Constructing Algorithms for Constraint Satisfaction and Related Problems

Methods and Applications

by

Ola Angelsmark
Constructing Algorithms for Constraint Satisfaction and Related Problems

Methods and Applications

Ola Angelsmark
Abstract

In this thesis, we will discuss the construction of algorithms for solving Constraint Satisfaction Problems (CSPs), and describe two new ways of approaching them. Both approaches are based on the idea that it is sometimes faster to solve a large number of restricted problems than a single, large, problem. One of the strong points of these methods is that the intuition behind them is fairly simple, which is a definite advantage over many techniques currently in use.

The first method, the covering method, can be described as follows: We want to solve a CSP with $n$ variables, each having a domain with $d$ elements. We have access to an algorithm which solves problems where the domain has size $e < d$, and we want to cover the original problem using a number of restricted instances, in such a way that the solution set is preserved. There are two ways of doing this, depending on the amount of work we are willing to invest; either we construct an explicit covering and end up with a deterministic algorithm for the problem, or we use a probabilistic reasoning and end up with a probabilistic algorithm.

The second method, called the partitioning method, relaxes the demand on the underlying algorithm. Instead of having a single algorithm for solving problems with domain less than $d$, we allow an arbitrary number of them, each solving the problem for a different domain size. Thus by splitting, or partitioning, the domain of the large problem, we again solve a large number of smaller problems before arriving at a solution.

Armed with these new techniques, we study a number of different problems; the decision problems $(d, l)$-CSP and $k$-Colourability, together with their counting counterparts, as well as the optimisation problems Max Ind CSP, Max Value CSP, Max CSP, and Max Hamming CSP. Among the results, we find a very fast, polynomial space algorithm for determining $k$-colourability of graphs.
Acknowledgements

Knowledge of the enemy’s dispositions can only be obtained from other men.

We shall be unable to turn natural advantage to account unless we make use of local guides.

Sun Tzu, The Art of War

Even though there is only one name on the cover of this thesis, it is not the work of a single person. Without the help, encouragement, and support of family, friends, and co-workers, it would never have seen the light of day.

First of all, I have to thank Peter Jonsson, my supervisor. It is said that the teacher learns as he teaches, and if this is the case, then Peter has to be one of the most well-educated persons in the world. None of this would have been possible if I had not been able to draw upon his knowledge and experience; always generous with ideas, and never hesitant to offer critique.

During my work, I had the luxury of working closely with a mathematician, something I suspect many computer scientists, even scientists in general, could benefit greatly from, thus I offer thanks to my main co-author, Johan Thapper.

Johan Wästlund read an early draft of this thesis and, in his own words, “read it with great interest and learned a lot from it.” (Encouraging words to hear when you are writing a thesis, even if I was not entirely convinced of your sincerity, Johan…) His comments greatly increased the quality of my work. I would also like to thank Ulf Nilsson and Svante Linusson, my co-advisors, for reading and commenting on one of my drafts.

In the summer of 2002, I attended a course on specification-based testing given by Jeff Offutt. At the time, I had serious doubts about
my choice of career, and I was not at all convinced I would ever finish. Though he probably has no memory of it, Jeff set me straight by referring me to an untapped fountain of wisdom. Jeff, qatho’!

My financiers, of course, also deserves mention. My work has been financially supported by the National Graduate School in Computer Science (CUGS), and by the Swedish Research Council (VR), under grant 621-2002-4126.

Finally, without the understanding and support from my sister, whose optimism is refreshingly contagious, my parents, who suggested I get an education rather than taking a job at the mill, and my wife, who believed in me when I did not, I would not be writing this.

Linköping, Sweden, March 2005
List of papers

The general who wins a battle makes many calculations in his temple ere the battle is fought. The general who loses a battle makes but few calculations beforehand. Thus do many calculations lead to victory, and few calculations to defeat.

Sun Tzu, The Art of War

Parts of this thesis have been previously published in the following refereed papers:


Additionally, results from the following papers are also included:


# Contents

1 Prologue 1

I Introduction and Background 5

2 Introduction 7
   2.1 Constraint Satisfaction Problems 9
      2.1.1 The $n$-Queens Problem 11
   2.2 The Methods 12
      2.2.1 Techniques for Algorithm Construction 13
      2.2.2 New Methods for Algorithm Construction 22
   2.3 The Problems 28
      2.3.1 Decision Problems 29
      2.3.2 Counting Problems 33
      2.3.3 Optimisation Problems 34

3 Preliminaries 43
   3.1 Mathematical Foundations 44
   3.2 Constraint Satisfaction Problems (cont’d) 45
   3.3 Graphs 47
   3.4 CSPs as Graphs 49
   3.5 The Split and List Method 52
   3.6 Complexity Issues 55
### Methods 59

#### 4 The Covering Method 61
- 4.1 Introduction to Coverings 62
- 4.2 The Covering Theorem 66
- 4.3 Algorithms for \((d, l)\)- and \((d, 2)\)-CSPs 68
- 4.4 A Probabilistic Approach to Coverings 71

#### 5 The Partitioning Method 73
- 5.1 Introduction to Partitionings 74
- 5.2 Algorithms for \((d, l)\)- and \((d, 2)\)-CSPs 76
- 5.3 Partitioning Colouring Problems 80

### Future Work 85

#### Applications 87

#### 7 Decision Problems 89
- 7.1 \(k\)-Colouring Algorithm 89
- 7.2 Quantum and Molecular Computing 93

#### 8 Counting Problems 97
- 8.1 \#\((d, 2)\)-CSP Algorithm 98
- 8.2 \#\(k\)-Colouring Algorithm 103
- 8.3 \#3-Colouring Algorithm 105

#### 9 Optimisation Problems 109
- 9.1 Max Value CSP 110
  - 9.1.1 Max Value \((d, l)\)-CSP Algorithm 111
  - 9.1.2 Max Value \((d, 2)\)-CSP Algorithm 112
  - 9.1.3 Max Value \((d, 2)\)-CSP Approximation Algorithm 119
  - 9.1.4 Max Value \(k\)-COL Algorithm 121
- 9.2 Max CSP 125
  - 9.2.1 Max \((d, 2)\)-CSP Approximation Algorithm 126
  - 9.2.2 Max \(k\)-COL and \#Max \(k\)-COL Algorithms 129
- 9.3 Max Ind CSP 131
Chapter 1

Prologue

Imagine a lowly government clerk, sitting alone in his office. On his desk is a large pile of print-outs, showing the flow of money between accounts in different banks. It is late in the evening, and he has been working for several days on a particularly tricky case. He has been given the task of identifying “suspicious” elements in this mess of transactions and false leads, and he has no idea of how to continue. At first, it seemed simple enough; read through the material and organise it. His problem is definitely not a lack of data — in fact, it is the exact opposite.

In desperation, he draws a grid on the blackboard in his office. One dot for each of the suspicious accounts he has found.

\[
\begin{array}{cccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

With nothing else to do, he adds the transactions to the grid, drawing a line between two dots if money has been moved between the
accounts these dots represent. It looks pretty much like a mishmash of lines and dots.

He knows that there is a pattern there somewhere, if he can only find it. Intense activity between two accounts could mean something strange is going on, so on a hunch, he removes the lines which correspond to just the occasional transaction, and makes the line thicker if there has been a substantial amount of money involved.

Finally, a pattern emerges! In the center, there seems to be a group of accounts, all of which have been heavily involved in transactions with the others. Erasing the non-suspicious transactions, it is obvious that there is something strange going on here.
Clearly, it could happen that one or two accounts have a large amount of business with each other, but five of them?

As the clerk finally leaves his office, with the comforting satisfaction of a job well done, he might, as he turns off the light and locks the door, in his idle musings, wonder if what he just did could be automated somehow, and maybe even be done routinely by a computer.

It can.

Let us dwell for a moment in the deserted office, and look at the board again. Superficially, the problem seems quite easy; it is merely a matter of drawing the grid, adding the lines and then look for patterns. Now, unfortunately, it is rarely that easy, and considering the number of accounts in an average bank, and the number transactions that take place every day, the clerk has actually done an amazing job of narrowing it down to just these few. Imagine what the grid would look like if there had been a million accounts and several millions, even billions, of transactions to look at. Clearly, not even the most persistent of government clerks would have been able to find a pattern in such a, yes, mess, so handing the task over to a computer is not only a good idea, it is the only viable one.

The question is then how we get a computer to understand what it is supposed to do. Computers are still very limited in their ability to look at drawings, and reason about them, but they are good at following instructions, provided they are given in a “language” they can understand. These instructions are usually called algorithms, and a good intuition for understanding them, is to think of them as descriptions, recipes if you will, of how to solve a particular problem.
The problem the clerk was investigating, that of finding a small tightly knit group of individuals, a clique, among a larger group, is usually called the **Maximum Clique Problem**, and it is a well known problem which can be found in a number of different settings, not all of which concern fraudulent bank accounts. Usually, we dispense with the details, such as accounts, transactions, etc., and only consider the dots, calling them vertices, and the lines, which are termed edges. The structure we get is then what is commonly called a graph.

There are literally dozens, possibly hundreds, of different algorithms designed to find a maximum (i.e. the largest possible) clique in a graph to choose from, but they all share a common trait: In order to arrive at a solution, a computer will have to perform a number of calculations that is *exponential* in the number of vertices in the graph. No one knows if it is possible to do better than that, and it is in fact one of the great unsolved problems in computer science today, sometimes expressed as $P \neq NP$. Intuitively, we can view $P$ as the “easy” problems, such as sorting a collection of numbers in increasing order, or multiplying two matrices, and $NP$ as the “hard” problems. The maximum clique problem is in $NP$. In fact, it has been shown to be one of the hardest of these problems, and in this thesis we will study different ways of constructing algorithms for solving such problems.

While it would be nice to keep the discussion on this level of informality, it is an unfortunate fact that the remainder of this thesis will have to be somewhat more technical. Stephen Hawking was once warned that for every equation he included in a book, he would lose half of his readership. If this holds for theses as well, then very few people will read past Chapter 3...
Part I

Introduction

and

Background
Chapter 2

Introduction

By method and discipline are to be understood the marshaling of the army in its proper subdivisions, the graduations of rank among the officers, the maintenance of roads by which supplies may reach the army, and the control of military expenditure.

Sun Tzu, The Art of War

It is a widely held belief that \( P \neq NP \) (see e.g. Gasarch [62]). If this is the case, then no \( NP \)-hard problem can be solved in polynomial time, and one could question the wisdom of devoting time to developing exact algorithms for solving them, since such algorithms by necessity must run in superpolynomial time.

In some applications, chiefly in engineering and statistical physics, it is not always necessary, or even desirable, to find an exact solution. Sometimes a rough approximation is enough, and due to the importance of these problems, the literature abound with methods aimed at finding, and developing, such algorithms. Among the more successful, we find meta-heuristics such as Simulated Annealing [1, 86], Genetic Algorithms [67] and Tabu-Search [64], the Primal-Dual [66] and randomised rounding [101] methods for finding approximation algorithms, and the Markov Chain Monte Carlo method for approximate counting [80], just to name a few.

So why would we want exact algorithms? If it really is the case that \( P \neq NP \), and there is no polynomial time algorithm, then why
not use an efficient, reasonably accurate, approximation algorithm? First of all, several of the problems we will look at are notoriously hard to approximate. For example, the Max Value CSP and Max Ind CSP problems, which we will study in detail later, cannot be approximated within $n^{1-\varepsilon}$ for $\varepsilon > 0$, where $n$ is the number of variables in the problem, unless $P = NP$ [82, 83], so if we want to find a good solution (provided “good” means better than the rather poor approximation we can achieve in polynomial time) our best bet is an exact algorithm. However, this is neither the only, nor the most important of the many reasons to study exact algorithms for NP-hard problems. Woeginger [115] notes the following important questions believed to have caused the recent interest in superpolynomial algorithms:

- **The need for a general theory of such algorithms.**
  
  There are some isolated results in the literature, but we have not even begun a much needed systematic investigation.

- **Why is there such a wide variation in the worst case complexities among the best known exponential algorithms?**
  
  For some NP-complete problems, the best known algorithms are radically faster than for others. What are the relations between the worst case behavior of different problems?

- **Exponential time does not necessarily mean inefficient.**
  
  So far, it seems that the development of computers follow the conjecture made by Moore [98] (also known as “Moore’s Law”) rather well, thus instances which were infeasible a few of years ago can be solved almost routinely today.

Consequently, we can draw the conclusion that it is of interest to find exact algorithms, and that will be the topic of this thesis. While it would be interesting to study the methods and algorithms we propose empirically, this has not been one of our goals; rather, we have focused their theoretical properties.
Chapter Outline

We will begin by discussing the central problem of this thesis, the Constraint Satisfaction Problem, somewhat informally in Section 2.1, and give an example of a CSP, the $n$-queens problem. In Section 2.2 we first describe some of the common techniques that are used for algorithm construction today — Dynamic Programming, Preprocessing, Local Search, and Case Analysis (also known as Branch-and-Bound) — before we introduce our proposed methods. Then, in Section 2.3 we discuss the problems we have applied our method to, and compare our results with previous work.

2.1 Constraint Satisfaction Problems

We will from now on restrict the discussion to Constraint Satisfaction Problems (CSPs), for several reasons. CSPs have in the past years been identified as a central problem in a vast range of areas, including combinatorial optimisation, artificial intelligence, planning & scheduling, etc. [90]. The methods and applications we will discuss henceforth will all be related to CSPs, though they can often be translated into other settings.

In its most basic form, a CSP consists of a collection of variables, an associated collection of values, and rules, or constraints, restricting which values the variables can simultaneously assume. The goal is, of course, to give each variable a value without violating any of the constraints.

A classical example of a problem often formulated as a CSP is the $n$-queens problem; given $n$ queens, place them on an $n \times n$ squares chess board in such a way that no queen threatens (can capture) any other queen, using the standard rules of chess — i.e. no pair of queens are placed on the same row, column, or diagonal on the board. This problem has been used extensively for benchmarking constraint solvers in the past.

Another example is the problem of scheduling a collection of tasks. Given a number of tasks and a set of constraints, e.g. how many, and
2.1. Constraint Satisfaction Problems

which, of the tasks can be performed in parallel, which tasks has to precede which others, etc., determine an order in which to carry out these tasks such that none of the constraints are violated.

These two problems are both examples of finite domain constraint satisfaction problems, and well-known to be NP-complete in general [61]. It is, of course, also possible to consider CSPs with infinite domains, such as the set of integers $\mathbb{Z}$, or CSPs with continuous domains, e.g. subsets of the reals, $\mathbb{R}$. In the remainder of this thesis, however, we will only concern ourselves with finite CSPs.

One could wonder why we enforce this restriction. It is a valid question, and the choice warrants an explanation. Among the NP-hard problems there is no shortage of interesting applications. Apart from the two problems mentioned earlier, we have, for example, the register allocation problem [36, 37, 53], where the task is to assign variables to a limited number of hardware registers during the execution of a program. Since variables stored in registers can be accessed much faster than those stored elsewhere (in RAM, say), and it is usually the case that there are more variables than registers, it is desirable to find a way to allocate the variables which optimises the use of registers.

Another problem, which has recently received much attention due to the dramatic increase in the popularity of mobile communication, is the frequency assignment problem [60, 96], which (at its very simplest) aims at giving users non-conflicting frequencies to communicate at. Users that are geographically close together can, for obvious reasons, not share the same frequency, or even frequencies that are too close to each other, and have to be assigned different ranges. On the other hand, if they are sufficiently far apart, they can share.

Though these two applications are in quite different areas — compiler construction versus mobile communication — they are very closely related. Both of the problems can be viewed as different kinds of graph colouring problems, and we will actually devote a significant part of the discussion to graphs. However, it is also true that the graph colouring problem is very nicely formulated as a constraint satisfaction problem. In fact, it is the very restricted CSP in which we only allow the constraint disequality, viz. given two vertices $v$ and $w$,
the only requirement we have is that they have different colours. So by studying CSPs, we get the best of both worlds; we get results that have a wide range of interesting applications, while at the same time allowing us to draw conclusions about a much more general class of problems.

Now in order to be able to reason about CSPs, we need a formal definition, but we save that for later (Chapter 3.) Instead, we will start by giving an example of a CSP, and introduce some rather intuitive concepts we will need later on.

2.1.1 The \(n\)-Queens Problem

In order to get a “feel” for what a CSP is, let us now try to formulate the \(n\)-queens problem as one. The representation we use here is from Angelsmark et al. [6], but it is neither original, nor the only possible one.

First of all, we number the queens from 1 to \(n\), and, for simplicity, we assume that queen \(i\) will always be placed in the \(i\)th column on the board. Next, we introduce the variables \(q_1, q_2, \ldots, q_n\), which will denote the placement of the queens; the value of variable \(q_i\) is interpreted as which row queen \(i\) is placed in (given that it is restricted to column \(i\).) Thus if we index the board with \((1, 1)\) being the upper left corner, queen \(i\) will be placed at position \((i, q_i)\). Additionally, we get the domains of the variables from this: We do not want to place any queen outside of the board, thus each variable is limited to assume a value between \(1\) and \(n\), or, formally, \(q_i \in \{1, \ldots, n\}\).

From the rules of chess, we know that a queen will threaten any piece placed on the same row, column, or diagonal, and from this we now get the constraints. Since we have already limited the number of queens in a column to 1, the column constraints are always satisfied. If we can also make sure no pair of queens is on the same row, we will achieve the row constraints, so we impose constraints \(q_1 \neq q_2, q_1 \neq q_3, \ldots, q_{n-1} \neq q_n\), or, more compactly, for \(i, i' \in \{1, \ldots, n\}\), if \(i \neq i'\), then \(q_i \neq q_{i'}\). The diagonals are more complicated. A queen in position \((i, j)\) threatens any queen in positions \((i+1, j+1)\), \((i+1, j-1)\), \((i+2, j+2)\), etc. (See Fig. 2.1). Since we are using
2.2. The Methods

Every NP-hard problem can be solved using a naïve exhaustive search. For small instances, this is often a viable approach, and it is sometimes
used as a last step in algorithms, being applied to small, restricted subproblems. However, as a general algorithm, it has several shortcomings. First of all, exhaustively searching through the possible solutions to a problem quickly becomes prohibitively time consuming as the number of variables increase. The search space of the $n$-queens problem, for example, has $n^n$ elements. Depending on the speed of your computer, it might be feasible to solve instances with up to 8 or even 9 variables, since $8^8 \approx 16,000,000$ and $9^9 \approx 388,000,000$, but it is evident that this is going to fail miserably for larger problems. For comparison, the experimental CSP-solver NG [7] returned a solution to the 100-queens problem in just over 11 seconds on an ordinary workstation. Since the search space contains a staggering $100^{100} \approx 10^{200}$ elements, it is fairly safe to say that we did not use an exhaustive search in order to find the solution.

There are a number of techniques one can use in order to achieve an improvement over the running time of an exhaustive search algorithm, and before we introduce our methods, we will have a look at some of the standard techniques that are in use today.

2.2.1 Techniques for Algorithm Construction

Woeginger [115] contains a very nice classification of algorithms into one of four categories:

- Dynamic Programming
- Preprocessing
- Local Search
- Case Analysis

These classes constitute general techniques for constructing algorithms of NP-hard problems, and it should be pointed out that the methods we introduce in this thesis will not in any way subsume them. Rather, they should be considered as complementing existing techniques. This will become evident later, when we use algorithms developed using standard methods as subroutines in the new algorithms.
Dynamic Programming

Basically, dynamic programming works by storing intermediate, and reusable, results, thereby obviating the need to recompute them. It can at times greatly improve the running time of an algorithm, but has the unfortunate side effect of requiring an exponential amount of memory. A number of interesting algorithms based on dynamic programming can be found in the literature. Of particular note is the Chromatic Number algorithms by Eppstein [52] and Byskov [32].

Dynamic programming (where “programming” has nothing to do with computers, but refers to the tabulation used in the solution method) was developed in the 1950’s by Richard Bellman [23]. In order for the method to be applicable to a problem, it has to exhibit two traits. First of all, the problem has to have an optimal substructure, i.e. an optimal solution contains within it an optimal solution to a subproblem. If this is the case, then it is a good indication that dynamic programming is applicable to the problem. The second trait an optimisation problem must have in order for dynamic programming to be an alternative, is that the space of the subproblem has to be “small.” Intuitively, this means that a recursive algorithm for the problem will solve the same subproblem over and over during the search, rather than generating new subproblems. We say that the problem has overlapping subproblems.

A classical problem it has successfully been applied to is the Travelling Salesman Problem, or TSP, a well-known NP-complete problem [61]. The use of dynamic programming for solving it was first suggested by Bellman [24]. One common formulation of the problem is:

\[ \text{A salesman is required to visit once, and only once, each of } n \text{ cities, starting and ending his journey in the same city. Which path minimises the total distance travelled by the salesman?} \]

A more formal, but equivalent, formulation is: Find a permutation \( \pi = (1, i_2, \ldots, i_n) \) of the integers from 1 to \( n \) which minimises the cost
of a complete tour, where the cost is given by
\[ \alpha_{1,2} + \alpha_{2,i_3} + \ldots + \alpha_{i_n,1}, \]
for \( \alpha_{ab} \in \mathbb{R} \).

Obviously, the naïve algorithm would have a running time of \( O(n!) \) — since it has to examine all possible such permutations. We will now construct a much more efficient algorithm using dynamic programming. The following is largely based on the versions by Bellman [24] and Held & Karp [72], and it is still the fastest known algorithm for the travelling salesman problem. In fact, whether or not there exists an \( O(c^n) \) time algorithm, where \( c < 2 \), is an open problem for TSP. This holds even for the case when all distances are set to unit length, i.e., the HAMILTONIAN CYCLE PROBLEM.

Let \( c_1, \ldots, c_n \) denote the cities to be visited, and let \( \delta(c_i, c_j) \) denote the distance between cities \( c_i \) and \( c_j \). Since the salesman is required to visit each of the cities once, we can, without loss of generality, fix one city, \( c_1 \) say, as a starting point. For each \( C \subseteq \{c_2, \ldots, c_n\} \), where \( c_i \in C \), let \( \Delta(c_i, C) \) denote the length of the shortest path which starts in \( c_1 \), visits exactly the cities in \( C \) (except \( c_i \), of course) before ending in \( c_i \). Since there is only one path which starts in \( c_1 \) and ends in \( c_i \) without visiting any additional cities, it holds that
\[ \Delta(c_i, \{c_i\}) = \delta(c_1, c_i). \] This will form the basis of the table we will build later, since this single-step path by necessity is optimal.

An optimal path from \( c_1 \) to some other city \( c_i \), which is not single-step, has to visit some city \( c_j \) immediately before \( c_i \). Furthermore, the length of this path has to be equal to the path from \( c_1 \) to \( c_j \) plus the single-step distance from \( c_j \) to \( c_i \), and it is also the case that the path \( c_1 \) to \( c_j \) is of minimum length, since otherwise we could have found another, shorter, path to \( c_i \). Consequently, we get

\[ \Delta(c_i, C) = \min\{\delta(c_j, c_i) + \Delta(c_j, C - \{c_i\}) \mid c_j \in C - \{c_i\}\}. \]

Now given that we know \( \delta(c_1, c_i) \) for all \( i \), we can build a table of optimal distances by first calculating \( \Delta(c_i, \{c_i\}) \), \( 2 \leq i, j \leq n \), i.e. all optimal tours from \( c_1 \) to \( c_i \) which visit one other city, then \( \Delta(c_i, \{c_i, c_j, c_{j2}\}) \), \( 2 \leq i, j_1, j_2 \leq n \), (all optimal tours ending in \( c_i \) visiting 2 other cities) etc. The optimal tour can then be obtained by finding the minimum value of \( \Delta(c_i, \{c_2, c_3, \ldots, c_n\}) + \delta(c_i, c_1) \), where \( 2 \leq i \leq n \). There are \( 2^{n-1} \) subsets of the cities \( c_2, \ldots, c_n \) to consider, thus we get both a time and space complexity of \( O(2^n) \) (omitting polynomial factors.)

For example, consider the case shown in Fig. 2.2. Assuming \( \delta(i, j) = \delta(j, i) \), the distances are

\[
\begin{align*}
\delta(1, 2) &= 1, & \delta(1, 3) &= 3, & \delta(1, 4) &= 4 \\
\delta(2, 3) &= 2, & \delta(2, 4) &= 3, & \delta(3, 4) &= 1
\end{align*}
\]

and this also gives the values of \( \Delta(i, \{i\}) \).

Next, we consider the tours which visit one more city, and get

\[
\begin{align*}
\Delta(2, \{2, 3\}) &= \delta(1, 3) + \delta(3, 2) = 3 + 2 = 5 \\
\Delta(2, \{2, 4\}) &= \delta(1, 4) + \delta(4, 2) = 4 + 3 = 7 \\
\Delta(3, \{2, 3\}) &= \delta(1, 2) + \delta(3, 2) = 1 + 2 = 3 \\
\Delta(3, \{3, 4\}) &= \delta(1, 4) + \delta(4, 3) = 4 + 1 = 5 \\
\Delta(4, \{2, 4\}) &= \delta(1, 2) + \delta(2, 4) = 1 + 3 = 4 \\
\Delta(4, \{3, 4\}) &= \delta(1, 3) + \delta(3, 4) = 3 + 1 = 4
\end{align*}
\]
2. Introduction

The tours which visit two additional cities are then given by

\[
\Delta(2, \{2, 3, 4\}) = \min(\Delta(3, \{3, 4\}) + \delta(3, 2), \Delta(4, \{3, 4\}) + \delta(4, 2))
\]

\[
= \min\{7, 7\} = 7
\]

\[
\Delta(3, \{2, 3, 4\}) = \min(\Delta(2, \{2, 4\}) + \delta(2, 3), \Delta(4, \{2, 4\}) + \delta(4, 3))
\]

\[
= \min\{9, 5\} = 5
\]

\[
\Delta(4, \{2, 3, 4\}) = \min(\Delta(2, \{2, 3\}) + \delta(2, 4), \Delta(3, \{2, 3\}) + \delta(3, 4))
\]

\[
= \min\{8, 4\} = 4
\]

Thus the shortest path ending in city 2 is of length 7, for city 3 it is 5, and for city 4, it is of length 4. If we now add the final step, that of moving back to city 1, we can easily find the (an) optimal route for the salesman to travel. The two minimal length routes, shown in Fig 2.3, have length 8.

Preprocessing

In preprocessing, as the name suggests, the problem is analysed, and possibly restructured, before the actual computation begins. After this initial step, consecutive questions about the data can then be answered more quickly. The approach has recently gained in popularity, due to the success of the Max CSP algorithm by Williams [114]. Since we will use this variant of the preprocessing method in some of our later algorithms, a more in-depth example of how it works can be found in Section 3.5.

Horowitz & Sahni [76] gave a preprocessing algorithm for the binary knapsack problem, which has a running time of \(O\left(\frac{2^n}{2}\right)\). This problem is often formulated as follows: Given \(n\) objects, \(a_1, \ldots, a_n\), each having a positive (integer) weight \(w_i\), and a number \(W\), denoting the maximum weight the knapsack can hold, find a maximum subset of the items whose total weight does not exceed \(W\). Apart from Schroeppel & Shamir’s [106] lowering of the space complexity to \(O\left(\frac{2^{n/2}}{4}\right)\), this is still the best known exact algorithm for this problem. The Subset Sum Problem, which is closely related to the binary knapsack problem, is: Given a set \(S = \{a_1, a_2, \ldots, a_n\}\) of positive
2.2. The Methods

a) b)

Figure 2.3: The two optimal solutions to the problem in Fig. 2.2.

integers and an integer $k$, is there a subset of $S$ the sum of which is equal to $k$? A trivial enumeration algorithm could, by merely considering all possible subsets of $S$ arrive at a solution in $O(2^n)$ time, but this can be lowered to $O\left(2^{n/2}\right)$ by preprocessing the data beforehand. The following example is quite similar to the one found in Woeginger [115].

