 1992c] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part III: Quantifier elimination. *Journal of Symbolic Computation*, 13(3):329–352, 1992.

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley & Sons, 1994.

[Sabin and Freuder, 1997] D. Sabin and E.C. Freuder. Understanding and improving the MAC algorithm. In G. Smolka, editor, *Principles and Practice of Constraint Programming*, pages 167–181. Springer Verlag, 1997.

[Sakai and Aiba, 1989] K. Sakai and A. Aiba. CAL: A theoretical background of CLP and its applications. *Journal of Symbolic Computation*, 8(6):589–603, 1989.

[Sakai and Sato, 1990] K. Sakai and Y. Sato. Application of ideal theory to boolean constraint solving. In *Proceedings Pacific Rim International Conference on Artificial Intelligence*, 1990.

[Schrijver, 1996] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1996.

[Siekmann, 1989] J.H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3&4):207–274, 1989.

[Smith, 1995] B.M. Smith. A tutorial on constraint programming. Technical Report 95.14, School of Computer Studies, University of Strathclyde, 1995.

[Sturm and Weispfenning, 1996] T. Sturm and V. Weispfenning. Computational geometry problems in redlog. Technical Report MIP-9708, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[Tarski, 1951] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Second edition revisisted, first edition: The RAND Corporation (1948).

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[Van Hentenryck and Ramachandran, 1994] P. Van Hentenryck and V. Ramachandran. Backtracking without trailing in CLPR(Lin). In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 349–360, 1994.

[Van Hentenryck *et al.*, 1992] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.

[Wallace and Freuder, 1992] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI '92*, pages 163–169, Vancouver, British Columbia, Canada, 1992.

[Wallace, 1993] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In R. Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 239–245, 1993.

[Weispfenning, 1993] V. Weispfenning. A new approach to quantifier elimination for real algebra. Technical Report MIP-9305, Fakultät für Mathematik und Informatik, Universität Passau, 1993.

[Weispfenning, 1994] V. Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, Fakultät für Mathematik und Informatik, Universität Passau, 1994.

[Weispfenning, 1996] V. Weispfenning. Applying quantifier elimination to problems in simulation and optimization. Technical Report MIP-9607, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[Weispfenning, 1997] V. Weispfenning. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189–208, 1997.

[Winkler, 1984] F. Winkler. *The Church-Rosser Property in Computer Algebra and Special Theorem Proving: An Investigation of Critical-Pair/Completion Algorithms*. PhD thesis, Johannes Kepler University, Linz, Austria, 1984.

List of Symbols

κ	cover, page 62
$K(S)$	covers of S, page 62
$\deg(C_T, S)$	degree of S in C_T, page 67
d	maximum domain size, page 10
$\mathrm{I}_X(I)$	elimination ideal of I with respect to X, page 29
$\gcd(p, q)$	greatest common divisor of p and q, page 32
$\mathrm{lt}_\prec(p)$	leading term of p with respect to $\prec$, page 39
S_x	x-layer of some constraint C_S, page 71
$\mathrm{lc}_\prec(p)$	leading coefficient of p with respect to $\prec$, page 39
$\mathrm{lm}_\prec(p)$	leading monomial of p with respect to $\prec$, page 39
$\mathrm{nf}_\prec(G, f)$	normal form of f with respect to G and $\prec$, page 39
π	partition, page 62
$\Pi(S)$	partitions of S, page 62
2^S	power set of S, page 62
$\mathrm{proj}_z(t)$	projection of t onto z, page 71
I_R	R-module of I, page 25
R_{xy}	directed relation between x and y, page 84
$\mathbb{T}_X$	power products of variables in X, page 38
$\mathrm{terms}(f)$	terms of f, page 39
$(\, X, D, C \,)$	constraint satisfaction problem, page 9

$A \times B$	Cartesian product of A and B, page 9
C	set of constraints, page 9
C_S	constraint, page 8
$D(x)$	domain of x, page 8
e	number of edges in constraint graph, page 10
R	set containing all directed relations, page 84
$s \prec t$	s precedes t with respect to term order $\prec$, page 38
$S \subset T$	proper set containment of S in T, page 25
$S \subseteq T$	set containment of S in T, page 25
$t \in C_S$	consistency-check, page 9
$u \mid v$	u divides v, page 39
$u \nmid v$	u does not divide v, page 39
V_S	variety of C_S, page 50
$x <_{\text{lex}} y$	x precedes y with respect to lexicographical variable ordering $<_{\text{lex}}$, page 8
X	set of variables, page 8
x_i	variable, page 8

List of Acronyms

AC Arc Consistency.

BNR Bell Northern Research.

CAD Cylindrical Algebraic Decomposition.

CAL Contrainte Avec Logique.

CLP Constraint Logic Programming.

CSP Constraint Satisfaction Problem.

DEE Domain Element Elimination.

DEEB Domain Element Elimination with Backtracking.

FOTR First-Order Theory of the Reals.

MAC Maintain Arc-Consistency.

RISC Research Institute for Symbolic Computation.

RLFAP Radio Link Frequency Assignment Problem.

Index