First we split $S$ into two sets of equal size (or as close as possible): $S_1 = \{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$ and $S_2 = \{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n\}$. Now, let $X$ be defined as the set of sums over the subsets of $S_1$, i.e.

$$X = \{x \mid x = \sum_{a \in S_1} a, \text{ where } s_1 \subseteq S_1\}$$

and, similarly, define $Y$ as

$$Y = \{y \mid y = \sum_{a \in S_2} a, \text{ where } s_2 \subseteq S_2\}.$$ 

Since $X$ contains all possible sums of elements in $S_1$, and $Y$ contains all possible sums of elements in $S_2$, it has to be the case that if we can find $x \in X$ and $y \in Y$, satisfying $x + y = k$, then we have solved the problem. (Note that, by construction, both $X$ and $Y$ contains 0 as a member, since $\emptyset$ is a subset of $S_1$ and $S_2$. ) Given that there are
2. Introduction

Algorithm 1 Local search algorithm for $k$-SAT.

$k$-SAT($\Gamma$)

1. Pick an initial assignment $\sigma$ at random, with uniform distribution.
2. repeat 3$n$ times (where $n$ is the number of variables in $\Gamma$.)
3. if $\sigma$ satisfies $\Gamma$ then
4. return $\sigma$
5. Let $c$ be some clause in $\Gamma$ not satisfied by $\sigma$.
6. Pick a literal in $c$ at random and flip its value in $\sigma$.
7. end repeat
8. return failure.

$n/2$ elements in $S_1$ and $S_2$, we can compute, and store, $X$ and $Y$ in $O(2^{n/2})$ time and $O(2^{n/2})$ space (omitting polynomial factors), and the problem has been reduced to a linear search among $2^{n/2}$ elements.

Local Search

Local search is most commonly associated with heuristic methods, such as Simulated Annealing [1, 86], Tabu Search [64], and, of course, the impressive $n$-queens algorithm by Sosič & Gu [107]. However, it has also been successfully used for exact algorithms, such as the $k$-SAT algorithm by Dantsin et al. [45].

Intuitively, a local search algorithm takes as input an assignment and then explores the “neighbourhood” of this assignment by changing the values of some of the variables, almost always with some degree of randomness involved. The particulars of the neighbourhood are highly problem specific.

One example of a very successful application of local search to $k$-SAT is the probabilistic algorithm by Schöning [105], given here as Algorithm 1. In $k$-SAT, we are given a boolean formula $\Gamma$ consisting of a conjunction of clauses, each of which is a disjunction of no more than $k$ literals. (A literal is either a variable or its negation.)
2.2. The Methods

Assume $\Gamma$, containing $n$ variables, is satisfiable, and fix a satisfying assignment (i.e. a solution) $\sigma^*$. We now have to determine the probability $p$ that the algorithm will actually find $\sigma^*$. It is often the case, and this is an example of this, that local search algorithms are conceptually easy to understand, but the actual calculations needed in order to prove their correctness can be tricky, and in this case it involves reasoning about Markov Chains. We, however, can “cheat” a bit, since we have the luxury of being able to look up the answer, and it turns out that

$$p \geq \left( \frac{1}{2} \left( 1 + \frac{1}{k-1} \right) \right)^n$$

Knowing this, it follows that we need to repeat the algorithm $1/p = (2(1 - 1/k))^n$ times in order to find the solution. As with any probabilistic algorithm, there is an error probability attached to this, but it can be arbitrarily small through iteration, and is thus negligible. Consequently, the algorithm runs in $\mathcal{O} ((2(1 - 1/k))^n)$ time, whereas the naïve algorithm which considers all possible assignments will have a running time of $\mathcal{O} (2^n)$, since there are two possible values for each variable.

There exist a number of different variations of the local search method, and this is not the only way of attacking $k$-SAT. For a survey of different approaches, see Schöning [104].

Case Analysis

Case analysis, which is also known as branch-and-bound, or branch-and-prune, is rather self-explanatory. During the search, the algorithm makes choices for the variables, and, depending on these choices, we get a number of cases to consider. Through careful analysis of these cases, we can sometimes get tighter upper bounds on the worst case, thereby reducing the number of “interesting” elements in the search space. This is probably the most commonly used of these techniques, and the literature abound with examples. We will make
2. Introduction

use of it in later chapters, in particular when discussing algorithms that exploit the microstructure of CSPs.

As an example using this approach, we will now construct an algorithm for solving the Maximum Independent Set problem. Recall that a graph $G$ consists of a set of vertices, which we denote $V(G)$, and a set of edges between them, denoted $E(G)$. The goal of the algorithm is to find a set $\alpha \subseteq V(G)$ such that no pair of vertices in $\alpha$ has an edge between them, hence the name independent. (The definitions used in this section can also be found in Section 3.3, but they are included here in order to make this section self-contained.)

We begin with an observation: If there are no vertices in the graph with more than two neighbours, then the graph will simply consist of a number of components that are either cycles or paths, and it is straightforward (i.e. polynomial) to find an independent set in such a graph—for example, in an even cycle with $2k$ vertices, there is an independent set with $k$ elements which one can find by taking every other vertex. We let $\text{MIS}_2$ denote an algorithm for solving this restricted problem, and our goal during the search is thus to try to remove vertices with more than 3 neighbours. Once all the remaining vertices have less than 3 neighbours, we are done.

Focusing on vertices with degree more than 2, we make two additional observations: For an independent set $\alpha$, and a vertex $v$, either

1. $v \notin \alpha$, or
2. $v \in \alpha$ and none of $v$’s neighbours are.

Using this observation, we get two cases to consider: If we include $v$ in $\alpha$, then we can discard all of $v$’s neighbours in that branch of the search tree, while if we do not include $v$, then we can only omit $v$ from further consideration—all of its neighbours are still valid members of $\alpha$. The resulting algorithm is given as Algorithm 2. (The set $N_G(v)$ in the algorithm denotes neighbourhood of $v$, i.e. every vertex which has an edge to $v$, and includes $v$.)

Lines 5 and 6 are, of course, where the actual work is done. The first of these takes care of the case when $v$ is in the independent set (corresponding to observation 2 above), thus removing 4 vertices from
Algorithm 2 Algorithm for finding a maximum independent set in a graph.

\[ MIS(G, \alpha) \]

1. if no vertex has degree more than 2 then
2. \textbf{return} \( MIS_2(G, \alpha) \)
3. end if

4. Let \( v \) be a vertex in \( V(G) \) with degree \( \geq 3 \).
5. \( \alpha_1 := MIS(G - N_G(v) - \{v\}, \alpha \cup \{v\}) \)
6. \( \alpha_0 := MIS(G - \{v\}, \alpha) \)
7. \textbf{return} the largest of \( \alpha_0 \) and \( \alpha_1 \).

further consideration, while the second one deals with the case when \( v \) is not in the set (observation 1), and only removes 1 vertex, \( v \) itself.

With 4 less vertices in the first branch, and 1 less in the second, we get a total time complexity of

\[ T(n) = T(n - 4) + T(n - 1) + p(n) \]

where \( p(n) \) is some polynomial in \( n \). Solving this (see Section 3.6) we find that the algorithm runs in \( \mathcal{O}(1.3803^n) \) time, where \( n \) is the number of vertices. It was shown by Moon & Moser [97] that a graph can contain at most \( 3^{n/3} \approx 1.4423^n \) independent sets, thus the algorithm \( MIS \) above really is an improvement over a naive enumeration.

Of course, a further case analysis would improve the algorithm. The latest achievement for \textsc{Maximum Independent Set} is due to Robson [102], and involves a detailed computer generated analysis, with several thousand subcases.

\subsection*{2.2.2 New Methods for Algorithm Construction}

In the forthcoming chapters, we will introduce and discuss two methods in particular which turn out to be quite successful on a number of different CSPs. One of the main strengths of the methods can be summarised in the following slogan:
Slogan: Solving an exponential number of small instances can be faster than solving a single large one.

On the surface, this might look fairly preposterous, but what it boils down to is how efficiently we can solve these small instances, and of course how many we need to solve.

For many NP-hard problems it is the case that for small domains, there exists a polynomial time algorithm for solving the problem. One such problem, which we will encounter and make use of frequently, is the graph colouring problem we mentioned earlier. Determining whether a graph can be coloured using \( k \) colours, for example, is a well-known NP-complete problem if \( k \geq 3 \), but for \( k = 2 \), there exists a straightforward polynomial time algorithm (since the graph has to be bipartite.) Similarly, there exists a polynomial time algorithm for solving \((d,2)\text{-CSPs}\) for \( d = 2 \). In this case, the problem becomes equivalent to 2-SAT, which is known to be solvable in polynomial time \([17]\). (Unfortunately, if the constraints have arity greater than 2, we no longer have this kind of “breakpoint” where the problem becomes polynomial, since \((2,l)\text{-CSP}\) is NP-complete for \( l \geq 3 \) \([61]\).)

Again, let us consider the \( n \)-queens problem for, say, \( n = 4 \). On a \( 4 \times 4 \) board, there are \( 4^4 = 256 \) possible ways to position the four queens. First, we observe that once we have a placed the queens on the board, it is straightforward (i.e. polynomial) to check if any of them threatens another. Now split each of the columns into two parts, one upper part and one lower part, each containing 2 squares, as in Fig. 2.4, and let \( U \) and \( L \) denote the upper and lower squares, respectively.

Next we create a new instance by restricting each queen to only be placed in one of the parts — viz. we further constrain the problem. For example, by adding the constraints \( q_1 \in U, q_2 \in U, q_3 \in U, q_4 \in U \), we get a problem where all 4 queens has to be placed on the upper half of the board. (This problem naturally has no solutions.) For each queen we now have two possibilities, either it is restricted to \( U \) or to \( L \). This gives us \( 2^4 \) restrictions to consider. Of course, we are still
not done, as each of the restrictions allows 2 values for each queen, still giving $2^4 \cdot 2^4 = 4^4$ possible ways to place them.

Now consider the restriction $q_1 \in U, q_2 \in U, q_3 \in L, q_4 \in U$. Figure 2.5 shows the possibilities for each queen, with the forbidden squares greyed out. For each queen $q_i$, we create a boolean variable $Q_i$ with an interpretation; $Q_i$ is true if $q_i$ is placed in the uppermost of its allowed squares, and false if it is placed in the lower one. We also translate the constraints: The fact that $q_1 \neq q_2$ (i.e. queens 1 and 2 cannot be placed in the same row) becomes $(Q_1 \lor Q_2) \land (\neg Q_1 \lor \neg Q_2)$ — which translates to either $Q_1$ is true, or $Q_2$ is true, but both cannot be true at the same time — and similarly for the pairs $q_1, q_4$ and $q_2, q_4$. The third queen does not play into this, since it cannot be on the same row as the others (it is restricted to the lower half of the board.)

The constraints which prevent diagonal threats can also be transformed into this new setting: For $Q_1, Q_2 \in L$, if $Q_1$ is true, then $Q_2$ cannot be false, since that would mean queen $q_1$ is placed in (1,1) and $q_2$ in (2,2). We also have to take into account the case when we have different restrictions; if $Q_1 \in U$ and $Q_2 \in L$, then there will never be a conflict, and thus no constraints are necessary, but if $Q_3 \in L$, we
have to ensure that it never happens that $Q_1$ and $Q_3$ have the same value (since that would imply a conflicting placement.)

Thus for each of the restrictions, we can transform the resulting instance into 2-SAT, which is solvable in linear time. The transformation is quite straightforward, and we now have $2^4 = 16$ 2-SAT instances to solve, rather than a single, large $(4,2)$-CSP.

Of course, in this small example, the work done during transformation and solving is probably not worth the effort, but for larger problems the gain is significant. For the general $n$-queens problem, there are now $(n/2)^n$ cases to consider, rather than $n^n$, and the running time of the resulting algorithm would be $O \left( p(n) \cdot \left(\frac{n}{2}\right)^n \right)$, where $p(n)$ is the (polynomial) time needed in the transformation and subsequent solving — an impressive improvement! There is a problem with this approach, however, and it was this problem which caused the development of the two methods we will describe later. What happens when $n$ is not divisible by 2? We eventually came up with two solutions to this, resulting in the two methods we will soon look at.

However, before we delve deeper into the details of the meth-
2.2. The Methods

ods, let us note that there is one additional strength to them, which should be pointed out explicitly. They are both conceptually easy to understand—an important property which is not to be underestimated. We will see later how the methods are used to derandomise the \((d, 2)\)-CSP algorithm from Eppstein [51]. Starting with a deterministic \((3, 2)\)-CSP algorithm, through careful case analysis, it is possible to transform a \((4, 2)\)-CSP into a \((3, 2)\)-CSP, thus getting a deterministic \((4, 2)\)-CSP algorithm. It is not obvious how this approach could be generalised, thus for \(d \geq 5\), a probabilistic reasoning is used in order to get an algorithm for \((d, 2)\)-CSP. In contrast, the “restrict and solve” intuition behind the two methods we present is straightforward, and easily applicable. Furthermore, as a general observation, it is usually the case that for any problem one cares to consider, it is possible to design a specialised algorithm which outperforms one that was conceived using general methods. Consequently, it is very likely that in the future, there will be algorithms which are faster than ours for some domain size. However, due to the generality of the methods we present, this will almost certainly mean that one or the other of our methods will in turn give improved bounds for problems with domains larger than this.

We have already discussed the basic idea behind the two methods, but before we move on, let us look at them in further detail, and highlight some of the differences between them.

The Covering Method

The intuition behind the covering method is the following: We have a problem with \(n\) variables, and each of these has a domain with \(d\) elements. At our disposal, we have an algorithm which can handle problems with domain size \(e < d\). Much like the previous example, we want to find a number of problems with a smaller domain size which cover the original problem—cover in the sense that a solution to one of these problems will also constitute a solution to the original one. The only way to ensure this is to make certain that no solution is lost in the transformation, i.e. for each variable, if we combine all the (restricted) domains in all the (restricted) problems, it
should be the case that we once again get the large domain. Here, we have two choices, and depending on which we choose, we get either a deterministic or a probabilistic algorithm. The idea is somewhat similar to those used by Ahlswede in an early paper from 1971, later published as a technical report [4]. Even though the work is mostly abstract, and no connection to algorithm construction is made, there are distinct similarities.

The first approach is to explicitly construct such a covering\footnote{This is not to be confused with covering codes, used in e.g. Dantsin et al. [45]}. It can be shown (see Theorem 9) that such a covering exists, and we also know the size of it, but unfortunately, the theorem only shows existence — it is not constructive. Consequently, for a given domain size we have to construct the covering from scratch, and there are several ways of doing this, e.g. table generation. This is reflected in the time complexity by a small, but present, $\varepsilon$, and though it can be chosen arbitrarily close to 0, we cannot omit it entirely.

The other approach involves less work, but, as was noted, we have to sacrifice determinism. The intuition is, again, straightforward. Rather than constructing a covering, we randomly restrict each of the variables to assume values from a domain of size $\varepsilon$, and then solve this new instance. By repeating this process a large enough number of times, we can ensure that there is a known probability of success.

The time complexity of these two approaches is the same, up to $\varepsilon$, and it is also the case that any algorithm we get from the latter can be derandomised using the first.

The Partitioning Method

In contrast, the partitioning method takes a completely different view of how to solve the problem with the 'remaining' domain elements. Rather than trying to use a single algorithm for all of the restricted problems, this method allows us to use different algorithms for different domain sizes. An example which we will study in-depth later is the following: An algorithm for solving $(4,2)$-CSPs has the running
2.3. The Problems

Time $O(1^{n_1+n_2} \cdot \alpha^{n_3} \cdot \beta^{n_4})$, where $n_i$ is the number of variables with domain size $i$. Thus for problems with domains of sizes 1 and 2, it is polynomial, for domain size 3 it runs in $\alpha^n$, and for domain size 4, it runs in $\beta^n$.

Using this algorithm, we want to solve, say, a $(7, 2)$-CSP. First, we split the domain of each variable into one part with 3 elements and one part with 4 elements. So if the original domain is $\{1, 2, 3, 4, 5, 6, 7\}$, we could, for example, use the partitioning $P_1 = \{1, 2, 3, 4\}$ and $P_2 = \{5, 6, 7\}$. Next, we consider each possible way of restricting the variables to only take values from one of these partitions. With $n$ variables in the original problem, we get $k$ variables restricted to $P_1$ and $n - k$ restricted to $P_2$, and thus get a total running time of

$$O\left(\sum_{k=0}^{n} \alpha^k \cdot \beta^{n-k}\right) = O((\alpha + \beta)^n)$$

It turns out that though this method achieves slightly worse performance than the covering method, it has some interesting properties; it is well suited for counting problems and, in particular, colouring problems. Its success on colouring problems stems from two facts: 1) the only allowed constraint is disequality, and 2) once a pair of variables is restricted to different partitions, any constraint between them is trivially satisfied. Consequently, we can consider the different partitions “in isolation,” since variables in one partition cannot constrain variables in another. This allows for extremely fast algorithms for a number of colouring problems.

2.3 The Problems

Once we have formalised and defined our two new methods for algorithm construction, it is only natural to see how, or even if, they can be applied. We will study a wide range of problems in this thesis, and, in order to get an overview of them, we will begin by presenting them somewhat informally here, together with previous and new results for these problems.
2. Introduction

Table 2.1: Time complexities of the currently best known CSP-algorithms.

<table>
<thead>
<tr>
<th>l = 2</th>
<th>d = 2</th>
<th>d = 3</th>
<th>d = 4</th>
<th>d = 5</th>
<th>d = 10</th>
<th>d ≥ 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>poly(n)</td>
<td>1.3645^n</td>
<td>1.8072^n</td>
<td>2.2590^n</td>
<td>4.5180^n</td>
<td>(dl)^n/d</td>
<td></td>
</tr>
<tr>
<td>l = 3</td>
<td>1.3302^n</td>
<td>2^n</td>
<td>2.6667^n</td>
<td>3.3334^n</td>
<td>6.6667^n</td>
<td>(d - d/\pi + \varepsilon)^n</td>
</tr>
<tr>
<td>l = 4</td>
<td>1.5^n</td>
<td>2.25^n</td>
<td>3^n</td>
<td>3.75^n</td>
<td>7.5^n</td>
<td></td>
</tr>
<tr>
<td>l = 5</td>
<td>1.6^n</td>
<td>2.4^n</td>
<td>3.2^n</td>
<td>4^n</td>
<td>8^n</td>
<td></td>
</tr>
</tbody>
</table>

2.3.1 Decision Problems

The most widely studied problem for CSPs is the decision problem, which aims at determining whether or not a CSP instance has a solution. There are numerous results for this problem to be found in the literature. Eppstein [51] gives probabilistic algorithms for \((d,2)\)-CSP, \(d > 4\), with running time \(O((0.4518d)^n)\), while for \(d \geq 11\), the probabilistic algorithm by Feder & Motwani [54] is faster. For constraints with arity greater than 2, the fastest algorithms are due to Hofmeister et al. [75] for \((2,3)\)-CSPs, with a running time of \(O(1.3302^n)\), and Schöning [105] for all other \(d\) and \(l\), which runs in \(O\left((d - d/\pi + \varepsilon)^n\right)\) time (for all \(\varepsilon > 0\)). The running times of these algorithms are summarised in Table 2.1, and with the exception of the \((2,2)\), \((3,2)\), and \((4,2)\)-CSP, the algorithms in the table are all probabilistic.

In Section 4.3, we use the covering method to construct algorithms for these problems. The running times of these algorithms can be found in Table 2.2. Note that these algorithms are all deterministic, which means that for \(d \leq 10\), it is the case that the best known deterministic algorithms have running times equal to the best non-deterministic algorithms. For \(d > 10\), the algorithm of Feder & Motwani is still faster, and when the arity of the constraints is greater than 2, the algorithms of Hofmeister et al. and Schöning are also faster than ours.

The \(k\)-colouring problem is probably one of the most intensely studied decision problems. This problem has been studied for a long time (see the Four Colour Theorem, in Section 3.3), and it was actu-
2.3. The Problems

Table 2.2: Time complexity for the deterministic, covering based, algorithms.

<table>
<thead>
<tr>
<th>l</th>
<th>d = 2</th>
<th>d = 3</th>
<th>d = 4</th>
<th>d = 5</th>
<th>d = 10</th>
<th>d ≥ 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 – 3.645n</td>
<td>1.8072n</td>
<td>2.2590n</td>
<td>4.5180n</td>
<td>(0.4518d)n</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2 – 2.2215n</td>
<td>2.9620n</td>
<td>3.7025n</td>
<td>7.4050n</td>
<td>(0.7405d)n</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.6n</td>
<td>2.4n</td>
<td>3.2n</td>
<td>4n</td>
<td>8n</td>
<td>(d – d/(l+1))n</td>
</tr>
<tr>
<td>5</td>
<td>1.6667n</td>
<td>2.5n</td>
<td>3.3334n</td>
<td>4.1667n</td>
<td>8.3334n</td>
<td></td>
</tr>
</tbody>
</table>

ally the 12th problem in the list of NP-complete problems presented by Karp [84]. The traditional approach to determine k-colourability of a graph is based on maximal independent sets — it was used as early as 1971 in an algorithm designed to find the chromatic number (i.e., the smallest k such that the graph is k-colourable) of a graph [38]. At the time of this writing, the fastest polynomial space algorithms for this problem are

- 3-Colouring in $O(1.3289^n)$ [51],
- 4-Colouring in $O(1.7504^n)$ [32],
- 5-Colouring in $O(2.1020^n)$ [34],
- 6-Colouring in $O(2.3289^n)$ [32],
- k-Colouring in $O((k/c_k)^n)$ [54] (where $c_k$ is discussed below.)

In fact, for $k > 6$, the fastest algorithm is actually the general, exponential space, algorithm for CHROMATIC NUMBER. The original version by Lawler [91] has a running time of $O\left((1 + \sqrt{3})^n\right) \in O(2.4423^n)$. This has been improved, first to $O(2.4151^n)$ by Eppstein [52], and recently to $O(2.4023^n)$ by Bykov [32].

However, if we take into account that exponential space soon becomes infeasible when we consider actual implementations of algorithms, none of these are very suited for real-world applications. Thus the probabilistic k-colouring algorithm by Feder & Motwani, which
Table 2.3: Comparison between our partitioning based $k$-colouring algorithm and that of Feder & Motwani.

<table>
<thead>
<tr>
<th>$k$</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>F &amp; M [54]</td>
<td>2.8277$n^3$</td>
<td>3.2125$n^3$</td>
<td>3.5956$n^3$</td>
<td>3.9775$n^3$</td>
<td>4.3581$n^3$</td>
</tr>
<tr>
<td>Partitionings</td>
<td>2.3290$n^2$</td>
<td>2.7505$n^2$</td>
<td>2.7505$n^2$</td>
<td>3.1021$n^2$</td>
<td>3.1021$n^2$</td>
</tr>
</tbody>
</table>

has a running time of $O((k/c_k)^n)$, is more interesting, and it is the fastest polynomial space algorithm (until now.) The running time of the algorithm is actually $O(\min(k/2, 2^{\varphi_k}))$, where $\varphi_k$ is given by the expression

$$\frac{1}{k+1} \sum_{i=0}^{k-1} \left(1 + \frac{i}{\binom{k}{2}}\right) \log_2(k-i).$$

(See also Table 2.3.) Asymptotically, $2^{\varphi_k}$ is bounded from above by $k/e$.

Now let $O(\beta^n_k)$, $k \in \{3, 4, 5\}$, be the time complexities for solving 3, 4, and 5-colouring with the algorithms in the list above. By combining these with the partitioning method, which we will do in Chapter 7, we get a polynomial space algorithm for the $k$-colouring problem which has a running time of $O(\alpha_k^n)$, where, for $k > 6$,

$$\alpha_k = \begin{cases} 
 i - 2 + \beta_5 & \text{if } 2^i < k \leq 2^i + 2^{i-2} \\
 i - 1 + \beta_3 & \text{if } 2^i + 2^{i-2} < k \leq 2^i + 2^{i-1} \\
 i - 1 + \beta_4 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} 
\end{cases}$$

for $i \geq 3$. This is a significant improvement over the $O((k/c_k)^n)$ time algorithm by Feder & Motwani, as we can see in Table 2.3, where a comparison between the algorithms is given.

Quantum and Molecular Computing

Both molecular and quantum computing are fairly new areas of research. The seeds for both fields were probably sown by Richard
2.3. The Problems

Feynman in his somewhat controversial speech “There’s Plenty of Room at the Bottom” in 1959 [56], but it would take the scientific community almost four decades to catch up with the vision; the feasibility of molecular computing was first demonstrated by Adleman [3] in 1994, when he successfully encoded a graph and solved an instance of the HAMILTONIAN PATH problem for this graph. Soon after, Lipton [92, 93] showed how to use biological experiments to solve any of the problems belonging to NP. Quantum computing was not far behind, and, in 1997, Jones & Mosca [81] achieved the first implementation of an actual quantum algorithm.

Since the field is still rather young, there is a limited amount of previous work to compare with. Cerf et al. [35] present a quantum algorithm for CSPs and it runs in \( O(d^{0.2080}n) \) time for \((d,2)\)-CSPs, but the time complexity of this algorithm is not given as an upper bound for the worst case. The analysis is made in the region where the number of constraints per variable is close to \( d^l \cdot \log d \), i.e. where the problem exhibits a phase transition. In Gent & Walsh [63], it is shown that many NP-complete problems have a different region where the problems are under-constrained and satisfiable, but can be orders of magnitude harder than those in the middle of the phase transition. Also note that the very existence of phase transitions has been questioned, cf. Achlioptas et al. [2].

Using the covering method together with results on bounded non-determinism by Beigel & Fu [22], we not only get

1. an \( O((d/3 + \varepsilon)^n \cdot 1.3803^n) \)-volume molecular computation algorithm, and

2. an \( O((d/3 + \varepsilon)^{n/2} \cdot 1.3803^{n/2}) \) time quantum algorithm

for solving \((d,2)\)-CSPs, but actually more general results regarding the existence of molecular and quantum algorithms for \((d,l)\)-CSPs. Again, the key issue is to find a “base” algorithm which we can then use together with the covering method. Our results in this area can be found in Section 7.2.
2.3.2 Counting Problems

The counting problem for CSPs (\#CSP) belongs to a complexity class known as \#P, which was introduced by Valiant [110, 111], and is defined as the class of counting problems computable in nondeterministic, polynomial time. Even if we restrict ourselves to binary CSPs, \#CSP is complete for \#P [103]. The \#CSP problem has many important applications. A broad range of classical combinatorial problems such as Graph Reliability [111] and Permanent [110] can be viewed as instances of \#CSP. This also holds for many AI problems, such as approximate reasoning [103], diagnosis [89] and belief revision [46]. Solving a CSP instance is equivalent to finding a homomorphism between graphs [77], for instance, finding a $k$-colouring of a graph $G$ is equivalent to finding a homomorphism from $G$ to a complete graph (i.e. a graph where each pair of vertices has an edge between them) with $k$ vertices. Determining the number of graph homomorphisms from one graph to another has important applications in statistical physics [50] — e.g. computations in the Potts model and the problem of counting $q$-particle Widom-Rowlinson configurations in graphs. Lately, a number of papers have addressed counting complexity issues in detail [31, 41, 94], as well as concrete algorithms. Dubois [49] presents an algorithm for counting solutions to the satisfiability problem, while Birnbaum & Lozinskii [28] proposed a modified version of the classic Davis-Putnam algorithm [47] for solving the problem. Overall, it appears satisfiability has so far been the most popular counting problem; Bayardo & Pehoushek [20] presents an algorithm for counting models to 3-SAT, and Dahllöf et al. [43, 44] contains several interesting results for \#2-SAT and \#3-SAT.

Probably the first non-trivial results for CSPs were presented by Angelsmark et al. [10], where we find an algorithm for counting solutions to binary CSPs which is based on an early version of the covering method. In [9], these results were much improved, using the partitioning method, which, as we have pointed out, is particularly well suited for counting problems. The algorithm, which we describe in Section 8.1, has a running time in $\mathcal{O}((0.6270d)^n)$, and if we compare this to the algorithms for $(d, 2)$-CSPs, the gap is not very large.
2.3. The Problems

Given the success when combining the partitioning method with the \(k\)-colouring problem, it was not surprising that it turned out equally successful when we applied it to the \(#k\)-colouring problem. Counting 2-colourings of graphs is straightforward (and polynomial), but to our knowledge, the \(#3\)-colouring algorithm we present in Section 8.3, which has a running time of \(O(1.7880^n)\), is the first non-trivial algorithm for counting 3-colourings to be found in the literature. In Section 8.2, we use these two algorithms together with the partitioning method, and the end result is an \(O(\alpha_k^n)\) time algorithm, with

\[
\alpha_k = \begin{cases} 
  i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \\
  i - 1 + \beta & \text{if } 2^i < k \leq 2^i + 2^{i-1}
\end{cases}
\]

where \(\beta = 1.7880\) comes from the running time of the \(#3\)-colouring algorithm, and \(i \geq 2\). Note the similarity to the corresponding decision problem. For large enough values of \(k\), the running times are almost identical. Again, we are only using polynomial space.

2.3.3 Optimisation Problems

When it comes to optimisation problems, it is generally the case that we are not satisfied with finding just any solution; we want to find one with certain properties.

Max Value

The first problem under consideration is the Max Value problem, where we want to find a solution which, intuitively, maximises the sum of the variable values in the solution. This problem is a generalisation of several NP-complete problems, such as Max Ones, Max DOnes, as well as the restricted variant of Integer Programming where we have 1) bounded variables, and 2) a bounded number of non-zero entries in each row of the matrix. It is NP-hard to even approximate the problem within \(O(n^{1-\varepsilon})\) for all \(\varepsilon > 0\) unless \(P = NP\) [82].

In Section 9.1, we present several algorithms for this problem: First, we introduces a covering based algorithm for the general Max
2. Introduction

Value \((d, l)\)-CSP problem. In order to be able to apply the covering method, we first construct an algorithm for a restricted problem, the MAX Value \((2, l)\)-CSP. This gives an algorithm for the general problem which has a running time of \(O((d/2 \cdot \gamma_l + \varepsilon)^n)\), \(\varepsilon > 0\), where \(\gamma_l\) is the largest, real-valued root to \(1 - \sum_{i=0}^{l} 1/x^i\) (In later chapters we will refer to \(\gamma_l = \tau(1, 2, \ldots, l)\) as the work factor.)

Next, we construct a more efficient algorithm for MAX Value \((d, 2)\)-CSP, and, again, we begin by constructing an algorithm for a restricted case. Here, we use a rather elaborate microstructure based algorithm for MAX Value \((3, 2)\)-CSP, which is then used together with the covering method to get two algorithms for MAX Value \((d, 2)\)-CSPs; one probabilistic and one deterministic, both with a running time of \(O((0.5849d)^n)\) (omitting the \(\varepsilon > 0\) for the deterministic case.)

Note, however, that the algorithm we develop for MAX Value \((3, 2)\)-CSP is not limited to MAX Value; it is, in fact, applicable to any problem where, given a CSP instance together with a function assigning weights to the variables, the goal is to maximise the sum of these weights.

We also give a split and list based \((1 - \varepsilon)\)-approximation algorithm for MAX Value \((d, 2)\)-CSP. This algorithm turns out to be quite efficient—especially for larger domains—but inherent in this preprocessing method is of course the exponential space requirement. The running time is \(O(d^{\omega n/3}/\varepsilon)\) (The constant \(\omega\) is discussed further in Section 3.5, for now it is sufficient to note that \(\omega < 2.376\).)

A further restriction of this problem, the MAX Value \(k\)-COL problem, turns out to be receptive to the partitioning method, and via the construction of a specialised \(O(1.6181^n)\) time algorithm for \(k = 3\), we get an algorithm which solves the problem for any \(k\), and has a running time in \(O(\alpha_k^n)\), where

\[
\alpha_k = \begin{cases} 
  i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\
  i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1}
\end{cases}
\]

with \(\beta_3 = 1.6180\), and \(i \geq 2\). Again, note the similarity to the \(k\)-colouring algorithm.
Max CSP

The next two problems are examples of partial constraint satisfaction problems, one of the most widely studied variant of CSPs. The word “partial” here refers to the solution, which is no longer a solution in the sense we have used it previously, but rather an assignment of values to the variables which has certain properties.

These problems arise naturally in a multitude of areas (see Freuder & Wallace [58] for a survey of the field) and can basically be divided into two main categories: The Minimal Violation Problem (MVP) and the Maximal Utility Problem (MUP) [109]. In an MVP, the goal is to find a solution which satisfies as many constraints as possible, i.e. maximises the number of satisfied constraints or, equivalently, minimises the number of violated constraints (hence the name). In contrast, in an MUP, the goal is to find a partial solution, which violates none of the constraints — partial in the sense that not all variables are assigned a value.

The literature abound with results concerning MVPs, such as approximation algorithms (i.e. finding an assignment which is guaranteed to be “close” to the optimal, i.e. within a factor times the optimal), limits on the number of clauses (in the case of Max SAT) etc. Unlike most other NP-complete problems, where a careful case analysis usually yields improved bounds on the time complexity, the MAX CSP problem, as well as MAX SAT have proven to be surprisingly difficult to solve faster than the obvious $O(d^n)$ time enumeration algorithm. Quite recently, the first algorithm which had a running time of provably less than $O(d^n)$ was found by Williams [114].

Our results in this area, discussed in Section 9.2, include a covering based approximation algorithm for MAX $(d,2)$-CSP, and as the “base case” of this, we use the deterministic, polynomial time, 0.859-approximation algorithm by Mahajan & Ramesh [95], which is a derandomisation of the algorithm by Feige & Goemans [55].

In Table 2.4 we give a comparison of the running times between our covering based 0.859-approximation algorithm and two other algorithms. Algorithm 18 is a generalisation of an approximation algorithm for MAX $k$-SAT proposed by Hirsch [73] — this generalisation
Table 2.4: Three algorithms for Max \((d, 2)\)-CSP; the first two approximate, the last one exact.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(d = 3)</th>
<th>(d = 4)</th>
<th>(d = 5)</th>
<th>(d = 10)</th>
<th>(d = 20)</th>
<th>(d = 30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 18</td>
<td>2.862(n)</td>
<td>3.895(n)</td>
<td>4.916(n)</td>
<td>9.958(n)</td>
<td>19.979(n)</td>
<td>29.986(n)</td>
</tr>
<tr>
<td>New algorithm</td>
<td>1.5(n)</td>
<td>2(n)</td>
<td>2.5(n)</td>
<td>5(n)</td>
<td>10(n)</td>
<td>15(n)</td>
</tr>
<tr>
<td>Williams [114]</td>
<td>2.388(n)</td>
<td>2.998(n)</td>
<td>3.578(n)</td>
<td>6.195(n)</td>
<td>10.726(n)</td>
<td>14.787(n)</td>
</tr>
</tbody>
</table>

was done in order to compare our algorithm to another approximation algorithm. Of course, a comparison between Algorithm 18 and ours is not strictly fair, since we are limited to an approximation ratio of 0.859 while the other algorithm works for any degree of approximation (by using even more computational resources). Clearly, Algorithm 18 is not very efficient, since even the exact algorithm by Williams [114] (which we discuss elsewhere) outperforms it — though it should be noted that the exact algorithm uses an exponential amount of space — whereas our 0.859-approximation is significantly faster for smaller domains, and is actually not overtaken by Williams’ algorithm until \(d = 29\).

The covering and partitioning methods cannot compete with the split-and-list method, at least not time complexity-wise, for the general Max CSP problem, but the situation is different if we restrict the problem to Max \(k\)-Colourable Subgraph, or Max \(k\)-COL for short. Note that while it is easy to confuse the two, this problem is quite different from the Max \(k\)-Colourable Induced Subgraph problem, which we will look at later, and is actually the unweighted case of the well-known Max \(k\)-CUT problem.

We know from Williams [114] that Max \(k\)-COL can be solved in \(O\left(k^{\omega n/3}\right)\) time, but unfortunately, the Max 2-COL and Max 3-COL algorithms based on the split-and-list approach use exponential space, and this carries over to our algorithm. Interestingly, the results hold equally well for \#Max \(k\)-COL.

Combining cases \(k = 2\) and \(k = 3\) with the partitioning method, we arrive at an algorithm for Max \(k\)-COL (and \#Max \(k\)-COL)
which has a, now familiar, running time of $O(\alpha_k^n)$, where

$$\alpha_k = \begin{cases} 
i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\
i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

with $\beta_2 = 1.7315$, $\beta_3 = 2.3872$, and $i \geq 2$. The space requirement carries over from the base algorithms and becomes $O(2^{n/3})$ for the first case, and $O(3^{n/3})$ for the second.

Max Ind CSP

Section 9.3 introduces the next problem, Max Ind CSP, which is an example of a MUP. Basically, the problem consists of finding a satisfiable subinstance of the original instance, which contains as many variables as possible. A subinstance is here a subset of the variables, together with the constraints which only involve these variables. (For example, if we have the variables $x, y$ in the subset, then the constraint $(x \lor y)$ would be included, but the constraint $(x \lor \neg y \lor z)$ would not, since $z$ is not in the subset.)

Max Ind $(d, 2)$-CSP is, in some sense, dual to the classical Max CSP in that it does not maximise the number of satisfied constraints, but instead tries to maximise the number of variables that are assigned values without violating any constraints. It is not difficult to come up with examples where Max Ind CSP is a more natural problem than Max CSP, e.g. in real-time scheduling. Say the scheduler has discovered that the tasks cannot be scheduled due to, say, resource limitations. If it were to consider the problem as a Max CSP instance, then the resulting schedule might still not be of any use, since the relaxation does not introduce more resources. If, on the other hand, it were to solve the corresponding Max Ind CSP instance, it would get a partial schedule which would be guaranteed to run — since no constraints are violated. Jonsson & Liberatore [83] contains several results regarding the complexity of this problem. In particular, it is shown that the problem is either always satisfiable (and trivial) or NP-hard.

By considering the microstructure of the CSP in question, which is basically a graph representation of the problem (defined in Sec-
2. Introduction

We can apply algorithm for Maximum Independent Set to get an algorithm for solving MAX IND \((d,2)\)-CSP, and if the MIS algorithm runs in \(O\left(c^{|V(G)|}\right)\) time, then the resulting MAX IND algorithm will have a running time of \(O\left(c^{|V(G)|}\right)\). We show, however, that we can do significantly better than this using coverings. In order to apply the covering method to the MAX VALUE problem earlier, we had to construct a specialised algorithm for the case with domain size 3, but this time we have access to algorithms for any domain size (since we can always use the MIS algorithm to construct one) and thus we have to choose which domain size we prefer. Depending on which MIS algorithm we choose, either the \(O\left(1.2025^{|V(G)|}\right)\) time polynomial space, or the \(O\left(1.1889^{|V(G)|}\right)\) time exponential space algorithm by Robson [102], we get a running time of \(O\left((0.5029d)^n\right)\) or \(O\left((0.4707d)^n\right)\), respectively. The results hold for both version of the covering method.

Applying the split-and-list method to the same problem yields an exponential space algorithm running in \(O\left((d+1)^{\omega n/3}\right)\) time (where \(\omega < 2.376\) as before.) This algorithm does not improve the results we achieved for the covering based algorithm until \(d \approx 29\) for the polynomial case, and \(d \approx 78\) for the exponential case.

We then once again restrict the problem to only containing disequality constraints, and get the MAX IND \(k\)-COL problem, and using the partitioning method we get an algorithm with running time \(O\left(\alpha_k^n\right)\), where

\[
\alpha_k = \begin{cases} 
  i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\
  i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} 
\end{cases}
\]

with \(\beta_2 = 1.4460, \beta_3 = 1.7388\), and, again, \(i \geq 2\).

Max Hamming Distance

The final problem considered in this thesis is the MAX HAMMING DISTANCE problem, which was first introduced by Crescenzi & Rossi [42] as a means of measuring ignorance, although this is far from the only application.
2.3. The Problems

Intuitively, this problem asks us to find two solutions that are as far away from each other as possible; i.e. we want to find two satisfying assignments that disagree on the values of as many variables as possible.

An interesting application of this problem can be found in interactive applications, say, at a car dealership. The user, i.e. a prospective buyer, gives the application a set of constraints, e.g. what colour the car should have, which price range he can afford, etc. By solving the corresponding Hamming distance problem, the system could then present the customer with the “extremes” of the matching cars. If one, or both, of these matches are infeasible for some reason, more constraints could be added to narrow the possibilities until a range of feasible matches is reached, and the customer can then focus on the details.

We present four algorithms for this problem in Section 9.4. The first two algorithms are needed for the case when we allow domains with more than 2 elements, or constraints with arity higher than 2. Intuitively, the algorithm for Max Hamming Distance \((d, l)-CSP\) works by first choosing which variables should assume different values in the two solutions, and then creating a new, larger instance. This new instance can be solved using existing CSP algorithms, and the solution can then be “taken apart” to get two solutions to the original problem, at a known Hamming distance. Starting with assuming that all variables are different in the two solutions, and then working downwards, the algorithm will, by trying out the different possible subsets of variables, arrive at a pair of solutions with maximum Hamming distance. Given that we can solve the \((d, l)-CSP\) in the last step in time \(O(a^n)\), the entire algorithm will have a running time of \(O(a^n(1 + a)^n)\).

The second algorithm is for the case when the domain has two elements and the constraints have arity \(l\), Max Hamming Distance \((2, l)-CSP\). Here, we note that since there are only two possible choices of values for a variable, it is unnecessary to duplicate the variables that should take different values — instead, only the constraints they are involved in are duplicated, and then any occurrence of a variable
which should assume different values in the two solutions is replaced by its negation in these constraints. The resulting algorithm will have a running time of $O((2a)^n)$, where $O(a^n)$ is the time needed to solve the $(2, l)$-CSP problem in each step.

We also give a split-and-list algorithm for Max Hamming Distance $(d, 2)$-CSP. Unlike the algorithms for Max Value CSP and Max Ind CSP, where the construction used was similar to the one used by Williams, we now have to consider two satisfying assignments. Consequently, we construct two graphs, rather than one, and add weighted edges between the graphs, where the weight of an edge denotes the distance between the assignments. Eventually, we arrive at an algorithm which has a running time of $O(d^{2an/3})$, using exponential space.

The final algorithm we present is a microstructure based algorithm for the special case when the domains have size two and we have binary constraints, Max Hamming Distance $(2, 2)$-CSP, and even in this restricted form, the problem is $NP$-complete. The algorithm exploits the microstructure by searching for a set of vertices where each vertex either does not have an edge to any other vertex — and thus can be interpreted as an assignment — or is part of a connected component with 2 or 4 vertices. Each vertex (i.e. assignment) in this set is then given a weight, and the original instance together with these weights is given to a weighted 2-SAT solver. This algorithm returns a solution with maximum weight $W$, and we can then construct a solution which differs on $W$ variables. By using the weighted 2-SAT algorithm from [44], we arrive at a running time of $O(1.7338^n)$, where $n$ is the number of variables in the problem.
2.3. The Problems
Chapter 3

Preliminaries

Whether the object be to crush an enemy, to storm a territory, or to assassinate an individual, it is always necessary to begin by finding out the names of the attendants.

Sun Tzu, The Art of War

This chapter is important for several reasons. First of all, the use of mathematical notation cannot be avoided when discussing algorithms and their computational complexity, thus we need this part in order for the ‘uninitiated’ to make sense of the following chapters. Additionally, it is an unfortunate fact that almost every research group, ours included, adopts, and sometimes invents, their own conventions, both notational and otherwise.

Chapter Outline

We begin with the mathematical foundations of the discussion in Section 3.1, before formally defining what we mean by a CONSTRAINT SATISFACTION PROBLEM in Section 3.2. Then, we introduce the graph in Section 3.3, which conveniently paves the way for the very important concept of the microstructure of a CSP, which we define next, in Section 3.4. Finally, in Section 3.5, we discuss the split-and-list method, as described by Williams [114], an interesting preprocessing
method for algorithm construction which we will make use of in later chapters.

3.1 Mathematical Foundations

We will assume the reader has some passing knowledge of mathematics in general and discrete mathematics in particular, and thus we will not dwell on the formalities. The basics are covered by most text books on discrete mathematics, e.g. by Grimaldi [69] or the somewhat more advanced book by Stanley [108]. The notation we use is fairly standard; \( \mathbb{N}, \mathbb{Z} \) and \( \mathbb{R} \) denote the natural numbers (including 0), the integers and the reals, respectively. Adding a +, e.g. \( \mathbb{R}^+ \), means they are restricted to the positive elements.

Notation and concepts which are only used within a certain section, specifically the probability theory needed in Section 9.2.1, are introduced and explained there. However, due to their frequent use, the following two theorems, the binomial theorem and its generalisation, the multinomial theorem, are given here.

**Theorem 1 (The Binomial Theorem).** If \( x \) and \( y \) are variables, and \( n \) a positive integer, then

\[
(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}.
\]

**Theorem 2 (The Multinomial Theorem).** If \( x_1, x_2, \ldots, x_k \) are variables, and \( n \) a positive integer, then

\[
(x_1 + x_2 + \ldots + x_k)^n = \sum_{n_1 + n_2 + \ldots + n_k = n} \binom{n}{n_1, n_2, \ldots, n_k} x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k},
\]

where

\[
\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{n_1! n_2! \cdots n_k!}
\]

is called the multinomial coefficient.

With that in mind, we now move on to present the central problem of this thesis.
3.2 Constraint Satisfaction Problems (cont’d)

Continuing the discussion from the introduction, we now formally define what a CSP is:

**Definition 3.** A **constraint satisfaction problem** is a **triple** \( \Theta = (X, D, C) \), where

- \( X \) is a set of variables, \( \{x_1, \ldots, x_n\} \),
- \( D \) is a (finite) set \( \{a_1, \ldots, a_{|D|}\} \) of values, and
- \( C \) is a set \( \{c_1, \ldots, c_{|C|}\} \) of constraints.

Each constraint \( c \in C \) is a pair \((s, \rho)\), where \( s \) is a list of variables of length \( m \) and \( \rho \) is an \( m \)-ary relation over the set \( D \).

In the algorithms we will look at, the actual elements in the domain(s) are uninteresting in the analysis — it is the size of the domain that is important — since we are only interested in computational complexity. A problem where all variables have a domain with at most \( d \) elements, and each constraint involves at most \( l \) variables will be denoted \((d, l)\)-CSP. Given a variable \( v \) and a set \( S \subseteq D \), we will let \((v; S)\) denote the unary constraint \( v \in S \). Another constraint which is of special interest is the binary relation **disequality**, denoted ‘\( \neq \)’, which simply states that two variables cannot assume the same value.

A **solution** to a CSP instance, sometimes called a **model**, is a function which assigns values to the variables. Of course, this cannot be done arbitrarily. The values the variables are assigned must respect the constraints of the problem, and every variable must be assigned one value. Such an assignment is said to **satisfy** the instance, and if an instance has such a satisfying assignment, we call it **satisfiable**. If the function does not assign values to every variable, but only a subset of them, it will be referred to as a **partial** solution.

Sometimes we are interested in an actual solution, and in those cases we will use the following notation: Given variables \( x_1, x_2, \ldots, x_n \) and domain values \( a_1, a_2, \ldots, a_d \), we write the solution \( f \) as the set \( \{x_1 \mapsto a_{i_1}, x_2 \mapsto a_{i_2}, \ldots, x_n \mapsto a_{i_n}\} \), where \( i_j \in \{1, \ldots, d\} \), of course.
If we want to stress that we are discussing microstructures (defined in Section 3.4), we write \( f \) as \( \{x_1[a_{i_1}], x_2[a_{i_2}], \ldots, x_n[a_{i_n}]\} \).

The following special cases will be of particular interest to us: \((d, 2)\)-CSPs, i.e. instances with binary constraints, \((2, l)\)-CSPs and \((2, 2)\)-CSPs. The latter two we will sometimes view as instances of \(l\)-Satisfiability, or \(l\)-SAT. An instance of \(l\)-SAT, or a formula, consists of the conjunction of a set of clauses, where each clause is the disjunction of (at most) \(l\) literals. (A literal is either a variable or its negation.) For example, we have the formula

\[
\Gamma := (p \lor q \lor r) \land (\neg p \lor s) \land (\neg q \lor \neg s \lor \neg t)
\]

which is an instance of 3-SAT. We can also view a formula simply as a set of clauses, rather than a conjunction, in which case we would write

\[
\Gamma := \{(p \lor q \lor r), (\neg p \lor s), (\neg q \lor \neg s \lor \neg t)\}.
\]

If we want to consider \(\Gamma\) as a \((2, 3)\)-CSP, we need relations to express the clauses; the clause \((\neg p \lor s)\) would require a binary relation \(R\) which, using a basic truth-table, we know has to be true for the values \((0, 0), (0, 1), (1, 1)\) (and false otherwise.) Consequently, with \(R = \{(0, 0), (0, 1), (1, 1)\}\), we get the constraint \(c = ((p, q), R)\). (Sometimes we will use a more convenient notation \(R(p, q)\), or even the standard infix notation \(pRq\), if the relation is binary.) The domain values in \(l\)-SAT are often named \{false, true\} or \{0, 1\}.

We will also be interested in weighted instances of 2-SAT, and we define them as follows:

**Definition 4 (Dahllöf et al. [44]).** Let \(\Gamma\) be a propositional formula and let \(L\) be the set of all literals for all variables occurring in \(\Gamma\). Given a weight vector \(w\), and a model \(M\) for \(\Gamma\), we define the weight \(W(M)\) of \(M\) as

\[
W(M) = \sum_{\{l \in L \mid l \text{ is true in } M\}} w(l)
\]

The problem of finding a maximum weighted model for \(\Gamma\) is denoted \(2\text{-SAT}_w\).
For some problems it is not sufficient to find just any solution. Among the problems we will look at later, there are several examples of optimisation problems, where the aim is to find a solution which maximises, or minimises, some property. For example, the Min ONES problem tries to find a solution to an $l$-SAT instance which minimises the number of ones (i.e. variables assigned true) in a solution. These problems are sometimes referred to as constraint satisfaction optimisation problems, or CSOPs, but we will not explicitly point this out every time we encounter one — hopefully, it should be evident from the context that it is an optimisation problem and which the underlying decision problem is. When we, somewhat carelessly, say that an optimisation problem is in $NP$, or even $NP$-complete, it is actually the corresponding decision problem that has this property.

Optimisation problems is the one case when we will consider inexact, or approximation, algorithms. These algorithms do not necessarily return an optimal solution, but they are guaranteed to return one which is ‘close’ to the optimal, where the closeness is usually expressed as a performance guarantee, or performance ratio, $\alpha$. An $\alpha$-approximation algorithm for an optimisation problem is thus an algorithm which finds a solution within $\alpha$ times the optimal.

3.3 Graphs

Graphs are possibly the most fascinating mathematical structure there is. Despite the fact that the intuition behind them is quite simple, and it is easy to find real-world examples, there is an abundance of (computationally) hard problems related to graphs. A very good example of this is the famous Four Colour Theorem which states that any map (in the intuitive, atlas related, sense of the word) can be coloured using at most four colours. The theorem dates back to 1852, when Francis Guthrie noticed that he only needed four different colours in order to colour the counties of England in such a way that no adjacent counties had the same colour. Even though it looks deceptively simple the problem withstood the efforts of some of the greatest mathematicians of all times for more than a century, and it was not proven until
3.3. Graphs

Figure 3.1: Two microstructure variables; \( x \) with domain \( \{0, 1\} \) and \( y \) with domain \( \{1, 2, 3, 4\} \).

1976 [15, 16] (this was actually one of the first, if not the first, major mathematical problem to be solved using computers.)

There exists a number of different flavors of graphs, but we will only concern ourselves with loop-free, undirected, simple graphs, or graphs for short. Again, we try to keep the discussion brief but not overly so. Should the reader like more on this subject, Berge’s [25] book contains a thorough treatment of graphs. If lighter fare is preferred, West’s [113] book is a good substitute.

A graph \( G \) consists of a set of vertices, \( V(G) \), and a set of edges \( E(G) \), where each element (edge) is an unordered pair of vertices. The size of a graph is the number of vertices, \( |V(G)| \). Usually, we will let \( n \) denote the number of vertices, but since this is also used for the number of variables in a CSP, in order to avoid confusion we will sometimes write \( |V(G)| \).

The neighbourhood of a vertex \( v \) in a graph \( G \), denoted \( N_G(v) \), or just \( N(v) \) if it is clear from the context which graph is referred to, is the set of vertices which have edges in common with \( v \), i.e. the set \( \{ u \in V(G) \mid (v, u) \in E(G) \} \). If we can get from a vertex \( v \) to a vertex \( u \) by following the edges of the graph, then we say that \( u \) is reachable from \( v \). If this is true for every pair of vertices in the graph, then the graph is connected. The equivalence classes of the “is-reachable-from” relation forms the connected components of \( G \).

If we pick a subset \( S \) of the vertices of a graph together with the edges between them (but no other edges), then we get the subgraph of \( G \) induced by \( S \), \( G(S) \). \( G(S) \) has vertex set \( S \) and edge set
\{(v, u) \mid v, u \in S, (v, u) \in E(G)\}. Somewhat sloppily, we will now and then let \(S\) denote both the vertices and the induced subgraph, if there is no room for confusion. Thus, rather than the somewhat cumbersome \(E(G(S))\), we simply write \(E(S)\) for the edge set. If \(E(S)\) is empty, i.e. there are no edges in the induced subgraph, then \(S\) forms an independent set. An independent set is maximal if it is not a proper subset of any other independent set. The problem of finding the largest maximal independent set in a graph is called the Maximum Independent Set problem. The dual problem, that of finding a maximum induced subgraph where all the vertices are pairwise connected is called Maximum Clique.

The complement of a graph \(G\) is a graph \(G^*\) with \(V(G^*) = V(G)\) and \(E(G^*) = \{(u, v) \mid u \neq v, (u, v) \notin E(G)\}\). If \(\alpha \subseteq V(G)\) is an independent set in \(G\), then it is a clique in \(G^*\), and vice versa.

### 3.4 CSPs as Graphs

Consider a binary constraint satisfaction problem \(\Theta = (X, D, C)\). If we view the variables in \(X\) as vertices, and add an edge between vertices \(x, y\) if there is a constraint \(xRy\) in \(C\), we get a structure usually referred to as the constraint graph, or, if we allow higher arity constraints, the constraint hypergraph, of \(\Theta\) \(\cite{48}\). (Hypergraphs are a generalisation of graphs where edges may contain more than 2 vertices, see e.g Berge \(\cite{25}\).)

By imposing restrictions on the structure of the constraint graph, it is possible to construct efficient polynomial time algorithms for solving the corresponding CSP. For example, if we require the graph to be free from cycles, then there is a straightforward way of finding a solution \(\cite{100}\). Consequently, constraint (hyper-)graphs have been very useful in the classification and characterisation of tractable CSPs \(\cite{68, 100}\), in particular different decomposition methods based on them.

The notion of decomposition of a CSP based on the constraint graph was first introduced by Freuder \(\cite{57}\), and has since been investigated and developed further by several authors, resulting in a
number of different methods. The idea behind it can be seen in the following example: Again, consider the constraint graph. If we can find two disconnected components, with no edges connecting them, then we can solve each of these in isolation and then combine the two solutions into a solution to the entire problem. Additionally, if there is only one, or few, edges between the components, then if we solve one of them, some information can be ‘carried over’ and make solving the other component easier.

While we will not study the constraint graph explicitly, we will make use of a related concept, that of the microstructure graph of a CSP [79]. The microstructure can be viewed as an ‘expanded’ constraint graph; usually, the constraint graph contains no information on the actual values of the variables, but in the microstructure graph we include this information.

Curiously, constructions similar to this have been suggested independently by several authors, e.g. Kozen [87], Barrow & Burstall [19] and Bačík & Mahajan [18], but despite this, it appears not much effort has been devoted to further study. We will use the following definition:

**Definition 5 (Jéou [79]).** Given a binary CSP $\Theta = (X, D, C)$, i.e. a CSP with binary constraints, the microstructure of $\Theta$ is an undirected graph $G$ defined as follows:

1. For each variable $x \in X$, and domain value $a \in D$, there is a vertex $x[a]$ in $G$.

2. There is an edge $(x[a], y[b]) \in E(G)$ iff $(a, b)$ violates the constraint between $x$ and $y$, i.e. if $xRy$ and $(a, b) \notin R$.

We assume there is exactly one constraint between any pair of variables; any variables without explicit constraints are assumed to be constrained by the universal constraint which allows all values.

Definition 5 is actually the complement of the graph defined as the microstructure by Jéou [79]. The reason for this is purely one of convenience. It makes sense ‘intuitively’ to have the edges of a graph
denoting incompatible values—we will see an example of this later—and the algorithms we will look at later are rather easier to both describe and understand with this formulation.

Since we add edges between incompatible values, especially between the different possible values for a variable, each variable will give rise to a clique—if the domain has \(d\) elements, we get a clique of size \(d\). Figure 3.1 shows the situation when \(d = 2\) and \(d = 4\). We let \(x[i]\) denote the vertex in the microstructure graph which corresponds to the assignment \(x \mapsto i\) from now on (cf. the discussion in Section 3.2.)

Now for the promised example. Let \(\Theta = (X, D, C)\) be a binary CSP instance with \(n\) variables and \(d\) domain elements, and let \(G\) be the microstructure graph of \(\Theta\). Since each \(x \in X\) give rise to \(d\) vertices, \(G\) will obviously have \(n \cdot d\) vertices. Let \(x\) be an arbitrary variable in the problem, and let \(i\) be one of the possible values for it. Each of the neighbours in \(N_G(x[i])\), by definition, is violating some constraint in \(C\), given that \(x = i\). So if we try to assign the value \(i\) to \(x\), we can remove the set \(N_G(x[i])\) from the graph, since none of its members can be part of the solution we are currently constructing. Repeating this for every variable in the problem, we will eventually end up with an empty graph and a set of vertices which forms an independent set. If we have chosen these vertices wisely, there will be precisely \(n\) of them, and since they all represent an assignment of a value to a variable in the original problem, it is straightforward to extract a solution to \(\Theta\) from them. The following theorem explains why this is the case. (The theorem has been rephrased to fit our setting.)

**Theorem 6 (Jégou [79]).** A \((d, 2)\)-CSP \(\Theta = (X, D, C)\) has a solution if and only if the microstructure graph of \(\Theta\) contains an independent set of size \(|X|\).

So a general outline of a microstructure based algorithm would look something like this:

- Construct the microstructure graph, if necessary augmented to fit the problem, e.g. adding weights to the vertices.
3.5 The Split and List Method

Algorithm 3 Microstructure based algorithm for \((d, 2)\)-CSP.

\(\text{MS-CSP} \ (\Theta = (X, D, C))\)

1. Let \(G\) be the microstructure of \(\Theta\)
2. \(f := \text{MIS}(G)\)
3. \(\text{if } |f| = |X| \text{ then}\)
4. \(\text{return } f\)
5. \(\text{else}\)
6. \(\text{return } \text{“unsatisfiable”}\)
7. \(\text{end if}\)

– Search for an independent set according to some criteria, as given by the problem, e.g. an independent set of maximum size or weight.

If we let \(\text{MIS}\) denote an algorithm for finding a maximum independent in a graph, then a microstructure based algorithm for solving \((d, 2)\)-CSPs would look like Algorithm 3.

3.5 The Split and List Method

In a paper by Williams [114], an approach called \textit{split and list} is used to construct algorithms for some optimisation problems — notably the \textsc{Max CSP} problem. The method is radically different from the usual branch-and-bound methods, being similar to algorithms by Schroeppel & Shamir [106] and Horowitz & Sahni [76], and has the disadvantage of giving algorithms which require an exponential amount of space, but, on the other hand, the algorithms are surprisingly efficient — especially for large domains. We will make use of this method later, so let us have a look at it in a small example.

Given a CSP instance \(\Theta = (X, D, C)\), we begin by \textit{splitting} \(X\) into a number of roughly equal parts. While it is possible to use any number of partitions, we will settle for 3 in order to simplify the discussion.
Now for each of these parts, we list all possible assignments for the variables in the respective partitions. With a domain size of $d$, we get $d^{n/3}$ assignments in each of the partitions, $3d^{n/3}$ in all. Using these assignments, we then build a graph in which we will find the solution we are looking for—usually in the form of a clique of size 3 (or a multiple thereof.) In the following, $\omega \in \mathbb{R}$ is the smallest real such that, for $\varepsilon > 0$, matrix multiplication over a ring can be done in $O(n^{\omega + \varepsilon})$ time.

**Theorem 7 (Něsetřil & Poljak [99]).** For $k \in \mathbb{Z}^+$, cliques of size $3k$ can be found in undirected graphs in $O(n^{\omega k})$ time.

**Proof.** Consider $k = 1$. Given a graph $G$ let $A(G)$ be the adjacency matrix of $G$, i.e. a $|V(G)| \times |V(G)|$ matrix where a 1 in position $(i, j)$ denotes the existence of an edge between vertices $v_i$ and $v_j$, and a 0 denotes the absence. The trace of a matrix $M$, written $\text{tr}(M)$, is the sum of the diagonal entries in $M$. We can compute $\text{tr}(A(G)^3)$ in two matrix multiplications, and obviously $\text{tr}(A(G)^3)$ is non-zero iff there is a triangle in $G$. For $3k$-cliques, $k > 1$, we build the graph $G_k$ with vertex set $V_k = \{\text{all } k\text{-cliques in } G\}$, and edges

$$E_k = \{(v_1, v_2) \mid v_1, v_2 \in V_k, v_1 \cup v_2 \text{ is a } 2k\text{-clique in } G\}.$$ 

Now each triangle in $G_k$ corresponds to a unique $3k$ clique in $G$ and, consequently, $\text{tr}(A(G_k)^3) \neq 0$ if and only if there is a $3k$-clique in $G$, which can be determined in $O(|V(G)|^{\omega k})$ time. Finding an explicit $3k$-clique, given that one exists, may be done using the $O(|V(G)|^\omega)$ time algorithm by Alon & Naor [5].

It was shown by Coppersmith & Winograd [39] that $\omega < 2.376$, so we can find a clique of size 3 in $O(n^{2.376})$ time. Furthermore, which was shown by Williams [114], we can, using the same approach, find the number of $3k$-cliques.

The details of the construction of this graph naturally depend largely on the problem at hand, but the following example should hopefully clarify the use of this method: Consider a $(2, 2)$-CSP instance $\Theta = (X, D, C)$, where:
3.5. The Split and List Method

Figure 3.2: A 2-colourable graph.

- \( X = \{x_1, x_2, x_3, x_4, x_5, x_6\} \), thus \(|X| = 6\),
- \( D = \{0, 1\} \), and
- \( C = \{x_1 \neq x_3, x_2 \neq x_3, x_3 \neq x_4, x_4 \neq x_5, x_5 \neq x_6\} \).

(The problem is actually a graph colouring problem, namely 2-colouring of the graph in Fig. 3.2.) Note that since \((2, 2)\)-CSPs can be solved in polynomial time, it is of course a rather poor idea to use the split and list method on this problem, however, since the graph we build is exponential in the size of the problem, we would quickly run out of space if the example was bigger.

Proceed as follows: We split the list of variables into 3 parts, \( P_1 = \{x_1, x_2\}, P_2 = \{x_3, x_4\}, P_3 = \{x_5, x_6\} \) (this is not the only possible partitioning, any partitioning will do.) For each of these, we list the possible assignments; i.e. we disregard the constraints and blindly assign values to the variables:

\[
\begin{align*}
P_1 & : \{x_1 = 0, x_2 = 0\} & P_2 & : \{x_3 = 0, x_4 = 0\} & P_3 & : \{x_5 = 0, x_6 = 0\} \\
& \{x_1 = 0, x_2 = 1\} & & \{x_3 = 0, x_4 = 1\} & & \{x_5 = 0, x_6 = 1\} \\
& \{x_1 = 1, x_2 = 0\} & & \{x_3 = 1, x_4 = 0\} & & \{x_5 = 1, x_6 = 0\} \\
& \{x_1 = 1, x_2 = 1\} & & \{x_3 = 1, x_4 = 1\} & & \{x_5 = 1, x_6 = 1\} \\
\end{align*}
\]

Next we build a graph from these. Since any solution will do, we do not need to take any special measures when building the graph — all we have to do is add edges between vertices which correspond to a
3. Preliminaries

valid, partial assignment. The resulting graph is shown in Fig. 3.3. For simplicity, we name the vertices by the domain values the variables assume, e.g. 01 in partition $P_1$ denotes the assignment $x_1 = 0, x_2 = 1$ above, and if a vertex contains an inconsistent assignment, i.e. one which violates one or more constraints, we omit the edges going to and from it.

Given this graph, all we have to do in order to find a satisfying assignment is to find a clique with 3 elements. For example, the assignment $\{x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 1, x_4 \mapsto 0, x_5 \mapsto 1, x_6 \mapsto 0\}$, given by the vertices 00 in $P_1$ and 10 in both $P_2$ and $P_3$. We will see other ways of building the graph in later sections.

3.6 Complexity Issues

A central part of the discussions that follow in later sections concerns computational complexity, thus it seems prudent to spend some time elaborating on what we are talking about. A thorough treatment is, of course, beyond the scope of this thesis, and the reader is referred to, e.g., Bovet & Crescenzi [29], for a more in-depth treatment. A somewhat less in-depth, but nevertheless good, introduction can be found in Cormen et al. [40].

What it boils down to is the need for a way of measuring the running time of an algorithm without having to implement it. Fur-
Algorithm 4 Algorithm \textit{FindMax}.

\begin{algorithm}
\begin{algorithmic}
\State $q := -1$
\For{$i := 1$ to $n$}
\State \If{$A[i] > q$}
\State $q := A[i]$
\EndIf
\EndFor
\State \Return $q$
\end{algorithmic}
\end{algorithm}

Furthermore, it is fairly useless to express the running time using the unit “seconds on a Pentium IV 2.8GHz with 1024Mb RAM and...”—thus we need to isolate something which is completely independent of the hardware but still captures the idea of running time. One possibility is to count the number of steps in the program, where a “step” is some basic operation, such as addition or multiplication. Now rather than just counting the number of steps, we compare the (size of the) input with the number of steps the program needs to produce an answer. For example, say we want to find the largest among $n$ positive integers. The program might look like Algorithm 4.

Line 1 is executed once. Line 2 is executed $n$ times, as is line 3, while line 4 may, or may not, be executed on each iteration. We will exclusively be interested in worst-case performance, thus we assume Line 4 really is executed as often as possible. (It is fairly easy to see when this worst-case scenario occurs here, but this is not always the case.) Finally, Line 7 is only called once. Summing this up, we find that the search will include $1 + n + n + n + 1 = 3n + 2$ steps given an input of size $n$. Actually, it is probably at least $4n$ since there is usually some things happening ’behind the scenes’—e.g. the jump from Line 6 to line 2. This is, however, strictly language specific, and we really do not want to concern ourselves with those details either. The running time of the algorithm is somewhere between $3n$ and $4n$ for input of size $n$.

In the remainder of this thesis we will only be interested in the asymptotic upper bound of the running times—some times called...
"big-Oh," or "ordo." Formally, for a given function $f(n)$, we use $O(f(n))$ to denote the set

$$\{g(n) \mid \text{there exist } c, n_0 > 0 \text{ s.t. } 0 \leq g(n) \leq cf(n) \text{ for all } n \geq n_0\}.$$ 

where $n, n_0 \in \mathbb{Z}^+$ and $c \in \mathbb{R}^+$. Using this notation, it is clear that the $\text{FindMax}$ algorithm we discussed previously has a running time (or time complexity) in $O(n)$—since we can ignore the constants 3 and 4. (In fact, we can ignore any constant as long as it is, indeed, a constant.) Furthermore, through careful rounding off of exponential time complexities, we can afford to drop any polynomial factors; For example, rather than $O\left(n^{3.44225}n\right)$, we would write $O\left(2^{4.4423n}\right)$

and let $O$ “swallow” the polynomial factors.

Determining the time complexity is not always as straightforward as in the example above. Several of the algorithms we will look at are recursive and sometimes a rather complicated case study is needed in order to determine the running time.

Assume we have a situation as in Fig. 3.4, and that the edges between the vertices denote incompatibility—i.e. it cannot be the case that we have $x$ together with any of the $y$’s in a solution. Consequently, this means that whenever we choose $x$ during a search, all of the $y$’s are excluded from being candidates in a solution, and thus

![Figure 3.4: The vertex $x$ is incompatible with $y_1, y_2, y_3$.](image-url)
we have decreased the number of vertices not by 1, i.e. $x$, but by 4, $x, y_1, y_2, y_3$. Given that we have a choice of either including or excluding $x$, we get two branches; one where we decrease the number of vertices by 4 and one where we only decrease it by 1 (when $x$ is excluded, we cannot say anything about the $y$'s.) The running time of the algorithm can thus be described as

$$T(n) = T(n - 4) + T(n - 1) + p(n)$$

The last term of the sum, $p(n)$ could be, for example, time spent joining the results of the two branches, or searching for a variable with a particular property to branch on, etc. As long as it is a polynomial, anything goes. Solving this (see e.g. Kullmann [88]), we find that $T(n)$ is in $O(\tau(1, 4)^n)$, where $\tau(1, 4)$, sometimes called the work factor, is the largest real-valued root of $1 - 1/x - 1/x^4$. Solving for $x$, we get that $T(n) \in O(1.3803^n)$, regardless of the polynomial $p(n)$ and any boundary conditions.

In general, when we have recurrence relations of the form

$$T(n) \leq \sum_{i=1}^{k} T(n - r_i) + \text{poly}(n),$$

they satisfy

$$T(n) \in O(\tau(r_1, \ldots, r_k)^n)$$

where $\tau(r_1, \ldots, r_k)$ is the largest, real-valued solution to the equation

$$1 - \sum_{i=1}^{k} x^{-r_i} = 0.$$ 

We will use this frequently.

Some of the algorithms we present in this thesis have an exponential space requirement, and reasoning similar to what has been presented in this section holds for this case. However, most of the subsequent discussion will focus on polynomial space algorithms, with those using exponential space included mostly for completeness.
Part II

Methods
Chapter 4

The Covering Method

Whether to concentrate or to divide your troops, must be decided by circumstances.

Sun Tzu, The Art of War

The first of the methods we will look at is called the covering method. A rather crude version of this was first described in [8], where it was used to construct a quantum computation algorithm for solving binary CSPs. At the time it was merely a useful tool, but we soon realised that this method could actually be useful in a rather more general way, and it was refined further in the papers [10], where it was applied to the counting problem for CSPs, and [11], where the version we present here is described. However, primitive though it may be, this first version of the method demonstrates the workings very nicely, thus in order to get a good intuition for how to use coverings, we will start with constructing an algorithm for solving \((d, 2)\)-CSPs using this earlier method.

Chapter Outline

This chapter begins with the introduction of coverings in Section 4.1, which also contains an example where we try to convey the intuition behind them, and then moves on to formally define what a covering is via the Covering Theorem in Section 4.2. Section 4.3 gives an example
of how the application of the Covering Theorem gives us an efficient algorithm for solving \((d, l)\)-CSPs

### 4.1 Introduction to Coverings

The basic idea is to take the CSP instance and then create a number of 2-SAT (or \((2, 2)\)-CSP) instances with the property that one of these instances has a solution if and only if the original instance has a solution, i.e., we want to cover the CSP instance using \((2, 2)\)-CSP instances. (Hence the name of the method.) Since 2-SAT can be solved in polynomial time \([17]\), the covering has to contain a super-polynomial number of them in order to qualify as a covering, if we assume \(P \neq NP\).

Now let \(\Theta\) be a \((d, 2)\)-CSP instance with variables \(X\), domain \(D\) and constraint set \(C\). For now we will assume the domain contains an even number of elements. For each of the variables in \(X\), we create \(d\) boolean variables, one for each possible value it can assume. For example, if variable \(x\) can assume the values \(a, b, c\), we get three variables \(x[a], x[b]\) and \(x[c]\). Each of these new variables has an interpretation attached to them; if, say, \(x[a]\) is true, then this means that \(x\) is assigned the value \(a\). (Note the similarity to microstructures.) With these \(nd\) variables, we can begin to construct the 2-SAT instances in the covering.

The clauses we will construct are of two basic types. First of all, there are clauses used to represent the constraints in \(C\), and these are the same for all of the instances in the covering. Each constraint \(R(x, y)\) in \(C\) is represented by formula \(C_{R(x,y)}\), defined as:

\[
C_{R(x,y)} := \bigwedge_{a,b \in D \atop (a,b) \notin R} (\neg x[a] \lor \neg y[b]).
\]  

(4.1)

In other words, if the constraint between \(x\) and \(y\) forbids the assignments \(x := a, y := b\), then we get the clause \((\neg x[a] \lor \neg y[b])\), which simply means that \(x\) cannot be assigned \(a\) if \(y\) is assigned \(b\), and vice versa. For example, if we have \(x \neq y\), where ‘\(\neq\)’ has the standard
interpretation, and the domain is \( \{1, 2, 3\} \), then we would get the formula

\[
C_{x \neq y} := (\lnot x[1] \lor \lnot y[1]) \land (\lnot x[2] \lor \lnot y[2]) \land (\lnot x[3] \lor \lnot y[3]).
\]

Now this is clearly not enough. In the formula for \( C_{x \neq y} \), there is nothing to prevent \( x[1] \) and \( x[2] \), say, to be true at the same time, and this would mean that \( x \) is assuming two values simultaneously in the solution. Since this is not desirable, we have to add clauses to prevent this, i.e. we have to enforce that exactly one of \( x[1], x[2] \), and \( x[3] \) is true, or, more generally, that there is exactly one \( i \in D \) for which \( x[i] \) is true. This is were the ‘trick’ that makes the method so successful comes in; rather than forcing a variable to assume a single value, we restrict it to assume either of two values. (Later we will generalise this to an arbitrary number of values, but for now two will be quite enough.)

So, say we want either \( x[a] \) or \( x[b] \) (but not both) to be true. This can be done by adding the clauses

\[
(x[a] \lor x[b]) \land (\lnot x[a] \lor \lnot x[b]) \land \bigwedge_{c \neq a, c \neq b} (\lnot x[c]). \tag{4.2}
\]

This is interpreted as: “\( x \) can be either \( a \) or \( b \), but not both, and nothing else.” Returning to the previous example with domain \( \{1, 2, 3\} \), we would get the clauses

\[
(x[1] \lor x[2]) \land (\lnot x[1] \lor \lnot x[2]) \land (\lnot x[3])
\]

if we wanted to restrict \( x \) to the values 1 and 2.

The pairing of values can be done in any way we like, as long as every value occurs in at least one pair. Since we have assumed an even number of domain elements, we get \( d/2 \) such pairs for each variable and combining these with the clauses in formula (4.1), we get \((d/2)^n\) 2-SAT instances to consider, giving a running time in \( \mathcal{O}((d/2)^n) \) (for even sized domains.) This is already an impressive improvement over the naïve enumeration algorithm (which of course has a running time of \( \mathcal{O}(d^n) \)), and it is not very far from the running times of the probabilistic algorithms of Eppstein [51] and Feder & Motwani [54], which
4.1. Introduction to Coverings

Figure 4.1: ‘New’ clauses introduced by the covering approach.

have running times of $O((0.4518d)^n)$ and $O((d!)^{n/d})$, respectively. At the time of this writing, they are the fastest known algorithms for $(d, 2)$-CSPs, with Eppstein’s algorithm taking the lead for domains between 3 and 10, but which is overtaken by its opponent for $d > 10$. Using the latest version of the covering method, we will arrive at an algorithm which is an almost perfect derandomisation of Eppstein’s algorithm.

Now there is one nagging irregularity in the covering we have constructed so far; it cannot handle domains of odd size. In the original version, this did not bother us too much, and the problem was solved by simply adding an element to the domain—a ‘dummy’ element which invalidated any solution in which it occurred. While this certainly is the easiest solution, it is also quite inefficient. It gives a running time of $O(\lceil d/2 \rceil^n)$, which means that, say, an instance with domain size 3 has the same running time as an instance with domain size 4. Since this seems rather unintuitive, we did devote some extra time studying the cases with 3 and 5 domain elements, and even though no ‘final’ solution presented itself, we observed that it is possible to get coverings smaller than $\lceil d/2 \rceil^n$.

Just to give a taste of what we did, let us consider the problem of covering problems with domain size 3 using 2-SAT instances. Rather than just looking at one variable at a time, we now consider pairs of variables (with corresponding pairs of values.) If we consider the case with domain $\{1, 2, 3\}$, we get something like what is shown in Fig. 4.1. Here, $x_i, y_j$ is shorthand for $x[i] \lor y[j]$. It is fairly obvious that we do not need the entire table in order to make sure every possible assignment to $x$ and $y$ is covered; for example, columns 2,
4. The Covering Method

Table 4.1: The case with \( D = \{1, 2, 3\}, V = \{x, y\} \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x_1, y_1 )</th>
<th>( x_2, y_1 )</th>
<th>( x_3, y_1 )</th>
<th>( x_1, y_2 )</th>
<th>( x_2, y_2 )</th>
<th>( x_3, y_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1 2</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>1 3</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>2 1</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2 2</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2 3</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>3 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>3 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>3 3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

4 and 9 (the boxed entries) will clearly suffice. We can now use this result to lower the running time of the covering based algorithm simply by replacing the clauses given by formula (4.2) with either of the clauses from Fig. 4.1 (one for each case.) In this way, rather than \( \lceil \frac{3}{2} \rceil n = 2n \) cases for each variable, we get \( 3n / 2 \) cases, and thus we can lower the running time for (3,2)-CSP instances from \( O(2^n) \) to \( O(\frac{3n}{2}) \approx O(1.7321^n) \), which is a definite improvement.

There is of course no particular reason to stop at considering two variables at a time. In fact, in [8] we considered 4 variables and, by letting a computer analyse the resulting table (which was rather larger than Table 4.1) we found that at least 8 cases were needed in order to make a full covering. Consequently, with 8 cases and 4 variables per case, we get \( 8^{n/4} \) instances in the covering and the algorithm has a running time of \( O(8^{n/4}) \approx O(1.6818^n) \). Of course, this approach was not feasible in the long run. We briefly considered looking at larger domains and more variables, but already with 4 variables and domain size 5 we get a prohibitively large number of clauses to consider — around \( 2.34 \cdot 10^{63} \) — and this is simply not an option.
4.2 The Covering Theorem

With this in mind, we will now introduce the latest version of the covering method. First, we need to formalise what we mean by a covering.

**Definition 8.** Let \( n, d \) and \( e \) be positive integers, and let \( D \) be a finite set with \(|D| = d\). Define the set \( \mathcal{L}_{d,e}^n \) as

\[
\mathcal{L}_{d,e}^n := \prod_{i=1}^{n} \{ E_i \mid E_i \subseteq D, |E_i| = e \}.
\]

A subset \( Q \subseteq \mathcal{L}_{d,e}^n \) is called an \( e \)-covering of \( D^n \) if \( \bigcup Q = D^n \). We let \( c_{d,e}^n \) denote the smallest size of an \( e \)-covering of \( D^n \). Obviously, \( c_{e,d}^n = (d/e)^n \) if \( e \mid d \).

While this looks complicated, it is actually quite straightforward if we consider what was discussed in Section 4.1, and we can use the example we looked at to concretise the definition: \( n \) is, as is usually the case, the number of variables in the problem, \( d \) the domain size of the problem we wish to cover, and \( e \) the domain size of the instances we use in the covering. Using Table 4.1, we can ‘work backwards,’ and build an example which should clarify it even further: We want to cover a \((3,2)\)-CSP using \((2,2)\)-CSP instances. For simplicity, we assume the problem only has two variables and order them, letting \( x \) be the first variable and \( y \) be the second. Consequently, we get \( n = 2 \), \( d = 3 \) and \( e = 2 \) in Definition 8, and we can immediately get a 2-covering from the table by noting the assignments corresponding to the boxed columns. For \( x \), we have \( \{1,2\}, \{1,3\} \) and \( \{2,3\} \), while for \( y \) we have \( \{1,3\}, \{1,2\} \) and \( \{2,3\} \). Thus we can construct the covering:

\[
Q = \{1,2\} \times \{1,3\} \cup \{1,3\} \times \{1,2\} \cup \{2,3\} \times \{2,3\}
\]

\[
= \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}
\]

It is easy to verify that this really is a covering since the last set is equal to \( \{1,2,3\}^2 = D^2 \).
4. The Covering Method

Since the size of the covering immediately gives the running time of the resulting algorithm, we are of course very much interested in finding the size of a smallest covering, i.e. determining $c_{d,e}^n$ in Definition 8. The following theorem tells us what we can expect:

**Theorem 9.** For any $\varepsilon > 0$ there exists an integer $n_0$ such that

$$c_{d,e}^n \leq \left(\frac{d}{e} + \varepsilon\right)^n$$

whenever $n \geq n_0$.

**Proof.** Pick a random $E = \prod_{i=1}^n E_i \in \mathcal{P}_{d,e}^n$ with uniform distribution and let $v$ be an arbitrary vector in $D^n$. Then, the probability $\Pr(v_i \in E_i) = e/d$ implies that $\Pr(v \notin E) = 1 - (e/d)^n$. Now pick a random $Q$ with uniform distribution from the set of $t$-subsets of $\mathcal{P}_{d,e}^n$.

For each $v \in D^n$, let $A_v$ be the event that $v \notin \bigcup Q$. Then,

$$\Pr(A_v) \leq (1 - (e/d)^n)^t < \exp(-t(e/d)^n).$$

To see this, note that $(1-1/x)^x < \exp(-1)$ whenever $1/x \leq 1$, and let $x = (d/e)^n$ (and thus $1/x = (e/d)^n$). Raise both sides of the equation to $t(e/d)^n$ and the inequality follows.

The event that $Q$ will fail to be a covering is equal to the union of $A_v$ over all $v$.

$$\Pr\left(\bigcup_{v \in D^n} A_v\right) \leq \sum_{v \in D^n} \Pr(A_v) < d^n \exp(-t(e/d)^n).$$

If we let $t = cn(d/e)^n$, with $c \leq \ln d$, we have $d^n \exp(-(e/d)n) < 1$. This shows the existence of a covering of size $cn(d/e)^n$. We note that for sufficiently large $n_0$, we have for all $n \geq n_0$

$$\left(\frac{d}{e} + \varepsilon\right)^n \geq cn(d/e)^n$$

which concludes the proof. \qed
Unfortunately, Theorem 9 does not give us an explicit way of finding the coverings. If we want to know more than just the existence of a covering, we have to come up with a way of doing this ourselves. First, we note that $d$ and $e$ are constants, thus for any fixed $\varepsilon$, we can choose an integer $n_0$, which we now know from the theorem exists, and compute an $e$-covering for domain size $d$ with $n_0$ variables. This result we can then store in a table — and since all of the above are constants, the table will be of constant size. If we split the variables into groups of size $n_0$ we can apply the tabulated coverings on each of these group. If $n$ is not a multiple of $n_0$, then we get a remainder of constant size which we can take care of using, for example, a brute-force approach.

Of course, this is not the only way of finding the coverings. If the size of the table — which, though of constant size, will be very large — is a problem, then one could adopt a probabilistic approach to finding the coverings.

### 4.3 Algorithms for $(d, l)$- and $(d, 2)$-CSPs

Now that we know the size of a minimal covering, as well as how to find them, we can revisit the problem we started with, namely constructing an algorithm for solving $(d, 2)$-CSPs. The algorithm we present here is deterministic and, as was previously mentioned, is an almost perfect derandomisation of Eppstein’s algorithm for the same problem.

In the example in the previous section, we did a bit of ‘cheating.’ We used an algorithm for solving $(2, 2)$-CSPs to solve problems which had a lot more than two domain values. This worked since in each of the instances, a single variable could only assume two values, although it was not necessarily the case that two variables were restricted to the same pair of values. We will now give a general lemma for this, and it turns out to be very useful; it applies to both of the methods as well as a range of different problems. Here we use the original formulation, which concerns CSPs, but it can easily be reformulated to hold for other problems.
Lemma 10. Assume there exists an \( O(\alpha^n) \) time algorithm for solving \((e, l)\)-CSP. Let \( \mathcal{I}_e \) denote the set of \((d, l)\)-CSP instances satisfying the following restriction: For every \((X, D, C) \in \mathcal{I}_e\), and every \( x \in X \), there exists a unary constraint \((x; S)\) in \( C \) such that \(|S| \leq e\). Then, the CSP problem restricted to instances in \( \mathcal{I}_e \) can be solved in \( O(\alpha^n) \) time.

Proof. For each variable \( x \) in \((X, D, C)\), we know that it can be assigned at most \( e \) out of \( d \) values due to the constraint \((x; S)\). Thus, we can modify the constraints so that every variable picks its values from the set \( \{1, \ldots, e\} \). This transformation can obviously be done in polynomial time, and the resulting problem is now an instance of \((e, l)\)-CSP, which can be solved in \( O(\alpha^n) \) time. \( \square \)

We now have everything we need to show the following theorem, which as a corollary will give us the algorithm we set out to find in the beginning of this section.

Theorem 11. If there exists a deterministic \( O(\alpha^n) \) time algorithm for solving \((e, l)\)-CSP, then for all \( d > e \), there exists a deterministic, \( O((d/e + \varepsilon)^n \alpha^n) \) time algorithm for solving \((d, l)\)-CSP.

Proof. First, arbitrarily choose an \( \varepsilon > 0 \). By Theorem 9 there exists an \( n_0 \) such that \( c_{d,e}^{n_0} \leq (d/e + \varepsilon)^n \), thus let \( Q \subseteq \mathcal{Q}_{d,e}^{n_0} \) be an \( e \)-covering of this size. Since \( n_0, d \) and \( e \) are fixed, this covering is of constant size.

Now assume, without loss of generality, that \(|X| \equiv 0 \mod n_0\), and let \( X_1, \ldots, X_K \) be a partitioning of \( X \) into \( K \) subsets such that \(|X_i| = n_0\). Assuming \( X_i = \{x_i^1, \ldots, x_i^n\} \), consider each \( q = \prod q_i \in Q \) as a function from \( X_i \) to \( 2^D \), i.e. \( q(x_i) = q_i \). Note that \(|q(x_i)| = e \) for all \( x \). Let \( Q^K \) denote the set of all functions from \( \{1, \ldots, K\} \) to \( Q \).

The resulting algorithm for \((d, l)\)-CSP is given as Algorithm 5, and correctness follows immediately from the fact that \( Q \) is a covering of each set of variables \( X_1, \ldots, X_K \), and the time complexity is in \( O(\alpha^n|Q|^K) \). Since \(|Q|^K \leq (d/e + \varepsilon)^{n_0 n / n_0} = (d/e + \varepsilon)^n \), the theorem follows. \( \square \)
4.3. Algorithms for \((d, l)\)- and \((d, 2)\)-CSPs

Algorithm 5 Covering based algorithm for \((d, l)\)-CSP.

\(C-(d, l)\)-CSP \((\Theta = (X, D, C))\)

1. for each \(f \in Q^K\) do
2. Let \(\Theta_f = (X, D, C \cup \{(x_i^j; f(i)(x_j^i)) \mid 1 \leq i \leq K \land 1 \leq j \leq n_0\})\)
3. solve \(\Theta_f\) using Lemma 10
4. if \(\Theta_f\) has a solution then
5. return “satisfiable”
6. end for
7. return “unsatisfiable”

Armed with Theorem 11, all we have to do is locate a fast, deterministic algorithm for solving \((e, l)\)-CSPs for some \(e\), and the theorem generalises this to any \(d > e\). Corollary 12 contains the result of combining the theorem with the (currently) fastest known deterministic algorithms for some CSP problems:

Corollary 12. There exist deterministic algorithms for solving

1. \((d, 2)\)-CSP, \(d \geq 4\) in \(O((0.4518d + \varepsilon)^n)\),
2. \((d, 3)\)-CSP, \(d \geq 2\) in \(O((0.7365d + \varepsilon)^n)\), and
3. \((d, l)\)-CSP, \(d > 2, l > 3\) in

\[\mathcal{O}\left(\left(d - \frac{d}{l+1} + \varepsilon\right)^n\right)\]

time.

Proof. By simply combining Theorem 11 with

1. the \(O(1.8072^n)\) time algorithm for \((4, 2)\)-CSP from [51],
2. the \(O(1.473^n)\) time algorithm for \((2, 3)\)-CSP from [30], and
3. the \(O\left((2 - \frac{2}{l+1})^n\right)\) time algorithm for \((2, l)\)-CSP from [45],

the result follows. \(\square\)
4.4 A Probabilistic Approach to Coverings

The main disadvantage with the previously described covering method is, as we have already mentioned, the explicit construction of the coverings. We will see later that the partitioning method offers a much simpler construction, but at the cost of some loss of speed. In an effort to mix the best qualities of the two methods, we came up with a probabilistic covering approach, where we have sacrificed determinism, but avoid both the problem of determining the coverings and the decrease in speed in the partitioning method.

The intuition behind this method is quite straightforward; we select a single partition size — most likely this is the domain size for which we have the fastest algorithm — and then choose a large number of instances where the variables have been restricted to take their values from sets of this size. By choosing a large enough number of these restricted instances, we get a good probability of finding the optimal solution among them. Restricting a problem with domain size $d$ to partitions of size $e$, we get a success probability of $1 - \frac{1}{e}$, or roughly 63.2%, by choosing at least $O \left( (d/e)^n \right)$ of the smaller instances. We call this probability “good,” even though this may seem rather low — after all, there is a 36.8% chance of failure. It is, however, possible to achieve arbitrarily low error probability by iterating the algorithm. For example, after only 5 iterations we have a probability of success greater than 99.3%, and after 5 more it is greater than 99.99%. In general, we can get any error probability $\kappa > 0$ by performing $-\lceil \ln \kappa \rceil$ iterations, and for a given $\kappa$, the number of iterations is constant, and thus does not add to the time complexity of the algorithm.

An interesting side effect of this method is that any algorithm we get from it can immediately be derandomised using the covering method, simply by combining the base algorithm with Theorem 11.
4.4. A Probabilistic Approach to Coverings
Chapter 5

The Partitioning Method

We can form a single united body, while the enemy must split up into fractions. Hence there will be a whole pitted against separate parts of a whole, which means that we shall be many to the enemy’s few. And if we are able thus to attack an inferior force with a superior one, our opponents will be in dire straits.

Sun Tzu, The Art of War

The second method for algorithm construction we will discuss is called the partitioning method. Unlike the covering method, which was discovered almost by accident, we knew what we were looking for when we developed the partitioning method — but we originally applied it to the ‘wrong’ problem. Theorem 9 provides us with the minimum size of a covering, but unfortunately there is the ever-present ε to deal with. Since the proof of the theorem is only a proof of existence and does not present us with the actual covering, we tried a different, constructive, approach. Unlike the previous method, the partitioning method allows us to use any domain size we like as a basis, provided we have an algorithm for dealing with that particular case. It is interesting to note that this approach to algorithm construction has been used at least twice before, but has apparently not been investigated in any detail. It appears in Lawler’s [91] algorithm for determining 2k-colourability of a graph, and a rather more elaborate version can
be found in in Wang & Rushforth’s local-search algorithm in [112] (a very nice, condensed, description of this algorithm can be found in a survey by Gu et al. [71].)

Chapter Outline

First, Section 5.1 begins by trying to convey the intuition behind the method, as well as formally define it. Then, in Section 5.2, we show how to apply the method to \((d, l)\)-CSPs in order to get an algorithm for solving them. Finally, in Section 5.3, the application of partitionings to colouring problems, i.e. CSPs where the only allowed constraint is disequality, is discussed.

5.1 Introduction to Partitionings

First, we need to define what a partitioning is.

**Definition 13.** A partitioning \(P = \{P_1, P_2, \ldots, P_m\}\) of a domain \(D\) is a division of \(D\) into \(m\) disjoint subsets such that \(\bigcup P = D\). A \(k\)-partition is an element of \(P\) with \(k\) elements. Given a partitioning \(P\), we let \(\sigma(P, k)\) denote the number of \(k\)-partitions in \(P\). Since the actual elements in the subset \(P_i \in P\) is often less interesting than their number, we let the multiset \([|P_1|, \ldots, |P_m|]\) represent \(P\).

The idea for this method originally came from the fact that the running time of the \((4, 2)\)-CSP algorithm presented by Eppstein [51] depends on the number of variables taking their values from domains of sizes 3 and 4. In the next section we will look at this in more detail, but for now, we just note that if \(n_i\) denotes the number of variables with domain size \(i\), then the algorithm has a time complexity of \(O(1^{n_1 + n_2} \alpha^{n_3} \beta^{n_4})\) (of course, \(1^n = 1\) for any \(n\)) where \(\alpha \approx 1.3645\) and \(\beta \approx 1.8072\).

As an example, we will look at the case with \(d = 7\). For each variable, the domain is partitioned into two parts—i.e. we have a partitioning \(P = [3, 4]\). Say we have the partitions \(P_3 = \{1, 2, 3\}\) and \(P_4 = \{4, 5, 6, 7\}\). Now, for each variable, we restrict it to take its
5. The Partitioning Method

Algorithm 6 Algorithm for solving $(7, 2)$-CSPs.

$P_{(7, 2)}$-CSP $(\Theta = (X, D, C))$

1. for each total function $f : X \to P$ do
2. Let $\Theta_f$ be the instance where each variable $x$ is limited to take its value from the domain $f(x)$.
3. Solve $\Theta_f$ using Lemma 10
4. if $\Theta_f$ has a solution then
5. return “satisfiable”
6. end for
7. return “unsatisfiable”

values from either $P_3$ or $P_4$ and thus get a subproblem which we can solve using Eppstein’s algorithm (note that Lemma 10 applies here.) The result is Algorithm 6.

Since each variable has two possible restrictions — $P_3$ or $P_4$ — we have $2^n$ combinations to explore in order to be sure we have solved the original problem. For each such combination, some variables are assigned domain $P_3$ and the rest are assigned $P_4$. If there are $k$ variables restricted to $P_3$, there must be $n - k$ restricted to $P_4$, so the algorithm of Eppstein will have a running time of $O(\alpha^k \beta^{n-k})$ for each instance. Consequently, the total running time for solving the entire problem is in

$$O \left( \sum_{k=0}^{n} \alpha^k \beta^{n-k} \right)$$

and using the binomial theorem, this simplifies to $O((\alpha + \beta)^n)$. Replacing $\alpha$ and $\beta$ with their numerical values, we get a time complexity of $O(3.1717^n)$. In contrast, by using the covering method and Corollary 12, we would end up with an $O((3.1626 + \varepsilon)^n)$ time algorithm.

Since the time complexity of a partitioning based algorithm depends largely on the partitioning, we will use the notation $T(P)$ for the running time in the following section. In the example we just discussed, the time complexity would thus be $T([3, 4]) \in O(3.1717^n)$. 
The function $T$ here has an interesting property: Given two partitionings $P = [a_1, \ldots, a_k]$ and $P' = [b_1, \ldots, b_m]$, where $T(P) < T(P')$, i.e. the partitioning $P$ gives a faster running time than the partitioning $P'$, it holds that if we add further partitions, say by increasing the domain of the problem and partitioning the new elements into $Q = [c_1, \ldots, c_l]$, it also holds that $T(P \cup Q) < T(P' \cup Q)$.

**Definition 14.** A function $T$ with the property that

$$T([a_1, \ldots, a_k]) < T([b_1, \ldots, b_m])$$

if and only if

$$T([a_1, \ldots, a_k, c_1, \ldots, c_l]) < T([b_1, \ldots, b_m, c_1, \ldots, c_l])$$

for all choices of $a_1, \ldots, a_k, b_1, \ldots, b_m, c_1, \ldots, c_l$, is said to be extendible.

The extendibility of the function describing the time complexity of partitioning based algorithms is important in the proof of optimality, which we will see in the next section (and later, when we study other applications of partitioning.)

### 5.2 Algorithms for $(d, l)$- and $(d, 2)$-CSPs

The idea we used for the $(7, 2)$-CSP case can easily be extended to arbitrary domains: Let $P = \{P_1, P_2, \ldots, P_m\}$ be a partitioning of $D$ where all partitions have size at most 4 (each partition has to contain less than 5 elements, since Eppstein’s algorithm is nondeterministic for domains larger than 4), and $\sigma(P, i) = p_i$, $1 \leq i \leq 4$. Thus we get a running time $T(P) = T(\{P_1, P_2, \ldots, P_m\})$ of

$$\mathcal{O} \left( \sum_{n_1 + n_2 + \ldots + n_m = n} \left( \frac{n}{n_1 \ldots n_m} \right) \prod_{i=1}^{m} (\gamma_i)^{n_i} \right)$$

where $\gamma_i = 1$ if $|P_i| \leq 2$, $\gamma_i = \alpha$ if $|P_i| = 3$, and $\gamma_i = \beta$ if $|P_i| = 4$. This we can simplify to $\mathcal{O} ((p_1 + p_2 + \alpha p_3 + \beta p_4)^n)$, using the multinomial theorem, and the resulting algorithm is found in Algorithm 7.
Theorem 15. Let $A$ be an algorithm for $(e,l)$-CSP with a running time of $O\left(\prod_{i=1}^{e} \alpha_i^{n_i}\right)$, $\alpha_i \geq 1$, when applied to an instance containing $n_i$ $i$-valued variables, $1 \leq i \leq e$. Choose $d$ and a partitioning $P = \{P_1, \ldots, P_k\}$ of $\{1, \ldots, d\}$ such that $|P_i| \leq e$ for all $i$. Then, there exists an algorithm for $(d,l)$-CSP running in $O\left(\left(\sum_{i=1}^{e} \sigma(P,i)\alpha_i\right)^n\right)$ time.

Proof. We will show that Algorithm 7 correctly solves $(d,l)$-CSP. Let $\Theta = (X,D,C)$ be an arbitrary instance of $(e,l)$-CSP with model $M$. The members of $P$ are pairwise disjoint (by definition), so there exists exactly one function from $X$ to $P$ such that $M(x) \in f(x)$ for all $x \in X$. Consequently, Lemma 10 guarantees that the subinstance $\Theta_f$ created on Line 2 will have a solution only if $\Theta$ has a solution. Consequently, the algorithm will return “yes” if and only if $\Theta$ has a solution, and “no” otherwise.

The running time $T$ of the algorithm is bounded by

$$O\left(\sum_{f \in F} \prod_{x \in X} \alpha_{f(x)}\right)$$

where $F$ is the set of (total) functions from $X$ to $P$, i.e. the algorithm will consider all possible combinations of elements from $P$ (restricted domains) and members of $X$ (variables.) The performance of the assumed algorithm $A$ depends only on the domain sizes associated with the variables, and thus gives us

$$T \in O\left(\sum_{k_1+\ldots+k_{|P|}=n} \left(\prod_{i=1}^{|P|} \alpha_{|P_i|}^{k_i}\right)\right)$$

where $k_i$ denotes the number of variables whose domain is $P_i$. Applying the multinomial theorem together with the fact that there are $\sigma(P,i)$ sets of size $i$ in $P$, we get

$$T \in O\left(\left(\sum_{i=0}^{|P|} \alpha_{|P_i|}\right)^n\right) = O\left(\left(\sum_{i=1}^{e} \sigma(P,i)\alpha_i\right)^n\right).$$
5.2. Algorithms for $(d,1)$- and $(d,2)$-CSPs

Of course, it remains to be determined exactly how we should partition the domain in order to minimise the running time, and this is fully dependent on the running time of the algorithms we have at our disposal. A general method for determining the optimal partitioning to use is not yet fully developed, but as a guideline for finding one, it is possible to use the idea of “forbidden sub-partitionings” suggested by Johan Wästlund (private communication.)

By exploiting the extendibility of the function describing the time complexity, it is possible to rule out certain partitionings as always being slower than others. If we use the $(4,2)$-CSP algorithm of Eppstein, the following lemma gives the optimal partitioning.

Lemma 16. Assume we have an $O(1^{n_1+n_2}n_3^{\alpha}n_4^{\beta})$ time algorithm for $(4,2)$-CSP, with $\alpha = 1.3645$ and $\beta = 1.8072$, and let $D$ be a domain of size $d \geq 2$. If $d \leq 4$, then the partitioning $P = [d]$ is optimal with respect to the running time of the resulting algorithm, and if $d = 5$, then $P = [2,3]$ is optimal. Otherwise, the following partitionings are optimal:

- if $d \equiv 0 \pmod{4}$, $P = [4,4,4,\ldots,4]$,
- if $d \equiv 1 \pmod{4}$, $P = [3,3,3,4,4,\ldots,4]$,
- if $d \equiv 2 \pmod{4}$, $P = [3,3,4,4,\ldots,4]$,
- if $d \equiv 3 \pmod{4}$, $P = [3,4,4,\ldots,4]$.

Proof. By assumption, the domain has at least 2 elements, so we begin with examining the domain with exactly 2 elements. Clearly, the partitioning $[1,1]$ will always be a worse choice than $[2]$ since $T([1,1]) > T([2])$, so, by extendibility, we should always avoid $[1,1]$. Similarly, for $d = 3$, we have $T([1,2]) > T([3])$, so there will be no occurrences of $[1,2]$ in an optimal partitioning. For $d = 4$, it is the case that $T([1,3]) > T([2,2]) > T([4])$, and thus we know that for $d \leq 4$, we should not partition at all, proving the first part of the lemma.
Algorithm 7 Partitioning based algorithm for \((d, l)\)-CSP.

\[ P-(d, l)-CSP \quad (\Theta = (X, D, C), P = \{P_1, \ldots, P_m\}) \]

1. for each total function \(f : X \rightarrow P\) do
2. \(\Theta_f := (X, D, C \cup \{(x; f(x) \mid x \in X)\})\)
3. Solve \(\Theta_f\) using Lemma 10
4. if \(\Theta_f\) has a solution then
5. return “satisfiable”
6. end for
7. return “unsatisfiable”

For every other domain, we now note that \(T([1,4]) > T([2,3])\), so we will, in fact, never have a partition of size 1 in an optimal partitioning, since we can always do better with [2,3]. Additionally, since \(T([2,4]) > T([3,3])\) and \(T([2,3,3]) > T([4,4])\), we will never have a partitioning of size 2. Of course, we will need partitionings of size 3 in order to deal with domains which are not divisible by 4, but in general we have that \(T([3,3,3,3]) > T([4,4,4])\), so there will never be more than three partitions of size 3, and the lemma follows. \(\square\)

To summarise the proof of Lemma 16, we have the following inequalities:

1. \(T([1,1]) > T([2])\)
2. \(T([1,2]) > T([3])\)
3. \(T([1,3]) > T([2,2]) > T([4])\)
4. \(T([1,4]) > T([2,3])\)
5. \(T([2,4]) > T([3,3])\)
6. \(T([2,3,3]) > T([4,4])\)
7. \(T([3,3,3,3]) > T([4,4,4])\)
5.3. Partitioning Colouring Problems

Since the function $T$ is extendible, we also know that the partitionings to the left of the inequalities will always give worse performance than those on the right, and thus should be avoided, if possible. A general guideline when searching for an optimal partitioning is to make a list of the possible partition sizes, and then eliminate the poor choices by ordering them as we did above.

We are now ready to put it all together and get the following corollary to Theorem 15:

**Corollary 17.** Using the $O(1^{n_1+n_2}n_3\beta n_4)$ time algorithm for $(4, 2)$-CSP from [51], together with the optimal partitioning from Lemma 16, a partitioning based algorithm for $(d, 2)$-CSP will have the following running times:

1. $O((0.4518d)n)$ if $d \equiv 0 \pmod{4}$
2. $O((0.4518d + 0.0273)n)$ if $d \equiv 1 \pmod{4}$
3. $O((0.4518d + 0.0182)n)$ if $d \equiv 2 \pmod{4}$
4. $O((0.4518d + 0.0091)n)$ if $d \equiv 3 \pmod{4}$.

These running times are only slightly slower than the algorithm we get from the covering method (see Corollary 12), as well as Eppstein’s probabilistic algorithm.

5.3 Partitioning Colouring Problems

For colouring problems, i.e. CSPs where the only allowed constraint is disequality, we can improve the running time considerable. If we have a partitioning where $x \in P_1$ and $y \in P_2$, say, it always holds that $x \neq y$ since $P_1$ and $P_2$ are disjoint by definition. We will see later that there are a number of problems where we can exploit this and get partitioning based algorithms with impressive speeds.

Let $\Theta = (X, D, C)$ and $\Theta' = (X', D', C')$ be two CSPs with the property that given solutions $f$ to $\Theta$ and $f'$ to $\Theta'$, they can be combined to get a solution to $\Theta \cup = (X \cup X', D \cup D', C \cup C')$ (possibly
modulo renaming of the variables and domain values.) Conversely, the two subinstances $\Theta$ and $\Theta'$ correspond to a partitioning of $\Theta \cup \Theta'$; the partitioning is $|[D], |D'|]$, and the variables in $X$ are mapped to $D$, while those in $X'$ are mapped to $D'$.

We will let $\text{Col}(k, n)$ denote an arbitrary instance of a problem with domain size $k$ and $n$ variables having this property. It turns out that a number of problems can be categorised in this way, and we will see numerous examples of this later.

**Theorem 18.** Let $A_1, \ldots, A_m$ be algorithms for solving $\text{Col}(k_1, n)$, $\ldots, \text{Col}(k_m, n)$, respectively, with running times in $O(\alpha^n)$. Given a partitioning $P = \{P_1, \ldots, P_p\}$ of the set $\{1, \ldots, k\}$ such that for any partition $P_i$, $|P_i| \in \{k_1, k_2, \ldots, k_m\}$, i.e. we have an algorithm for solving instances with domain size $|P_i|$, there exists a partitioning based algorithm for solving $\text{Col}(k, n)$ which has a running time of

$$O((|P| - 1 + \alpha)^n).$$

**Proof.** Let $\text{Col}(k, n) = (X, D, C)$ be an instance as defined earlier, $P$ a partitioning of $D = \{1, \ldots k\}$, and let $F$ be the set of all total functions $f : X \to P$—i.e. every possible way of restricting the variables to partitions. Choose an arbitrary $f \in F$ and consider the set $\{x \in X \mid f(x) = P_i\}$ of variables. The problem restricted to these variables we can solve using algorithm $A_{|P_i|}$ and after solving two of these restricted problems, we can combine the solutions (by assumption) into a solution to a larger instance. By induction we can do this for all partitions, and if we repeat this for all $f \in F$, and choose the solution we are most satisfied with, we are guaranteed to find a solution to $\text{Col}(k, n)$, if one exists.

The time complexity of this is bounded from above by $O(\beta^n)$, where

$$\beta^n = \sum_{f \in F} \sum_{P_i \in P} a^{\{x \in X, f(x) = P_i\}}.$$ 

Now $|F|$ is the number of ways we can assign vertices to partitions. Given that $n_1$ vertices are assigned partition $P_1$, there are $\binom{n}{n_1}$ ways
5.3. Partitioning Colouring Problems

to choose which of the vertices goes to $P_1$. Similarly, we get \( \binom{n-n_1}{n_2} \) ways to select the vertices in partition $P_2$ given that we have already chosen $n_1$ vertices for $P_1$. Repeating this, we note that we have

\[
\binom{n - \sum_{j=1}^{i-1} n_j}{n_i}
\]

ways to choose the vertices for partition $P_i$, and, consequently, we get

\[
\binom{n}{n_1} \binom{n-n_1}{n_2} \cdots \binom{n - \sum_{j=1}^{p-1} n_j}{n_p}
\]

ways in all. This expression is, of course, the multinomial coefficient of the term $x_1^{n_1} x_2^{n_2} \cdots x_p^{n_p}$ in the polynomial \((x_1 + x_2 + \ldots + x_p)^n\), usually written as

\[
\binom{n}{n_1 \ldots n_p}
\]

Additionally, we note that there are no occurrences of $P_i$ in the terms, and get:

\[
|P| \sum_{n_1 + \ldots + n_p = n} \binom{n}{n_1, \ldots, n_p} \alpha^{n_i}
\]

which, using the multinomial theorem, becomes

\[
|P| \left( \frac{1 + 1 + \ldots + 1 + \alpha}{|P|-1} \right)^n \in O((|P| - 1 + \alpha)^n),
\]

and the result follows.

Thus if we want to use this theorem on a problem, we have to, first of all, show that the problem has the properties given earlier, and, second, find or construct algorithms for solving the restricted problems we get from the partitionings. As we mentioned in the introduction to this chapter, Lawler [91] observed that we can determine
2k-colourability of a graph by examining all possible partitionings of the vertices into two disjoint subsets and checking k-colourability of the resulting induced subgraphs. While we unaware of this result when we developed our method, our approach can be seen as a refinement and generalisation of this—and k-COLOURING fulfills the properties we have laid out.

We note that the running time given by Theorem 18 is largely dependent on the number of partitions and less so on the running times of the algorithms for the different partitions. Consequently, in order to minimise the running times, we want to use as few partitions as possible. If we have an algorithm for $Col(k, n)$, then we can of course get an algorithm for $Col(2k, n')$ by using the partitioning $[k, k]$. Consequently, the idea here is to use a recursive partitioning to build the $Col(k, n)$-algorithm bottom up; if we want an algorithm for $Col(4k, n)$, we first create an algorithm for domains of size $2k$ from a $Col(k, n)$ algorithm, together with the partitioning $[k, k]$—provided, of course, that we have an algorithm for $Col(k, n)$! If not, then we have to construct one by using the partitioning $[[k/2], [k/2]]$, etc. Whether this partitioning is optimal, however, is still an open question.

In general, if we have algorithms for solving instances with domain sizes $k_1, \ldots, k_m$, with running times $O\left(\beta_{k_i}^n\right)$, $i \in \{1, \ldots, m\}$, and it is faster to use the available algorithm for size $k_i$ than using the partitioning $[[k_i/2], [k_i/2]]$, i.e. $T([k_i]) < T([[k_i/2], [k_i/2]])$, then there exists a partitioning based algorithm for solving for domain size $k$ which has a running time of $O\left(\alpha_k^n\right)$, where $\alpha_k$ is the solution to the following recurrence:

$$\alpha_k = \begin{cases} 
\beta_k & \text{if } k \in \{k_1, \ldots, k_m\} \\
1 + \alpha_{[k/2]} & \text{otherwise.}
\end{cases}$$

Solving this equation is straightforward, albeit tedious, thus we will omit this part of the proofs, but one way of solving these equations is demonstrated here.

The following recurrence is the result of applying the recursive partitioning scheme to the problem of counting $k$-colourings of a graph.
For this particular problem, we know that we can do this in polynomial time for $k = 2$, and, for $k = 3$, we have a specialised algorithm which runs in $\mathcal{O}(\beta^n)$, $\beta = 1.7880$ (see Section 8.2 for details.) Thus we get:

$$
\alpha_k = \begin{cases} 
1 & \text{if } k = 2 \\
\beta & \text{if } k = 3 \\
1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise.}
\end{cases}
$$

First, let $k = 2^i$. By iterating the recursion, we get

$$
\alpha_k = 1 + \alpha_{k/2} = 2 + \alpha_{k/4} = \ldots = (i - 1) + \alpha_{k/2^{i-1}} = (i - 1) + \alpha_2 = i = \log_2 k.
$$

Now, let $2^i < k \leq 2^i + 2^{i-1}$. For $i = 2$, this amounts to $4 < k \leq 6$, and $\lceil k/2 \rceil = 3$. Since $\alpha_3 = \beta$ by definition, $\alpha_k = \beta + 1 = [\log_2 k] + \beta - 1$.

For $i > 2$ we have

$$
\alpha_{\lceil k/2 \rceil} = [\log_2 k] + \beta - 1 = [\log_2 k] + \beta - 2,
$$

so

$$
\alpha_k = 1 + \alpha_{\lceil k/2 \rceil} = 1 + [\log_2 k] + \beta - 2 = [\log_2 k] + \beta - 1.
$$

Finally, we have $2^i + 2^{i-1} < k < 2^{i+1}$. For $i = 2$, $k = 7$, and $\lceil 7/2 \rceil = 4$, which gives $\alpha_7 = \alpha_4 + 1 = 2 + 1 = [\log_2 7] + 1$. If $2^i + 2^{i-1} < k < 2^{i+1}$, we have $2^{i-1} + 2^{i-1} < k/2 < 2^i$, and $\alpha_{\lceil k/2 \rceil} = [\log_2 k/2] + 1 = [\log_2 k]$ if $\lceil k/2 \rceil < 2^i$. From earlier, we know that if $\lceil k/2 \rceil = 2^i$, then $\alpha_{\lceil k/2 \rceil} = [k]$. Consequently, $\alpha_k = 1 + \alpha_{\lceil k/2 \rceil} = [\log_2 k] + 1$.

The end result is thus that

$$
\alpha_k = \begin{cases} 
i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \\
i - 1 + \beta & \text{if } 2^i < k \leq 2^i + 2^{i-1}
\end{cases}
$$

for some $i \geq 4$. 
Chapter 6

Future Work

While heading the profit of my counsel, avail yourself also of any helpful circumstances over and beyond the ordinary rules.

According as circumstances are favorable, one should modify one’s plans.

Sun Tzu, The Art of War

It goes without saying that this thesis is not the final word on the topics we discuss. There are numerous obvious roads we can take from here, and most likely (even hopefully) a number of not-so-obvious ones that other people will find some day.

Perhaps the most obvious of the open questions that we have not addressed is the implementation of our methods. While this was never one of our goals — our focus has on providing a solid theoretical foundation — it would be interesting to see how well the methods would work in practice.

Coverings The covering theorem, Theorem 9, does not give us an explicit way of determining which coverings to use, and if we want something more tangible, we have to find it in some other way. The problem, of course, lies with the probabilistic proof of the theorem, thus finding a constructive version is still an open question.
Partitionings  For the partitioning method, the most pressing matter is a way, preferably given as a formula, to find the optimal size of the partitionings to use. The “forbidden sub-partitioning” approach which we outline is a first step, but it remains to be seen where this will lead.

If we look at the application of the partitioning method to colouring problems, we find another open question which needs investigating. We use the recursive partitioning $\lceil k/2 \rceil, \lfloor k/2 \rfloor$ in several of our algorithms, but we have not yet determined if this is an optimal partitioning.

There is another important question which arises from the application of the partitioning method to colouring problems. Restricting the allowed constraints to be disequality, we can reason about the partitions in isolation, and thereby get a significant improvement in running times. It does not seem too far-fetched to assume that this same idea could be applied to other constraints. Consider, for example, if we only allow the constraint less than ($<$), and have as domain the integers $\{1, 2, 3, 4, 5, 6, 7\}$. Partitioning the domain into disjoint subsets $P_1 = \{1, 2, 3, 4\}$ and $P_2 = \{5, 6, 7\}$, we observe that if variable $x \in P_1$ and variable $y \in P_2$, then $x < y$ trivially holds, and $y < x$ is trivially false. We have not yet studied restrictions of this kind, but, at least superficially, it appears to be a viable approach. As an added incentive to this, there exist a number of interesting results for subclasses of CSPs with this kind of restrictions, see, e.g., Jeavons & Cooper [78].
Part III

Applications
Chapter 7

Decision Problems

The quality of decision is like the well-timed swoop of a falcon which enables it to strike and destroy its victim. Therefore the good fighter will be terrible in his onset, and prompt in his decision.

Sun Tzu, The Art of War

This chapter is somewhat brief, since we have already discussed algorithms for \((d, l)\)-CSP (and \((d, 2)\)-CSP), and how to apply the partitioning and covering methods to them. (See Sections 4.3 and 5.2, respectively.)

Chapter Outline

We begin in Section 7.1, by constructing a partitioning based algorithm for the \(k\)-COLOURING problem for arbitrary graphs, and then, in Section 7.2, we discuss the application of coverings in quantum and molecular computing.

7.1 \(k\)-Colouring Algorithm

We will now show how the partitioning method applies to the problem of finding a \(k\)-colouring of a graph. While we do not improve on the running time of the best known algorithms for this problem, we do
manage to achieve the same bounds for the 6-COLOURING problem as [32] (which holds the current record.) It should be noted, however, that the algorithm we give in Theorem 21, which works for any \( k \), runs in polynomial space, whereas the general CHROMATIC NUMBER algorithm, though it is faster, requires exponential space. Furthermore, the bounds given by Theorem 21 is a drastic improvement over the currently best known polynomial-space \( k \)-colouring algorithm, the randomized \( \mathcal{O}((k/c_k)^n) \) time algorithm presented by Feder & Motwani [54] which we discussed in Section 2.3.1.

Formally, we define the problem as follows:

**Definition 19.** Let \( G \) be an arbitrary graph with vertex set \( V(G) \) and edge set \( E(G) \). Given a natural number \( k \), the \( k \)-COLOURING problem consists of finding a function \( f : V(G) \to \{1, \ldots, k\} \) which assigns ‘colours’ to the vertices in such a way that for \( v, w \in V(G) \), \( f(v) = f(w) \) if \( \{v, w\} \notin E(G) \), i.e. adjacent vertices are given different colours.

Before we can apply Theorem 18, we need to show that the \( k \)-colouring problem is a \( \text{Col}(k, n) \) problem. To see this, we note that if we partition the vertices of a graph \( G \) into two disjoint subsets, \( S_1, S_2 \), and colour the subgraphs induced by these sets using colours \( \{1, \ldots, k_1\} \) for \( S_1 \) and \( \{k_1 + 1, \ldots, k_2\} \) for \( S_2 \), getting solutions \( f_1 \) and \( f_2 \), respectively, then \( f_1 \cup f_2 \) must be a valid \( k_2 \)-colouring of \( G \).

Thus we get the following theorem:

**Theorem 20.** Let \( P = \{P_1, P_2, \ldots, P_m\} \) be a partitioning of \( \{1, \ldots, k\} \) such that for any \( P_i \in P \) there exists an algorithm for solving the \( |P_i| \)-COLOURING problem in \( \mathcal{O}(\alpha^n) \) time. Then, there exists a partitioning based algorithm for solving the \( k \)-COLOURING problem in time

\[
\mathcal{O}((|P| - 1 + \alpha)^n).
\]

**Proof.** First, we show that the algorithm in Fig. 8 correctly solves \( k \)-COLOURING. Let \( G \) be an instance of \( k \)-COLOURING, \( P \) a partitioning the domain values and \( f \) an arbitrary total function from \( V(G) \) to \( P \)—i.e. a function which assigns the vertices to partitions. Lines 3
Algorithm 8 Partitioning based $k$-COLOURING algorithm.

$P$-$k$-$COL$ $(G, P)$

1. for each total function $f : V(G) \rightarrow P$ do
2. $a := \text{true}$
3. for each $P_i \in P$ do
4. $G' := G|\{v \in V(G) \mid f(v) = P_i\}$
5. $a := a \land A_i|_{P_i}(G')$
6. end for
7. if $a$ then
8. return “yes”
9. end for
10. return “no”

The vertices restricted to partition $P_i$ induces a subgraph, and we can determine $|P_i|$-colourability of this subgraph in $O(\alpha^n)$ time. Obviously, if we have two of these induced subgraphs, and we know that we can colour these two using, say, $k_1$ and $k_2$ colours, respectively, then we can colour the union of them using $k_1 + k_2$ colours. So, by induction, the variable $a$ will, once all of the induced subgraphs have been examined, contain true iff the graph is $k$-colourable. Repeating this for all total functions ensures that we will examine all possible restrictions of vertices to partitions.

The running time of the algorithm is given by Theorem 18, which we have shown applies to this problem.

While Algorithm 8 only determines the existence of a $k$-colouring, it is of course straightforward to adapt it to return an explicit colouring; assuming the algorithm $A_i$ used on Line 5 returns an $i$-colouring, make a note of this colouring and on Line 6, rename the colours (in order to avoid clashes) and return the union the partial solutions.

In the literature, we find a number of different algorithms for determining $k$-colourability of a graph. Among these, the following constitute the ‘state-of-the-art’ in (polyspace) algorithms at the time of this writing:
7.1. $k$-Colouring Algorithm

- 3-Colouring in $O(1.3289^n)$ [51],
- 4-Colouring in $O(1.7504^n)$ [32],
- 5-Colouring in $O(2.1020^n)$ [34] (See also [33])
- 6-Colouring in $O(2.3289^n)$ [32]

For $k > 6$, the most efficient polynomial space algorithm for $k$-colouring is the $O((k/c_k^n)$ time algorithm by Feder & Motwani [54] (See Section 2.3.1.) We will not get a improvement over the bounds for $k \leq 6$ from the partitioning method, but what we will get is a way of constructing algorithms for any $k$, which is faster than $O((k/c_k^n)$.

Let $\beta^n_i, i \in \{2, 3, 4, 5\}$ denote the running times of the 3-, 4-, 5-colouring algorithms above (we do not need the 6-colouring algorithm — but more on that later) except for 2-colourability, which we can decide in polynomial time. (Hence $\beta_2 = 1$ since we omit polynomial factors.)

As can be seen in Theorem 20, the number of partitions has a large impact on the running time of the algorithm. For example, if we want an algorithm for, say, 8-colouring, it is tempting to use the partitioning [2, 2, 2, 2], since 2-colouring is polynomial. This, however, gives a running time of $O(4^n)$, while if we use the partitioning [4, 4], we get a running time of $O(((2i - 1 + 1.7504)^n) = O(2.7504^n)$, which is an enormous improvement. Using the partitioning [3, 3] we get a 6-colouring algorithm with the same running time as in [32], i.e. $O((2i - 1 + 1.3289)^n) = O(2.3289^n)$.

**Theorem 21.** If we can solve 3-, 4-, 5-Colouring in $O(\beta^n_i)$ time, for $i = 3, 4, 5$, respectively, then there exists a partitioning based algorithm for solving $k$-Colouring, $k > 6$, which has a running time of $O(\alpha^n_k)$, where

$$
\alpha_k = \begin{cases} 
  i - 2 + \beta_5 & \text{if } 2^i < k \leq 2^i + 2^{i-2} \\
  i - 1 + \beta_3 & \text{if } 2^i + 2^{i-2} < k \leq 2^i + 2^{i-1} \\
  i - 1 + \beta_4 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} 
\end{cases}
$$

for $i \geq 3$. 

Proof. Using the partitioning $\lfloor k/2 \rfloor, \lceil k/2 \rceil$ recursively, together with
the colouring algorithms above, we get from Theorem 20 that a partitioning based algorithm will have a running time of $\mathcal{O}(\alpha_k^n)$, where $\alpha_k$ is given by the solution to the following recurrence:

$$
\alpha_k = \begin{cases} 
\beta_k & \text{if } k \in \{3, 4, 5\} \\
1 + \alpha_{\lfloor k/2 \rfloor} & \text{otherwise}
\end{cases}
$$

Solving the equation gives the result.

\[\]
Quantum Computation. It is probably easiest to think of a quantum algorithm as a probabilistic algorithm where we know the possible states the algorithm can be in, and we know the initial configuration, but we cannot observe any of the random guesses the algorithm makes. This means that at any given time, the algorithm can be in a range of states, called a linear superposition, each having a certain probability, or amplitude. Of course, should we decide to check which state the algorithm is currently in, it will immediately choose one out of the many possible ones, with a probability equal to the amplitude.

The details of how this works are rather complicated, but a good introduction to the field of quantum computing and quantum complexity theory is given in [27] (although for the less technically minded of us, the conference version of the paper uses a lighter terminology [26].) For the interested reader, it should be pointed out that it is still not known whether quantum computation will have any impact on the $P \neq NP$ question. What is known, is that there are problems which we can solve more efficiently using quantum computation than is possible in the classical setting, e.g. Grover’s $O(\sqrt{n})$ time quantum search algorithm [70].

Definition 23 (Bernstein & Vazinari [27]). $\text{BQTime}(T(n))$ is defined as the class of problems solvable in $O(T(n))$ time on a quantum computer, with a probability of error bounded by $2/3$.

Molecular Computing. In molecular computing, the measure of complexity is not time, as is normally the case. Rather, the focus is on the size of the solution space, given as the number of strands used in the computation, often denoted the volume. The operations are usually defined as working on test tubes, which can be viewed as a multiset of strings over some alphabet. Starting with an initial test tube, $t_0$, which is initialised to hold encodings of the search space, a sequence of operations is then performed, and the computation accepts if $t_0$ is non-empty after the last of these. The allowed operations differ somewhat in the literature, but common ones, all of which are described further in [22], include
– *Separate*($t, c, i, t_1, t_2$), which separates the strings in $t$ into test tubes $t_1$ and $t_2$, depending on whether the character at position $i$ in the string is $c$ or not,

– *Append*($t, c$), which appends the character $c$ to every string in $t$,

– *Merge*($t_1, t_2, t$), which simply merges the contents of tubes $t_1$ and $t_2$ into $t$, and finally

– *Detect*($t$), which tests if there is at least one string left in the test tube. This operation is necessary for finding the result of a computation.

**Definition 24 (Beigel & Fu [22]).** MOL($T(n)$) is the class of problems solvable in polynomial time on a molecular computer using $T(n)$ volume.

The following theorem provides a link between the three classes of problems above, which allows us to dispense with the details.

**Theorem 25.** If *(d,l)*-CSP is in NPinit($\alpha^n$), then

(i) *(d,l)*-CSP is in MOL($\alpha^n$), and

(ii) *(d,l)*-CSP is in BQTime($\alpha^{n/2}$).

**Proof.** (i) Beigel & Fu [22] has shown that NPinit($T(n)$) $\subseteq$ MOL($T(n)$).

(ii) The non-deterministic choice can be made by Grover’s quantum search algorithm [70] in $O(\alpha^{n/2})$ time. □

We can immediately use this to improve an old result on quantum 3-COLOURABILITY. In [8] we find a quantum computing algorithm for determining 3-colourability of a graph which has a running time of $O(1.2185^n)$, but using Theorem 25 we get:

**Corollary 26.** Using quantum computation, 3-COLOURABILITY can be decided in $O(1.3447^{n/2}) \approx O(1.1597^n)$ time.
Proof. 3-Colourability belongs to $\text{NPinit}(1.345^n)$ [22] hence the result follows from Theorem 25 (ii).

Combining Definitions 22, 23 and 24 with Theorem 9, we get the following theorem:

**Theorem 27.** If $(e, l)$-CSP is in $\text{NPinit}(\alpha^n)$, then, for any $d > e$, there exist molecular and quantum algorithms for solving $(d, l)$-CSP in

- $\mathcal{O} \left((d/e + \varepsilon)^n \cdot \alpha^n\right)$ volume, and
- $\mathcal{O} \left((d/e + \varepsilon)^{n/2} \cdot \alpha^{n/2}\right)$ time,

respectively.

Proof. First we show that if $(e, l)$-CSP is in $\text{NPinit}(\alpha^n)$, then for all $d > e$, $(d, l)$-CSP is in $\text{NPinit}((d/e + \varepsilon)^n \cdot \alpha^n)$. The proof is similar to that of Theorem 11, but instead of giving an algorithm, we observe that each instance $\Theta_f$ can be solved by guessing a number between 1 and $\alpha^n$, and since there are at most $(d/e + \varepsilon)^n$ such instances, according to Theorem 9, we can solve the problem by guessing a number between 1 and $(d/e + \varepsilon)^n \cdot \alpha^n$, and thus $(d, l)$-CSP is in $\text{NPinit}((d/e + \varepsilon)^n \cdot \alpha^n)$.

This, together with Theorem 25 gives that $(d, l)$-CSP can be solved using $\mathcal{O} \left((d/e + \varepsilon)^n \cdot \alpha^n\right)$ volume, with molecular computation, and, with quantum computation, in $\mathcal{O} \left((d/e + \varepsilon)^{n/2} \cdot \alpha^{n/2}\right)$ time.

By using concrete algorithms with bounded nondeterminism, we get the following corollary.

**Corollary 28.** There exists

1. $\mathcal{O} \left((d/3 + \varepsilon)^n \cdot 1.3803^n\right)$-volume molecular computation, and
2. $\mathcal{O} \left((d/3 + \varepsilon)^{n/2} \cdot 1.3803^{n/2}\right)$ time quantum algorithms for solving $(d, 2)$-CSPs.

Proof. Beigel & Fu [22] show $(3, 2)$-CSP to be in $\text{NPinit}(1.3803^n)$ and Theorem 27 implies the result.
Chapter 8

Counting Problems

In battle, there are not more than two methods of attack — the direct and the indirect; yet these two in combination give rise to an endless series of maneuvers. The direct and the indirect lead on to each other in turn. It is like moving in a circle — you never come to an end. Who can exhaust the possibilities of their combination?

Sun Tzu, The Art of War

We will now look at problems where we want to find the number of solutions, rather than a single one, which was the case in the previous chapter.

Chapter Outline

We begin this chapter with Section 8.1 where we construct a partitioning based algorithm for $(d, 2)$-CSPs, and then, in Section 8.2, we construct a specialised algorithm for the $\#k$-COLOURING problem, which as a part will utilise the $\#3$-colouring algorithm given in Section 8.3.
8.1 $\#(d, 2)$-CSP Algorithm

The algorithm we will now construct for $\#(d, 2)$-CSPs is based on partitionings, and was first presented in [9]. First, we need the counting version of Lemma 10:

**Lemma 29.** Assume there exists an $O(\alpha^n)$ time algorithm for solving $\#(e, l)$-CSP. Let $I_e$ denote the set of $\#(d, l)$-CSP instances satisfying the following restriction: For every $(X, D, C) \in I_e$, and every $x \in X$, there exists a constraint $(x; S)$ in $C$ such that $|S| \leq e$. Then, the $\#CSP$ problem restricted to instances in $I_e$ can be solved in $O(\alpha^n)$ time.

*Proof.* Almost identical to that of Lemma 10.

The following theorem is similarly the counting equivalent version of Theorem 15:

**Theorem 30.** Let $A$ be an algorithm for $\#(e, 2)$-CSP with a running time of $O(\prod_{i=1}^{e} \alpha_i^n)$, $\alpha_i \geq 1$, when applied to an instance containing $n_i$ $i$-valued variables, for $1 \leq i \leq e$. Choose $d$ and a partitioning $P = \{P_1, \ldots, P_k\}$ of $\{1, \ldots, d\}$ such that $|P_i| \leq e$ for all $i$. Then, there exists an algorithm for $\#(d, 2)$-CSP running in $O((\sum_{i=1}^{e} \sigma(P, i)\alpha_i)^n)$ time.

*Proof.* We claim that Algorithm 9 correctly solves the $\#(d, 2)$-CSP problem.

Let $\Theta = (X, D, C)$ be an arbitrary instance of $\#(d, 2)$-CSP, and let $M$ be an arbitrary model of $\Theta$. The members of $P$ are by definition pairwise disjoint, so there exists exactly one function from $X$ to $P$ such that $M(x) \in f(x)$ for all $x \in X$. Consequently, Lemma 29 guarantees that the variable $c$ in the algorithm will contain the number of models after completion of the algorithm, and the algorithm will return the correct result.

The running time of the algorithm follows from Theorem 15.

The following theorem is similarly the counting equivalent version of Theorem 15:
Algorithm 9 Partitioning based \((d,2)\)-CSP algorithm.

\begin{algorithm}
\caption{\textsf{\#}(d,2)-CSP \((\Theta = (X,D,C), P)\)}
\begin{algorithmic}
\STATE \textbf{1.} \(c := 0\)
\STATE \textbf{2.} \textbf{for each} total function \(f : V \rightarrow P\) \textbf{do}
\STATE \quad \(\Theta_f := (X,D, C \cup \{(x_i, f(x_i)) \mid 1 \leq i \leq |X|\})\)
\STATE \quad Compute the number of solutions \(c_f\) to \(\Theta_f\) using Lemma 29
\STATE \quad \(c := c + c_f\)
\STATE \textbf{end for}
\STATE \textbf{3.} \textbf{return} \(c\)
\end{algorithmic}
\end{algorithm}

as the 'base case' for our algorithm. The transformation from \((d,2)\)-CSP to \#2-SAT is somewhat similar to the one we used in the example in Section 4.1, but there are some subtle and not-so-subtle differences.

Given a \((d,2)\)-CSP instance \(\Theta = (X,D,C)\) containing \(k_1\) 1-valued, \(k_2\) 2-valued, \(k_3\) 3-valued, etc., variables, we can transform it into a weighted \#2-SAT instance as follows: If a variable \(x \in X\) takes its value from a one-valued set, we can remove \(x\) in polynomial time by assigning it the only possible value it an assume, and propagate the variable.

If \(x\) takes its value from a two-valued domain \([d_1,d_2]\), we introduce a proposition \(x[d_1]\) with the interpretation that \(x = d_1\) if \(x[d_1]\) is true, and \(x = d_2\) otherwise. For a \(k\)-valued variable, we create \(k\) propositional variables \(x[1], \ldots, x[k]\), with the interpretations that \(x = i\) if \(x[i]\) is true, and \(x \neq i\) if \(x[i]\) is false. To ensure that at most one of these can be true in a satisfying assignment, we add the clauses

\[
\bigwedge_{i,j \in \{1,\ldots,k\}, i \leq j} (\neg x[i] \lor \neg x[j]).
\]

Constraints involving only variables with more than 2 possible values can be transformed as follows: Given the constraint \(R(x, y) \in C\), with
$x$ and $y$ having domains $D_x$ and $D_y$, respectively, we add the clauses

$$\bigwedge_{a \in D_x, b \in D_y} (\neg x[a] \lor \neg y[b]).$$

However, if one of the variables has two values, we need to take into account that its negation also corresponds to an assignment, thus, if $R(x, y) \in C$, $x \in \{1, 2\}$ and $y \in D_y$, we add the clauses

$$\bigwedge_{b \in D_y} (\neg x[1] \lor \neg y[b]) \land \bigwedge_{b \in D_y} (x[0] \lor \neg y[b]).$$

The case when both variables in the constraint are 2-valued can be transformed analogously. Now we note that there is a slight difference between the two kinds of propositions: A proposition stemming from a 2-valued variable will, regardless of truth value, have an interpretation which assigns a value to the original variable, but this does not hold for variables with domains larger than 2. To remedy this, we exploit the possibility of adding weights to propositions: We give weight 0 to each proposition corresponding to a 2-valued variable, and the remaining propositions are assigned weights 1, thus a model of the instance we have just created will have the weight $\sum_{i \geq 3} k_i$. In this way, we force the true variables in the model to have an interpretation that assigns a value to each of the variables in the original problem.

After this transformation, we get a 2-SAT formula with $k_2 + 3k_3 + 4k_4 + \ldots$ variables, and if we apply a weighted #2-SAT solver to this formula, we get a running time of

$$O \left( 1^{k_1} \cdot \alpha^{k_2} \cdot \alpha^{3k_3} \cdots \right),$$

since each 2-valued variable introduces one propositional variable, and each $k$-valued, $k > 2$, variable introduces $k$ propositional variables.

Given a partitioning $P$ of a domain $D$ containing $d$ elements, if we combine the algorithm above with Theorem 30, it follows that we can solve $\#(d, 2)$-CSP in $O(T(P))$ time, where

$$T(P) = \left( \sigma(P, 1) + \sigma(P, 2) \alpha + \sum_{i=3}^{d} \sigma(P, i) \alpha^i \right)^n.$$  \hfill (8.1)
Using, say, the trivial partitioning $P = [d]$, we can solve the problem in $O(n^d)$ time, but this can be improved by using a non-trivial partitioning, and the question is now how to determine the optimal partitioning.

First, let the multiset $P = [p_1, \ldots, p_k]$ represent an arbitrary partitioning of $D$ and note that the function $T$ is extendible (see Definition 14.) In Lemma 16, we had at our disposal a polynomial time algorithm for $(2,2)$-CSP and the $(3,2)$- and $(4,2)$-CSP algorithms mentioned previously [51]. Among these, the $(4,2)$-CSP algorithm has the best ‘performance per domain value,’ running in $O((0.4518d)^n)$, so partitions of size 4 were preferred over size 2, even though the $(2,2)$-CSP algorithm is polynomial. Partitions of size 3 are somewhere in between, being better than those of size 2, but worse than those of size 4, but they are needed in order to handle odd sized domains.

For the counting problem, we no longer have a polynomial algorithm for $d = 2$, and we use a $(2,2)$-CSP algorithm (i.e. a #2SAT algorithm) to handle this case. From Theorem 30 we know how to combine algorithms to get new ones, and, as was the case in Theorem 16, it turns out that we do not want to use partitions of size 1. Using the weighted #2SAT algorithm by Dahllöf et al. [44], which has a running time of $O(1.2561^n)$, or $O((0.6281d)^n)$, it turns out that, this time, 3-partitions are also to be avoided, since we get better performance by considering partitions of size 5. This stems from the fact that the algorithm for 3-partitions is constructed by a combination of algorithms for domain sizes 1 and 2, while the 5-partition comes from algorithms for domain sizes 1 and 4, and #$(4,2)$-CSP gives better performance.

**Lemma 31.** Given an $O(1.2561^n)$ time algorithm for solving weighted #2-SAT, let $D$ be a domain of size $d \geq 2$. If $d \leq 5$, then the partitioning $P = [d]$ is optimal with respect to the running time of the resulting partitioning based algorithm for #$(d,2)$-CSP. Otherwise, the following partitionings are optimal:

- if $d \equiv 0 \pmod{4}$, $P = [4,4,4,\ldots,4]$
- if $d \equiv 1 \pmod{4}$, $P = [5,4,4,\ldots,4]$
8.1. \( \#(d, 2) \)-CSP Algorithm

- if \( d \equiv 2 \pmod{4} \), \( P = [2, 4, 4, \ldots, 4] \)
- if \( d \equiv 3 \pmod{4} \), \( P = [2, 5, 4, 4, \ldots, 4] \)

**Proof.** Using Equation 8.1 to get the running times for partitions of sizes 1 through 5, we can observe the following inequalities:

- \( T([1, 1]) > T([2]) \),
- \( T([1, 2]) > T([3]) \)
- \( T([1, 3]) > T([2, 2]) > T([4]) \)
- \( T([1, 4]) > T([2, 3]) > T([5]) \)
- \( T([3, 3]) > T([2, 4]) \)
- \( T([3, 4]) > T([2, 5]) \)
- \( T([3, 4]) > T([4, 4]), \) and
- \( T([5, 5, 5, 5]) > T([4, 4, 4, 4, 4]). \)

Similar to Lemma 16, we have to draw the conclusion that the partitioning is optimal. \( \square \)

Summing it all up, we get the following corollary:

**Corollary 32.** There exists a partitioning based algorithm for solving \( \#(d, 2) \)-CSP with a running time of

- \( \mathcal{O} \left( (\alpha^4 d/4)^n \right) \subseteq \mathcal{O} \left( (0.6224 d)^n \right) \) if \( d \equiv 0 \pmod{4} \)
- \( \mathcal{O} \left( (\alpha^4 [d/4 - 1] + \alpha^5)^n \right) \subseteq \mathcal{O} \left( (0.6254 d)^n \right) \) if \( d \equiv 1 \pmod{4} \)
- \( \mathcal{O} \left( (\alpha + \alpha^4 [d/4])^n \right) \subseteq \mathcal{O} \left( (0.6243 d)^n \right) \) if \( d \equiv 2 \pmod{4} \)
- \( \mathcal{O} \left( (\alpha + \alpha^4 [d/4 - 1] + \alpha^5)^n \right) \subseteq \mathcal{O} \left( (0.6262 d)^n \right) \) if \( d \equiv 3 \pmod{4} \).
8. #k-Colouring Algorithm

We will now look at the #k-COLOURING problem. In the discussion we will be using graph theoretic rather than CSP notation, since this allows for the use of familiar concepts such as induced subgraph and independent set. (For simplicity, we will often write “COL” rather than COLOURING in the discussion.)

The following theorem is by necessity similar to Theorem 20, but with important differences. We are no longer interested in just the existence of a colouring, but are aiming at determining how many there are, thus we have to devote some additional effort in the proof.

**Theorem 33.** Let $P = \{P_1, P_2, \ldots, P_m\}$ be a partitioning of $\{1, \ldots, k\}$ such that for any $P_i \in P$ there exists an algorithm for solving the $\#|P_i|$-COLOURING problem in $O(\alpha^n)$ time. Then, there exists a partitioning based algorithm for solving the $\#k$-COLOURING problem in time

$$O \left( (|P| - 1 + \alpha)^n \right).$$

**Proof.** The proof is very similar to that of Theorem 20, and consists in showing that Algorithm 10 solves the problem, and that Theorem 18 applies.

Unlike the case with $\#(d, 2)$-CSP in the previous section, we now once again have a case we can solve in polynomial time; $\#2$-COL can easily be determined since the number of 2-colourings of a 2-colourable graph is simply $2^c$, where $c$ is the number of connected components of the graph. This fact, together with the algorithm for counting 3-colourings from [9], presented in the next section, which has a running time of $O(1.7880^n)$, allows us to use Theorem 33 to construct partitioning based algorithms for any $\#k$-colouring problem.

However, it is still the case that we have to figure out what partitioning to use in order to minimise the time complexity of the algorithm. As we saw in Theorem 18, the number of partitions has a large impact on the running time, and, using the same reasoning as in Section 5.3, we get the following theorem:
Algorithm 10 Partitioning based \#\text{k}-COLOURING algorithm.

\textbf{P- \#\text{k}-COL (G)}

1. \( c := 0 \)
2. \textbf{for each} total function \( f : V(G) \rightarrow P \) \textbf{do}
3. \( c' := 1 \)
4. \textbf{for each} \( P_i \in P \) \textbf{do}
5. \( G' := G[V \{ v \in V(G) \mid f(v) = P_i \}] \)
6. \( c' := c' \cdot \#P_i \)-COL\((G')\)
7. \textbf{end for}
8. \( c := c + c' \)
9. \textbf{end for}
10. \textbf{return} \( c \)

\textbf{Theorem 34.} If we can solve \#2-COLOURING in polynomial time and \#3-COLOURING in \( \mathcal{O}(\beta^n) \) time, then there exists a partitioning based algorithm for solving the \#\text{k}-COLOURING problem which runs in \( \mathcal{O}(\alpha_k^n) \) time, where

\[
\alpha_k = \begin{cases} 
  i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \\
  i - 1 + \beta & \text{if } 2^i < k \leq 2^i + 2^{i-1} 
\end{cases}
\]

and \( i \geq 4 \).

\textbf{Proof.} We will use the partitioning \( [\lfloor k/2 \rfloor, \lceil k/2 \rceil] \) recursively, with the algorithms \#2-COL and \#3-COL as base cases. By Theorem 33, the resulting algorithm runs in \( \mathcal{O}(\alpha_k^n) \) time, where \( \alpha_k \) is given by the recursion

\[
\alpha_k = \begin{cases} 
  1 & \text{if } k = 2 \\
  \beta & \text{if } k = 3 \\
  1 + \alpha_{\lfloor k/2 \rfloor} & \text{otherwise.}
\end{cases}
\]

Solving the recurrence equation gives the result. \( \square \)

In the following section, we will present an algorithm for the \#3-COLOURING problem which gives \( \beta \approx 1.7880 \) in the previous theorem.
8.3 \#3-Colouring Algorithm

We will now construct an $O(1.7880^n)$ time algorithm for counting the number of 3-colourings in a graph which can be used in Theorem 34 from the previous section. The three colours will be denoted $R$, $G$, and $B$ — and in order to avoid confusion, we use $H$ to denote a graph in this section. Thus, let $H$ be a graph. To each vertex, we associate a variable to hold its colour, and we let $H[x := X]$ denote the graph $H$ with the colour of vertex $x$ changed to $X$.

We define an $\{R, GB\}$ assignment of a graph $H$ as a total function $f : V(H) \rightarrow \{R, GB\}$. We say that an $R\{G/B\}$ assignment $f$ is refineable to a 3-colouring of $H$ iff for each of the vertices $v$ having colour $GB$, we can assign $v := G$ or $v := B$ in such a manner that we obtain a 3-colouring of $H$. We note that having an $\{R, GB\}$ assignment for $H$ which is refineable to a 3-colouring of $H$, is equivalent to the assignment having the following properties:

**P1.** the vertices with colour $R$ form an independent set;

**P2.** the induced subgraph of vertices with colour $GB$ is 2-colourable.

Obviously, these conditions can be checked in polynomial time. We can also find the number of possible refinements of an $\{R, GB\}$ assignment: Consider the graph $H' = H\{|v \in V(H) \mid f(v) = GB\}$ and note that the number of refinements equals $2^c$ where $c$ is the number of connected components in $H'$. Given an $\{R, GB\}$ assignment $f$, let $Count_2(H, f)$ denote this number (which is easily computable in polynomial time).

**Theorem 35.** There exists an algorithm for counting the number of 3-colourings of an arbitrary graph in $O(1.7880^n)$ time.

**Proof.** Let $\varphi = (1 + \sqrt{5})/2$ and let $C \approx 0.4711$ be the unique real positive root of the equation $\varphi^{1-C} \cdot 2^C = 3^{1-C}$. Begin by identifying an independent set $I$ in $H$ of maximum size using, for instance, Beigel’s [21] algorithm. If $|I| \leq C \cdot |H|$, then apply Algorithm 11. Otherwise, apply Algorithm 12.
Algorithm 11 Algorithm #3C₁.

#3C₁

1. if all vertices are \{R, GB\}-coloured then
   2. return \(\text{Count}_2(H)\)
3. elsif there exists an uncoloured vertex \(x\) with
   an uncoloured neighbour \(y\) then
   4. return \(#3C₁(H[x := R, y := GB]) + #3C₁(H[x := GB])\)
5. else cycle through all \{R, GB\} assignments of the
   uncoloured vertices, apply \(\text{Count}_2\) on each
   graph and return the total number of 3-colourings.
6. endif

To see that algorithm #3C₁ is correct, we note that the algorithm considers all \{R, GB\}-assignments that can be refined to a 3-colouring. In line 3, an uncoloured vertex \(x\) with an uncoloured neighbour \(y\) is chosen. In the first recursive branch, \(x\) is assigned the colour \(R\) which implies that \(y\) must be coloured \(GB\). In the other branch, \(x\) is coloured \(GB\) and this choice does not restrict the possible colourings of \(y\). In line 4, the algorithm exhaustively considers all \{R, GB\}-assignments of the uncoloured vertices.

The correctness of #3C₂ is shown by the following: For each 3-colouring \(f\) of \(H - I\), we claim that

\[
\prod_{v \in I} (3 - |\{ f(w) \mid w \in N(v) \}|)
\]

is the number of ways \(f\) can be extended to a 3-colouring of \(H\). Assume for instance that \(v \in I\) has three neighbours \(x, y, z\) that are coloured with \(R, G\) and \(B\), respectively. Then, \(3 - |\{ f(w) \mid w \in N(v) \}|\)
equals 0 which is correct since \(f\) cannot be extended in this case. It is easy to realise that the expression gives the right number of possible colours in all other cases, too. Since \(I\) is an independent set, we can simply multiply the numbers of allowed colours in order to count the number of possible extensions of \(f\).
Algorithm 12 Algorithm \#3C₂.
\[
\text{\#3C₂} \\
1. \ c := 0 \\
2. \text{ for every 3-colouring } f \text{ of } H - I \text{ do} \\
3. \quad c := c + \prod_{v \in I} (3 - |\{f(w) \mid w \in N(v)\}|) \\
4. \text{ end for} \\
5. \text{ return } c
\]

Finally, we consider the time complexity of our \#3-COLOURING algorithm. Assume \( n \) is the number of vertices in the input graph \( H \). Beigel’s [21] algorithm for finding independent sets of maximum size runs in \( O(1.2226^n) \) time. We show below that the worst-case running times of algorithms \#3C₁ and \#3C₂ are in \( O(1.7880^n) \) which clearly dominates Beigel’s algorithm.

If algorithm \#3C₁ does not reach line 4 of the algorithm, it is straightforward to calculate its running time: lines 2 and 3 satisfy the recursive equation \( T(n) \leq T(n - 1) + T(n - 2) + \text{poly}(n) \), and thus we get a work factor of \( \tau(1,2) \), giving a running time \( T(n) \in O(\varphi^n) \) where \( \varphi = (\sqrt{5} + 1)/2 \).

We continue by studying line 4 in the algorithm. If this case is reached, the uncoloured vertices form an independent set \( I' \) in \( H \). Consequently, \( T(n) \in O((2^p \cdot \varphi^{(1-p)})^n) \) where \( p = |I'|/n \). Since \( I \) is a maximum independent set in \( H \), it follows that \( |I'| \leq |I| \leq C \cdot n \). Consequently, the worst case of the algorithm appears when \( g(p) = 2^p \cdot \varphi^{(1-p)} \) is maximised under the constraint \( p \leq C \). Since \( g(p) \) is strictly increasing on \([0,C]\), \( g(p) \) is maximised when \( p = C \). In this case, the algorithm runs in \( O((2^C \cdot \varphi^{(1-C)})^n) \in O(1.7880^n) \) time.

In Algorithm \#3C₂, we know that \( |I| \geq C \cdot n \). Let \( p \) satisfy \( |I| = p \cdot n \). The number of assignments considered is \( 3^n - |I| = 3^{1-p} n \). Since the function \( g(p) = 3^{(1-p)} \) is strictly decreasing when \( p > C \), the largest number of assignments we need to consider appears when \( p \) is close to \( C \). In this case, the algorithm runs in \( O((3^{1-C} n^2) \in O(1.7880^n) \) time. \( \square \)
8.3. \#3-Colouring Algorithm
Chapter 9

Optimisation Problems

The clever combatant looks to the effect of combined energy, and does not require too much from individuals. Hence his ability to pick out the right men and utilize combined energy.

In the wise leader’s plans, considerations of advantage and of disadvantage will be blended together.

Sun Tzu, The Art of War

In this chapter, we will discuss optimisation problems, i.e. problems where we are not satisfied with just any solution, but we require some property of it to be maximised (or minimised.)

Chapter Outline

In Section 9.1 we introduce MAX VALUE CSP, and we begin the discussion by constructing an algorithm for solving the general MAX VALUE \((d,l)\)-CSP in Section 9.1.1. Section 9.1.2 continues with a probabilistic algorithm for MAX VALUE \((d,2)\)-CSP, while Section 9.1.3 contains an approximation algorithm for the same problem, which based on the split-and-list method. The section on MAX VALUE CSP is then concluded with an algorithm for the MAX VALUE \(k\)-Colouring problem.
Next, in Section 9.2, we discuss the Max CSP problem, and give an approximation algorithm for MAX \((d, 2)\)-CSP in Section 9.2.1, while Section 9.2.2 presents an algorithm for MAX \(k\)-COL (which additionally can be used to solve the \#MAX \(k\)-COL problem.)

Section 9.3 then defines MAX IND CSP, and we give a probabilistic algorithm for solving MAX IND \((d, 2)\)-CSP in Section 9.3.1, while Section 9.3.2, contains a split-and-list based algorithm for the same problem. Concluding the section on MAX IND CSP is an algorithm for the MAX \(k\)-COL problem in Section 9.3.3.

The last problem under consideration is the MAXIMUM HAMMING DISTANCE problem, which we discuss in Section 9.4. We begin by giving algorithms for MAX HAMMING \((d, l)\)-CSP and MAX HAMMING \((2, l)\)-CSP in Section 9.4.1. In Section 9.4.2 we find a split-and-list based algorithm for MAX HAMMING \((d, 2)\)-CSP, which we follow up with a microstructure based algorithm for MAX HAMMING \((2, 2)\)-CSP in Section 9.4.3.

### 9.1 Max Value CSP

First, we formally define the problem:

**Definition 36 (Angelsmark et al. [11]).** Let \(\Theta = (X, D, C)\) be an instance of \((d, l)\)-CSP, where \(D = \{a_1, a_2, \ldots, a_d\} \subseteq \mathbb{R}\), \(X = \{x_1, x_2, \ldots, x_n\}\). Given a real vector \(w = (w_1, \ldots, w_n) \in \mathbb{R}^n\), the **MAX VALUE problem** for \(\Theta\) consists in finding a solution \(f : X \rightarrow D\) such that

\[
\sum_{i=1}^{n} w_i \cdot f(x_i)
\]

is maximised.

Note that even though the domain consists of real-valued elements, we are still dealing with a *finite* domain.
9. Optimisation Problems

9.1.1 Max Value \((d, l)\)-CSP Algorithm

The first step in constructing a covering based algorithm for a problem is to find an algorithm for the “base case.” Unfortunately, there exists no previous algorithms for the Max Value problem, and we have to begin with constructing one ourselves. The following lemma gives an upper bound for the running time of an algorithm for Max Value \((2, l)\)-CSP:

**Lemma 37.** There exists an algorithm for solving Max Value \((2, l)\) - CSPs in \(O(\tau(1, 2, \ldots, l)^n)\) time. For \(i = 2, 3, 4, \ldots, \tau(1, 2, \ldots, i) \approx 1.6180, 1.8393, 1.9276, \ldots\)

**Proof.** Let \(((X, D, C), w)\) be an instance of Max Value \((2, l)\)-CSP, with \(D = \{a, b\}\). Using a transformation similar to the one we used in Section 4.1, we reformulate the constraints in \(C\) into an equivalent boolean formula \(\Gamma\) as follows: A variable \(x\) having the value \(a\) corresponds to the boolean variable \(x'\) being true, and \(x = b\) corresponds to \(x'\) begin false. (Using the notation introduced in Section 4.1, we would have written \(x[a]\) rather than \(x'\), but for simplicity we will use this shorter notation here.)

For each constraint \(R(x_1, \ldots, x_l)\) and tuple \((d_1, \ldots, d_l)\), \(d_i \in D, not in R\), add the clause \((y_1 \lor \ldots \lor y_l)\), where \(y_i = x'_i\) if \(d_i = a\) and \(y_i = \neg x'_i\) otherwise, to \(\Gamma\). For example, the familiar constraint \(x \neq y\) gives the clauses

\[(x' \lor y') \land (\neg x' \lor \neg y')\]

just as it did earlier. This transformation can of course be done in polynomial time without introducing new variables. Algorithm 13 shows the algorithm for solving this modified problem.

Recall that \(\Gamma[x = a]\) denotes the boolean expression \(\Gamma\), where \(x\) has been assigned the value \(a\), and note that all satisfying assignments are covered by these \(m\) recursive branches. Since the length \(m\) of a clause is at most \(l\), it can be reduced with a work factor of \(\tau(1, 2, \ldots, l)\), hence Algorithm 13 will have a running time in \(O(\tau(1, 2, \ldots, l)^n)\)

This lemma can now be combined with Theorem 9 to get the following result:
Algorithm 13 Algorithm for Max Value $(2,l)$-CSP.

$\text{MaxVal}_2(\Gamma, w)$

1. Pick a clause $c = (x'_1 \lor \ldots \lor x'_m) \in \Gamma$
2. if $c$ is empty then return 0
3. Let
   
   \begin{align*}
   v_1 &= w_1a + \text{MaxVal}_2(\Gamma[\{x'_1 = \text{true}\}]) \\
   v_2 &= w_1b + w_2a + \text{MaxVal}_2(\Gamma[\{x'_1 = \text{false}, x'_2 = \text{true}\}]) \\
   &\quad \ldots \\
   v_m &= b \cdot \sum_{i=1}^{m-1} w_i + w_ma + \\
   &\quad \text{MaxVal}_2(\Gamma[\{x'_1 = \text{false}, \ldots, x'_{m-1} = \text{false}, x'_m = \text{true}\}])
   \end{align*}

4. return $\max\{v_1, \ldots, v_m\}$

Theorem 38. There exists an algorithm for solving Max Value $(d,l)$-CSP in $O\left((d/2 \cdot \tau(1,2,\ldots,l) + \varepsilon)^n\right)$ time.

Since $\tau(1,2,\ldots,l) < 2$ for all $l$, it holds that the algorithm in Theorem 38 is strictly faster than an exhaustive $O(d^n)$ time search algorithm.

Note that if we restrict the weights of the problem to be positive, then for $l = 2$ and $l = 3$, we can use the weighted 2-SAT and 3-SAT algorithms of Dallholf et al. This would lower the running times for Max Value $(2,2)$-CSP and Max Value $(2,3)$-CSP from $O(1.6180^n)$ and $O(1.8393^n)$ to $O(1.2561^n)$ and $O(1.6737^n)$, respectively.

9.1.2 Max Value $(d,2)$-CSP Algorithm

In this section we present a probabilistic algorithm for solving Max Value $(d,2)$-CSP, together with its derandomisation, which we get
Algorithm 14 Microstructure based algorithm for Max Value (d,2)-CSP.

\textbf{MS-MaxVal (d,2) - CSP}

1. Let $\Theta = (X,D,C)$ and $w = (w_1, \ldots, w_k)$ be the instance.
2. Construct the microstructure graph $G$ of $\Theta$.
3. To each vertex $x[i]$ in $V(G)$, assign the weight $w_i \cdot i$.
4. Find a maximum weighted independent set of size $|X|$ in $G$.

‘for free’ via the covering theorem. Much like in the previous section, we will begin with constructing a deterministic algorithm for a restricted variant of the problem — in this case, we focus on the Max Value (3,2)-CSP.

It is quite straightforward to construct a microstructure based algorithm for the Max Value problem by adding weights to the vertices of the resulting graph and then searching for an independent set of maximum weight, as is shown in Algorithm 14. In this algorithm, we note that

- If $\Theta$ is satisfiable, an independent set of size $|X|$ does exist in $G$, and
- there can be no independent set larger than $|X|$ in $G$.

Clearly, a maximum weighted independent set of size $|X|$ in $G$ maximises $\sum_{i=1}^{k} w_i \cdot (f(x_i))$. An algorithm for counting such independent sets presented by Dahllöf \textit{et al.} [44] has a running time of $O(1.2561^{|V(G)|})$, and it can be modified to return a single solution, thus we can solve Max Value (d,2)-CSP in $O(1.2561^{dn})$ time.

We start by constructing an algorithm for Max Value (3,2)-CSP which runs in $O(1.7458^n)$. This will later form the basis for a probabilistic algorithm for Max Value (d,2)-CSP. Our algorithm actually solves a more general problem where we are given a weight function $w : X \times D \rightarrow \mathbb{R}$ and want to find an assignment $f$ which maximises $\sum_{(x,d) \in f} w(x,d)$. The idea behind the algorithm is to use the microstructure graph and recursively branch on variables with
three possible values until all variables are either assigned a value or only have two possible values left. We then transform the remaining microstructure graph to a weighted 2-SAT instance and solve this.

Before we start, we need some additional definitions: A variable having three possible domain values we call a 3-variable, and a variable having two possible domain values will be called a 2-variable. In order to analyse the algorithm we define the size of an instance as \( m(\Theta) = n_2 + 2n_3 \). Here, \( n_2 \) and \( n_3 \) denote the number of 2- and 3-variables in the instance \( \Theta \), respectively. This means that the size of an instance can be decreased by 1 by either removing a 2-variable or removing one of the possible values for a 3-variable, thus turning it into a 2-variable.

For a variable \( x \in X \) with possible values \( \{d_1, d_2, d_3\} \) ordered so that \( w(x, d_1) > w(x, d_2) > w(x, d_3) \), let \( \delta(x) := (c_1, c_2, c_3) \) where \( c_i = \deg_G(x[d_i]) \), \( G \) being the microstructure graph. If \( x \) is a 2-variable then, similarly, we define \( \delta(v) := (c_1, c_2) \). We will sometimes use expressions such as \( \geq c \) or \( \cdot \) (dot) in this vector, for example \( \delta(v) = (\geq c, \cdot, \cdot) \). This should be read as \( \delta(v) \in \{(c_1, c_2, c_3) : c_1 \geq c\} \). The \textit{maximal} weight of a variable \( x \), i.e., the domain value \( d \) for
which \( w(x,d) \) is maximal, will be denoted \( x_{\text{max}} \). The algorithm is presented as a series of lemmata, and applying these as indicated in Algorithm 15 solves the (slightly generalised) \textsc{Max Value} \((3,2)\)-\textsc{CSP} problem.

**Lemma 39.** For any instance \( \Theta \), we can, in polynomial time, find an instance \( \Theta' \) with the same optimal solution as \( \Theta \), with size smaller or equal to that of \( \Theta \) and to which neither of the following cases apply.

1. There is a 2-variable \( x \) for which \( \delta(x) = (2, \geq 1) \).
2. There is a variable \( x \) for which the maximal weight is unconstrained.

*Proof.* The transformation in Fig. 9.1 takes care of the first case, removing one 2-variable (and therefore decreasing the size of the instance). For the second case, we can simply assign the maximal weight to the variable, leaving us with a smaller instance. \( \square \)

**Lemma 40.** If there is a variable \( x \) with \( \delta(x) = (\geq 3, \geq 2) \) then we can reduce the instance with a work factor of \( \tau(3,2) \).

*Proof.* We branch on the two possible values of \( x \) and propagate the chosen value to the neighbours of \( x \). In one of the branches, the size will decrease by at least 3 and in the other by at least 2. \( \square \)

**Lemma 41.** If there is a variable \( x \) for which \( \delta(x) = (3, \cdot, \cdot) \) then we can reduce the instance with a work factor of \( \tau(3,2) \).

*Proof.* In one of the branches, we choose \( x = x_{\text{max}} \) and propagate this value, decreasing the size by at least 3. In the other branch, choosing \( x \neq x_{\text{max}} \) implies that the value of its exterior neighbour must be chosen in order to force a non-maximal weight to be chosen in \( x \). Therefore, in this branch, the size decreases by at least 2. \( \square \)

**Lemma 42.** If there is a variable \( x \) with \( \delta(x) = (\geq 5, \cdot, \cdot) \) then we can reduce the instance with a work factor of \( \tau(5,1) \).
116

9.1. Max Value CSP

Figure 9.2: The first two cases in Lemma 43. (Note that $x_i$ denotes a variable, while $d_j$ denotes a value.)

Proof. Choosing $x = x_{\text{max}}$ decreases the size of the instance by at least 5. In the other branch, we choose $x \neq x_{\text{max}}$, turning a 3-variable into a 2-variable and thereby decreasing the size by 1. □

If none of Lemma 39 to Lemma 42 are applicable, then every 3-variables must satisfy $\delta(x) = (5, \cdot, \cdot)$ and every 2-variables must satisfy $\delta(x) = (\geq 4, \cdot)$ or $\delta(x) = (3, 0)$.

Lemma 43. If none of Lemma 39 to Lemma 42 is applicable then we can remove any remaining 3-variables with a work factor of $\tau(4, 4, 4)$.

Proof. Let $x_1$ be a 3-variable and $d_1$ its maximal value, with neighbours $x_2$ and $x_3$ as in Figures 9.2 and 9.3.

If $x_2$ is a 2-variable and $\delta(x_2) = (3, 0)$ (see Fig. 9.2a) then we can let one of the branches be $x_1 = d_1$ and the other $x_1 \neq d_1$. This makes $\delta(x_2) = (2, 0)$ in the latter branch, and $x_2$ can be removed by Lemma 39 which means we can decrease the size by 2 in this branch, giving a work factor of $\tau(4, 2) < \tau(4, 4, 4)$. Otherwise if $x_3[d_3]$ has only three neighbours, $x_3$ must be a 3-variable, which implies that $x_3[d_3]$ can not be maximally weighted. If this holds for 0 or 1 of the neighbours of $x_1$ (Fig. 9.2b), we can branch on $x_1 = d_1$, $x_3 = d_3$ and $\{x_2 = d_2, x_3 \neq d_3\}$. In this case, we decrease the size by 4 in each
branch. If both of $x_2[d_2]$ and $x_3[d_3]$ are of degree 3 (Fig. 9.3) and it is not possible to choose an $x_1$ without this property, then for every 3-variable remaining, the maximal weighted value can be assigned without branching at all. \hfill \qed

Algorithm 15 takes as input a microstructure graph $G$ and a weight function $w$, and returns an assignment with maximal weight. Note that in order to actually get a solution to the original problem, one has to a) keep track of the variables eliminated by Lemma 39, and b) extract them from the solution returned on line 4.

**Theorem 44.** *Max Value $(3, 2)$-CSP can be solved by a deterministic algorithm in $O(1.7548^n)$ time.*

**Proof.** We claim that Algorithm 15 correctly solves the Max Value $(3, 2)$-CSP problem.

First we show the correctness of the algorithm. Lemma 39 does not remove any solutions to the problem, it merely reduces the number of vertices in the microstructure. Lemma 40 branches on both possible values for the chosen variable, while Lemmata 41 and 42 try all three possible assignments, as is the case for Lemma 43. This, together with the proof of correctness of the $2$-$SAT_w$ algorithm in [44], shows the correctness of the algorithm.
Algorithm 15 Algorithm for Max Value \((3, 2)\)-CSP.

\begin{algorithm}
\textbf{MaxVal} \((3, 2)\)-CSP \((G, w)\)
\begin{enumerate}
\item \textbf{if} at any time, the domain of a variable becomes empty, that branch can be pruned.
\item Apply the transformations in Lemma 39, keeping track of eliminated variables.
\item \textbf{if} applicable, \textbf{return} the maximum result of the branches described in Lemma 40 to 42
\item \textbf{else if} applicable, \textbf{return} the maximum result of the branches described in Lemma 43
\item \textbf{else return} \(2\)-SAT\(_w\)(\(G, w\))
\end{enumerate}
\end{algorithm}

Now, apart from the call to \(2\)-SAT\(_w\), the highest work factor found in the algorithm is \(\tau(3, 2) = \tau(1, 5) \approx 1.3247\). Recall that we measure the size of \(\Theta\) by \(m(\Theta) = n_2 + 2n_3\) which, for \((3, 2)\)-CSP, is \(2n\), where \(n\) is the number of variables in \(\Theta\). If we can solve weighted 2-SAT in \(O(n^a)\) time, then the entire algorithm will run in \(O(\max(1.3247, c)^{2n})\) time. Using the construction with weighted microstructures mentioned in Section 3.4, a weight function \(w(x, d) = w_i \cdot d\) together with the algorithm for weighted 2-SAT from [44], we get \(c \approx 1.2561\), and the result follows.

The following lemma is needed before we can state the theorem concerning Max Value \((d, 2)\)-CSP:

**Lemma 45.** Assume there exists an \(O(n^a)\) time algorithm for Max Value \((k, l)\)-CSP. Let \(I_k\) denote the set of \((d, l)\)-CSP instances satisfying the following: For each \((X, D, C) \in I_k\), and every \(x \in X\), there exists a unary constraint \((x; S)\) in \(C\) s.t. \(|S| \leq k\). Then, the Max Value problem restricted to instances in \(I_k\) can be solved in \(O(n^a)\) time.

**Proof.** For each variable in \((X, D, C)\), we know it can be assigned at most \(k\) out of the \(d\) possible values. Thus, we can modify the constraints so that every variable picks its value from the set \(\{1, \ldots k\}\).
This transformation can obviously be done in polynomial time, and the resulting instance is an instance of \( \text{MAX VALUE} (k, l) \)-CSP, which can be solved in \( O(n^a) \) time.

**Theorem 46.** There exists an algorithm for solving \( \text{MAX VALUE} \ (d, 2) \)-CSP which has a running time of \( O((0.5849d)^n) \) and finds an optimal solution with probability \( 1 - e^{-1} \), where \( e = 2.7182 \ldots \).

**Proof.** Let \( S = \{ D_1 \times \cdots \times D_n \subseteq D^n, \ \text{with} \ |D_i| = 3 \} \). For each \( \Delta \in S \), define \( \Theta_{\Delta} \) to be the instance \( \Theta \) where each variable \( x_i \) has been restricted to take its value from the set \( \Delta_i \), with \( |\Delta_i| = 3 \), i.e. for each \( x_i \in X \), we add the constraint \( (x_i; \Delta_i) \). Theorem 44 together with Lemma 45 tells us that we can solve this instance in \( O(1.7548^n) \) time.

For a randomly chosen \( \Delta \in S \), the probability that an optimal solution to \( \Theta \) is still in \( \Theta_{\Delta} \) is at least \( (3/d)^n \). It follows that the probability of not finding an optimal solution in \( t \) iterations is at most \( (1 - (3/d)^n)^t < \exp(-t(3/d)^n) \). Therefore, by repeatedly selecting \( \Delta \in S \) at random and solving the induced \( \text{MAX VALUE} \ (3, 2) \)-CSP problem, we can reduce the probability of error to \( \exp(-1) \) in \( (d/3)^n \) iterations. Consequently, we get a running time of \( O((d/3)^n \cdot 1.7548^n) \approx O((0.5849d)^n) \).

We can, of course, also apply the covering theorem and get a derandomised version of the previous theorem:

**Theorem 47.** There exists a covering based algorithm for solving \( \text{MAX VALUE} \ (d, 2) \)-CSP with time complexity in \( O((0.5849d + \varepsilon)^n) \), where \( \varepsilon > 0 \) can be chosen arbitrarily small.

**Proof.** Follows from Theorems 9 and 44, together with Lemma 45.

### 9.1.3 Max Value \( (d, 2) \)-CSP Approximation Algorithm

We will now use the split-and-list method to construct an approximation algorithm for \( \text{MAX VALUE} \ (d, 2) \)-CSP. There are some limitations with this approach, and we will discuss them shortly. However, the main reason for this is that in the \( \text{MAX VALUE} \) problem, as we
9.1. Max Value CSP

state it in Definition 36, we allow arbitrary real-valued weights. If we
enforce some restrictions on the weights, we can apply the method to
get either an exact or an approximation algorithm for the problem,
depending on the restriction.

Consider a Max Value \((d, 2)\)-CSP instance \(\Theta = (X, D, C)\), with
weight vector \(w = (w_1, \ldots, w_n) \in \mathbb{R}^n\), where, without loss of gen-

erality, we assume \(n\) to be an even multiple of 3. Begin by splitting \(X\)
into three disjoint parts, \(X_1, X_2, X_3\), with \(|X_1| = |X_2| = |X_3| = n/3\),
and then listing all \(d^{n/3}\) possible assignments for each of the parti-
tions in lists \(L_1, L_2\) and \(L_3\). Next, we build a graph \(G\) with vertices
\(V(G) := L_1 \cup L_2 \cup L_3\), and edge set

\[ E(G) := \{ (u, v) \mid u, v \in V(G), u \in L_i, v \in L_j, i \neq j \}. \]

Each element \(v\) of \(L_i\) corresponds to a partial assignment of values to
variables, so we define a weight function

\[ w(v) := \sum_{x[a] \in v} w(x) \cdot a, \]
i.e. we define the weight of a vertex in \(G\) to be the sum of all the
weights defined by the partial assignment. For edges, we define the
weight function

\[ w((u, v)) := w(u)/2 + w(v)/2. \] (9.1)

Unfortunately, in order for the construction to work, either of the
following has to hold:

1. \(w((u, v))\) can be stored using \(O(\log(n))\) bits, or
2. \(w((u, v)) \geq 0\) for all \(u, v\).

If the first criterion holds, then we get an exact algorithm, while if
only the second one is true, we have to settle for an approximation
algorithm. The reason for this limitation is that if the first condition
is false, then we are forced to use the all-pair-shortest-path algorithm
by Zwick [116] to find the solution, and it only holds for non-negative
weights.
Now we have a graph with $O(n^{d/3})$ vertices, and depending on the previous conditions, we get different algorithms. If the first condition holds, i.e. we can store each weight using no more than $O(\log_2(n))$ bits, then we can apply either the reduction from weighted to unweighted graphs (as used in [114, Theorem 3.1] together with Theorem 7. If, on the other hand, only the second condition holds, i.e. all weights are non-negative, then we can use the $(1 - \epsilon)$-approximation all-pairs-shortest-paths algorithm by Zwick [116], and get an approximation algorithm. The all-pairs-shortest-paths algorithm has a running time of

$$O\left(\frac{|V(G)|^\omega}{\epsilon} \cdot \log \left(\frac{W}{\epsilon}\right)\right),$$

where $W$ is the largest edge weight found in the graph after scaling the weights such that the smallest non-zero weight is 1, and $\epsilon > 0$ is the approximation ratio.

If neither of the conditions hold, then we cannot apply either of the algorithms. Condition 1 can be checked by examining the weight-function $w$, or we can enforce the second condition by restricting the problem in the following way; in Definition 36, $D$ should be a subset of $\mathbb{R}^+ \cup \{0\}$, and the weight vector $w$ cannot contain any negative weights.

This gives us the following theorem:

**Theorem 48.** If Condition 1, as given earlier, holds, then there exists a split-and-list based $(1 - \epsilon)$-approximation algorithm for Max Value $(d, 2)$-CSP which runs in $O\left(d^{\omega n^{d/3}}/\epsilon\right)$ time and uses $O\left(d^{n/3}\right)$ space, where $\omega$ is the exponent of matrix multiplication. Furthermore, if Condition 2 holds, then there exists an exact split-and-list algorithm which runs in $O\left(d^{n^{d/3}}\right)$ time.

**Proof.** Follows from the previous discussion. \qed

### 9.1.4 Max Value $k$-COL Algorithm

The version of the Max Value problem where we only allow disequality as a constraint, i.e. the Max Value $k$-COL problem, is defined as follows:
Algorithm 16 Algorithm for Max Value 2-COL

MaxVal 2-COL (Θ, w)
1. Let $G$ be the microstructure of $Θ$.
2. $m := 0$
3. if $G$ is 2-colourable then
4. Let $f : V(G) \rightarrow \{1, 2\}$ be a 2-colouring of $G$
5. for each connected component $c$ of $G$ do
6. $m := m +$
7. end for
8. end if
9. return $m$

Definition 49. Given a graph $G$, with $|V(G)| = n$, a real vector of weights $w = (w_1, \ldots, w_n) \in \mathbb{R}^n$ and a natural number $k$, the Max Value $k$-COL problem consists in finding a function $f : V(G) \rightarrow \{1, \ldots, k\}$, with $f(v) \neq f(w)$ if $(v, w) \in E(G)$, such that
$$\sum_{v \in V(G)} w_v \cdot f(v)$$
is maximised.

In order to avoid confusion, we will from now on let $(Θ, w)$, where $Θ = (X, D, C)$, be an instance of Max Value 2-COL. Let $G$ be the microstructure graph of $Θ$, and, for $x \in X$, let $η(x)$ be the number of constraints $x$ is involved in—if we look at the microstructure, this corresponds to $\deg(x[i]) - 1$.

Theorem 50. There exists an algorithm for solving the Max Value 2-COL problem which runs in polynomial time.
Proof. We will show that Algorithm 16 correctly solves the MAX VALUE 2-COL problem.

First of all, we note that if the microstructure graph is not 2-colourable, then the MAX VALUE 2-COL instance has the trivial solution 0, since there are no colourings, and this is what the algorithm returns.

Next we observe that if a 2-colouring exists, then for each of the connected component in \( G \), there are exactly two possible colourings. Consequently, since we can choose the colour with largest weight for each connected component in isolation, when the algorithm reaches line 9, \( m \) will contain the weight of the maximum solution.

In order to successfully apply the partitioning method here, we need to take care of odd-sized colourings, and thus we need an algorithm for the MAX VALUE 3-COL problem.

First, recall the definitions from Section 9.1.2; variables with two and three possible domain values are called 2- and 3-variables, respectively. The size of an instance is defined as \( m(\Theta) = n_2 + 2n_3 \), where \( n_i \) denotes the number of \( i \)-variables in \( \Theta \). Consequently, the size of an instance can be decreased by either fully remove 2-variables or removing one of the possible values for a 3-variable, and turn it into a 2-variable.

Given a variable \( x \) with three possible values, \( \{d_1, d_2, d_3\} \), ordered in such a way that \( w(x, d_1) > w(x, d_2) > w(x, d_3) \), let \( \delta(x) := (c_1, c_2, c_3) \) where \( c_i = \deg_G(x[d_i]) \), \( G \) being the microstructure graph. If \( x \) is a 2-variable then, similarly, we define \( \delta(v) := (c_1, c_2) \). The maximal weight of a variable \( x \), i.e. the domain value \( d \) for which \( w(x, d) \) is maximal, will be denoted \( x_{\text{max}} \).

Since the only allowed constraint is ‘\( \neq \)’, it is never the case that a 3-variable can have two unconstrained values — for example, if \( x[d_1] \) had an edge to \( y[d_1] \), but \( x[d_2] \) and \( x[d_3] \) had no edges to \( y \), this would mean that vertices \( y[d_2] \) and \( y[d_3] \) had already been removed, and thus we could propagate \( y[d_1] \), the only possible value for \( y \).

Lemma 51. If there is a variable \( x \) with \( \delta(x) = (\geq 3, \cdot, \cdot) \), we can reduce the instance with a work factor of \( \tau(4,2) \).
Algorithm 17 Algorithm for Max Value 3-COL.

\begin{align*}
\text{MaxVal 3-COL} \ (G, w) & \\
1. & \text{if at any time, the domain of a variable becomes empty,} \\
& \quad \text{that branch can be pruned.} \\
2. & \text{Apply Lemma 39, keeping track of eliminated variables.} \\
3. & \text{if applicable return the maximum of the branches described in} \\
& \quad \text{Lemma 51} \\
4. & \text{else} \\
5. & \quad \text{Let } \Gamma_w \text{ be the weighted 2-SAT instance corresponding to } G. \\
6. & \quad \text{return } 2\text{-SAT}_w(\Gamma_w) \\
7. & \text{endif}
\end{align*}

Proof. If \( x_{\max} \) is chosen, then we remove \( x \) together with its two external neighbours, thus reducing the size of the instance by (at least) 4. The only reason \textit{not} to choose \( x_{\max} \) is, of course, that one of its external neighbours was chosen — thus reducing the size of the instance by 2.

After applying the reduction in this lemma, it is the case that \textit{no} variable \( x \) has \( x_{\max} \) with degree greater than 2. This means that either the neighbours if \( x_{\max} \) are the other possible values of \( x \), and thus \( x \) is unconstrained, or one of the other values has been eliminated, and there are only two possible values for \( x \). We can apply Lemma 39 to get rid of all of the cases of unconstrained variables, and what we have left is an instance of weighted 2-SAT.

Theorem 52. Max Value 3-COL can be solved by a deterministic algorithm in time \( \mathcal{O}(1.6181^n) \).

Proof. Algorithm 17 has, apart from the call to 2-SAT, a work factor of \( \tau(4, 2) \leq 1.2721 \). Since we used \( m(\Theta) = n_2 + 2n_3 \) as the measure of size, the size of an instance is \( 2n \). Consequently, the algorithm has a running time of \( \mathcal{O}((\max(1.2721, 1.2561))^{2n}), \) i.e. \( \mathcal{O}(1.6181^n) \).}

Since we are only considering colourings, we can apply Theorem 18 and use the recursive partitioning scheme from Section 7.1.
Theorem 53. If we can solve MAX VALUE 2-COL in polynomial time, and MAX VALUE 3-COL in $O(\beta_3^n)$ time, respectively, then there exists a partitioning based algorithm for solving MAX VALUE $k$-COL which has a running time of $O(\alpha_k^n)$, where

$$\alpha_k = \begin{cases} 
    i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\
    i + 1 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} 
\end{cases}$$

and $i \geq 4$.

Proof. Similar to the proof of Theorem 58, we recursively use the partitioning $[\lfloor k/2 \rfloor, \lceil k/2 \rceil]$. Consequently, we can use Theorem 18 to get an algorithm which will have a running time of $O(\alpha_k^n)$, where $\alpha_k$ is given by the solution to the following recurrence:

$$\alpha_k = \begin{cases} 
    1 & \text{if } k = 2 \\
    \beta_3 & \text{if } k = 3 \\
    1 + \alpha_{\lfloor k/2 \rfloor} & \text{otherwise}
\end{cases}$$

Solving the equation gives the result. □

By combining the previous theorem with algorithms 16 and 17, we then get an algorithm for MAX VALUE $k$-COL.

9.2 Max CSP

The first of the partial constraint satisfaction problems we will look at is MAX CSP, where the goal is to maximise the number of satisfied constraints in the problem.

We show how a straightforward application of the covering method will give an approximation algorithm for MAX $(d, 2)$-CSP. When this algorithm was first developed, no algorithm for this problem running in provably less than $O(d^n)$ time was known. Consequently, in order to get an idea of its efficiency, we carried out the generalisation of the MAX $k$-SAT approximation algorithm suggested by Hirsch [73]. Since then we have seen the arrival of the split-and-list algorithm of Williams [114]. It turns out that our approximation algorithm is
faster for domains less than 29, thus it remains a viable alternative for small domains.

Formally, we define the problem as follows:

**Definition 54.** Given an instance \( \Theta = (X, D, C) \) of \((d, 2)\)-CSP, the Max \((d, 2)\)-CSP problem is to find an assignment \( f : X \rightarrow D \) which satisfies the maximum number of constraints.

### 9.2.1 Max \((d, 2)\)-CSP Approximation Algorithm

In order to successfully apply the covering method to this problem, we need a deterministic, preferably polynomial, case to start with. Unfortunately, Max \((2, 2)\)-CSP is not polynomial but, luckily, there exist very good approximation algorithms for it. Goemans & Williamson, in their celebrated and award-winning Max Cut paper [65], present a 0.796-approximation algorithm for this problem. (By 0.796-approximation we mean that the algorithm returns a solution which is within 0.796 times the optimal.) This was later improved to 0.859 by Feige & Goemans [55], and Mahajan & Ramesh presents a derandomised version of the algorithm [95], which is exactly what we need to state the following theorem:

**Theorem 55.** There exists an 0.859-approximation algorithm for Max \((d, 2)\)-CSP which has running time of \( \mathcal{O}((d/2 + \varepsilon)^n) \).

**Proof.** By combining the derandomised, polynomial time approximation algorithm by Mahajan & Ramesh [95] with the results in Section 4.2, specifically Theorem 9, the result follows.

To our knowledge, this was the first such algorithm to have been presented, although it had been suggested in [73] that the \((1 - \varepsilon)\)-approximation for Max \(k\)-SAT algorithm presented there could be generalised to an approximation algorithm for Max \((d, 2)\)-CSP. The generalisation we finally came up with is described in Theorem 56, and the resulting algorithm is Algorithm 18.

**Theorem 56.** There exists an algorithm for finding an assignment of values to variables of a Max CSP instance which satisfies at least \((1 - \varepsilon)\)-approximation.
9. Optimisation Problems

Optimisation Problems

$\varepsilon \cdot \text{OptVal}(\Theta)$ constraints, where $\text{OptVal}(\Theta)$ denotes the maximum number of satisfiable constraints, with probability at least $1 - 1/e$, where $e \approx 2.7182\ldots$, and has a running time of

$$O \left( \left( d - \frac{d\varepsilon}{l \cdot (d^2 - 1 + \varepsilon) + \varepsilon} \right)^n \right)$$

Proof. Let $\Theta = (V, D, C)$ be a CSP instance with $|D| = d$ and $|V| = n$, and let $S^{\text{opt}}$ denote the optimal assignment of values to the variables in $\Theta$, satisfying $m^{\text{opt}}$ constraints. We say that an assignment is admissible if it satisfies at least $(1 - \varepsilon)m^{\text{opt}}$ constraints.

Assume assignment $A$ is not admissible. Then $A$ does not satisfy $u > |C| - (1 - \varepsilon)m^{\text{opt}}$ constraints. Among these, at least $u - (|C| - m^{\text{opt}})$ are satisfied in $S^{\text{opt}}$. (See Fig. 9.4.)

From now on, we only consider the worst-case scenario, i.e. there is only one optimal assignment $S^{\text{opt}}$. The probability of changing the value of a variable with different values in $S^{\text{opt}}$ and $A$ follows:

$$p_0 = \frac{u - (|C| - m^{\text{opt}})}{u}$$

Here $u$ is the number of unsatisfied constraints in $A$ and $u - (|C| - m^{\text{opt}})$ the number of constraints which are satisfied in $S^{\text{opt}}$ but not in $A$. Thus $p_0$ is the probability of choosing a constraint $c$ which is satisfied in $S^{\text{opt}}$ but not in $A$. 

---

Figure 9.4: Satisfied and unsatisfied constraints.
A constraint has arity \( l \), i.e. there are \( l \) variables to choose from once we have chosen the constraint. In the worst case, only one of these has differing values in \( S^{\text{opt}} \) and \( A \), giving the probability \( p_1 = l^{-1} \) of picking it, and, having chosen a variable, the probability of picking the correct value for it is \( p_2 = (d-1)^{-1} \). This gives us the probability of changing the value of a variable which differs in the assignments \( A \) and \( S^{\text{opt}} \) into the value it has in \( S^{\text{opt}} \) to be

\[
p_3 = p_0 \cdot p_1 \cdot p_2 = \frac{u - (|C| - m^{\text{opt}})}{u} \cdot \frac{1}{l \cdot (d-1)} = \\
\frac{1}{l \cdot (d-1)} - \frac{|C| - m^{\text{opt}}}{l \cdot (d-1) \cdot u} \geq u > |C| - (1 - \varepsilon) \cdot m^{\text{opt}}/\varepsilon m^{\text{opt}} \\
\frac{1}{l \cdot (d-1)} - \frac{|C| - m^{\text{opt}}}{l \cdot (d-1) \cdot (|C| - (1 - \varepsilon) \cdot m^{\text{opt}})} = \frac{1}{l \cdot (d-1) \cdot (|C| - (1 - \varepsilon) \cdot m^{\text{opt}})}
\]

Pick a random assignment. Let the binary stochastic variable \( X_i = 1 \) if \( c_i \) is satisfied by this assignment and 0 otherwise. For all \( i \), \( \Pr(X_i = 1) \geq (1/d)^l \) so \( E(X_i) \geq (1/d)^l \). The \( X_i \) variables are not necessarily independent but their expectation is still additive and the expected number of satisfied constraints is \( E(\sum_i X_i) = \sum_i E(X_i) \geq |C|(1/d)^l \). Hence, there exists at least one assignment with at least \( |C|(1/d)^l \) satisfied constraints. We conclude that \( m^{\text{opt}} \geq |C|/d^l \).

Hence;

\[
\frac{\varepsilon m^{\text{opt}}}{l \cdot (d-1) \cdot (|C| - (1 - \varepsilon) \cdot m^{\text{opt}})} \geq \frac{\varepsilon m^{\text{opt}}}{l \cdot (d-1) \cdot (d^l m^{\text{opt}} - (1 - \varepsilon)m^{\text{opt}})}
\]

which gives

\[
p_3 = \frac{\varepsilon}{l \cdot (d-1) \cdot (d^l - 1 + \varepsilon)}
\]

as the probability of changing the value of a variable that differs in \( S^{\text{opt}} \) and \( A \) into the value it has in \( S^{\text{opt}} \).
There are \(d^n\) possible assignments, \(\binom{n}{i}(d-1)^i\) of which differ from \(S^{opt}\) in \(i\) positions. The success probability for such an assignment is \(p_3^i\). (For each of the differing variables we have probability \(p_3\) of choosing and changing it correctly.)

\[
\frac{\binom{n}{i}(d-1)^i}{d^n} \left(\frac{\varepsilon}{l \cdot (d-1) \cdot (d^l - 1 + \varepsilon)}\right)^i = \frac{\binom{n}{i}}{d^n} \left(\frac{\varepsilon}{l \cdot (d^l - 1 + \varepsilon)}\right)^i
\]

Summing over all the possible assignments we get

\[
p_4 = \frac{1}{d^n} \sum_{i=0}^{n} \binom{n}{i} \left(\frac{\varepsilon}{l \cdot (d^l - 1 + \varepsilon)}\right)^i = \frac{1}{d^n} \left(1 + \frac{\varepsilon}{l \cdot (d^l - 1 + \varepsilon)}\right)^n
\]

so by choosing

\[
t = d^n \left(1 + \frac{\varepsilon}{l \cdot (d^l - 1 + \varepsilon)}\right)^{-n} = d^n \cdot \left(1 - \frac{\varepsilon}{l \cdot (d^l - 1 + \varepsilon) + \varepsilon}\right)^n
\]

initial assignments, the algorithm will fail to find an admissible assignment with probability \((1 - p_4)^{1/p_4} < 1/e\), giving a success probability of \(1 - 1/e\).

9.2.2 Max \(k\)-COL and \(#\text{Max } k\)-COL Algorithms

If we restrict MAX CSP to only use the disequality constraint, we get the Max \(k\)-Colourable Subgraph problem, or Max \(k\)-COL—also known as the unweighted case of the Max \(k\)-CUT problem. Formally, we define it as follows:

**Definition 57.** Given a graph \(G\) and a natural number \(k\), the Max \(k\)-COL problem is to find a subset \(E'\) of \(E(G)\) such that the graph \((V(G), E')\) is \(k\)-colourable and \(|E'|\) maximised. The problem of determining the number of such subsets is denoted \(#\text{Max } k\)-COL.

From Williams [114], we know that Max \(k\)-COL can be solved in \(O(k^{\omega n/3})\) time, but we can improve this bound using the partitioning method:
Algorithm 18 $(1 - \varepsilon)$ approximation algorithm for Max $(d,l)$-CSP.

$(1 - \varepsilon)$ Max $(d,l)$-CSP

1. $s_{\text{max}} := -1$
2. repeat $\left\lceil \left( d - \frac{d \varepsilon}{l(d'-1+\varepsilon) + \varepsilon} \right) \right\rceil^n$ times
3. Randomly choose an assignment $S$
4. repeat $n - 1$ times
5. if $S$ satisfies $\Theta$ then return $S$
6. Choose an unsatisfied constraint of $\Theta$ at random
7. Choose a variable $x$ in this constraint at random
8. Change the assignment of $x$ in $S$ randomly
9. Let $s$ be the number of constraints satisfied by $S$
10. if $s > s_{\text{max}}$ then
11. $s_{\text{max}} := s$
12. end repeat
13. end repeat
14. return the assignment corresponding to $s_{\text{max}}$ satisfied constraints.

Theorem 58. Given that we can solve Max 2-COL and Max 3-COL ($\#\text{Max }2\text{-COL}$ and $\#\text{Max }3\text{-COL}$) in time $O(\beta_2^n)$ and $O(\beta_3^n)$, respectively, there exists a partitioning based algorithm for solving Max $k$-COL ($\#\text{Max }k\text{-COL}$) which has a running time of $O(\alpha_k^n)$, where

$$\alpha_k = \begin{cases} 
  i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\
  i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1}
\end{cases}$$

for $i \geq 4$. Furthermore, the space requirement is $O(2^{n/3})$ for the first case, and $O(3^{n/3})$ for the second.

Proof. Again, we use the partitioning $[[k/2], [k/2]]$ recursively. From Theorem 18, we know that a partitioning based algorithm will have a running time of $O(\alpha_k^n)$, where $\alpha_k$ is given by the solution to the
following recurrence:

\[
\alpha_k = \begin{cases} 
\beta_2 & \text{if } k = 2 \\
\beta_3 & \text{if } k = 3 \\
1 + \alpha_{\lceil k/2 \rceil} & \text{otherwise}
\end{cases}
\]

Solving the equation gives the time complexity stated in the theorem.

As for the space complexity, the algorithms for Max 2-COL and Max 3-COL utilises \(O\left(2^{n/3}\right)\) and \(O\left(3^{n/3}\right)\) space, respectively. \(\square\)

We can then combine this theorem with the fact that there exists algorithms for Max 2-COL (\#Max 2-COL) and Max 3-COL (\#Max 3-COL) with running times of \(O(1.7315^n)\) and \(O(2.3872^n)\), respectively, to get an algorithm for the general Max \(k\)-COL (and \#Max \(k\)-COL) problems.

### 9.3 Max Ind CSP

We will now consider the Max Ind problem for CSPs—the second of the partial CSPs we study. Formally, we define the problem as follows:

**Definition 59 (Jonsson & Liberatore [83])**. Let \(\Theta = (X, D, C)\) be an instance of \((d, l)\)-CSP. The Max Ind \((d, l)\)-CSP problem consists of finding a maximal subset \(X' \subseteq X\) such that \(\Theta|X'\) is satisfiable.

Here, \(\Theta|X' = (X', D, C')\) is the subinstance of \(\Theta\) induced by \(X'\), i.e. the CSP we get when we restrict \(\Theta\) to the variables in \(X'\) and the constraints which involve only variables from \(X'\), viz.

\[C' := \{c \in C \mid c(x_1, x_2, \ldots, x_l) \in C, x_1, \ldots, x_l \in X'\}.\]

### 9.3.1 Max Ind \((d, 2)\)-CSP Algorithm

We restrict our study to the case with binary constraints, i.e. Max Ind \((d, 2)\)-CSP, and the algorithm we present is based on the microstructure of the CSP. First, we will need the following lemma:
Lemma 60. A binary CSP $\Theta = (X, D, C)$ contains a maximum induced subinstance $\Theta|X' = (X', D, C')$ iff the microstructure graph of $\Theta$ contains a maximum independent set $f$ with $|f| = |X'|$.

Proof. $\Rightarrow$): Assume $\Theta|X'$ is a maximum induced subinstance of $\Theta$. Then there exists an assignment $f$ of values to the variables in $X'$ such that no constraint in $C'$ is violated. In the microstructure graph we can identify $f$ as a set of vertices (by definition) $X_f$, and since $\Theta|X'$ is satisfiable, $X_f$ will induce a subgraph with empty edge set, i.e. an independent set. Furthermore, $X_f$ is obviously an independent set, since otherwise we could add another variable-value pair to it, and this would contradict the assumption that $\Theta|X'$ is a maximum induced subinstance.

$\Leftarrow$): Assume the microstructure graph of $\Theta$ contains a maximum independent set $f$. Since each vertex $x[i] \in f$ corresponds to an assignment $x := i$ in $\Theta$, we can observe two things: Since $f$ is an independent set, this means that no constraint of $\Theta$ is violated by the assignments made in $f$, and since $f$ is a maximum independent set there exists no such set which is strictly larger. Consequently, $\Theta$ when restricted to variables being assigned values by $f$ will form a maximum induced subinstance of $\Theta$.

Thus we get the following corollary:

Lemma 61. Max Ind $(d, 2)$-CSP can be solved in $O(dn^c)$ time, where $d$ is the domain size, $n$ the number of variables and $O(e^n)$ is the running time of an algorithm for solving Maximum Independent Set.

Proof. Let $\Theta = (X, D, C)$ and consider the microstructure of $\Theta$. This graph will contain $d \cdot n$ vertices, and if we can find a maximum independent set in $O(e^{V(G)})$ time then, by Lemma 60, we can solve MAX IND $(d, 2)$-CSP in $O(e^{dn})$ time.

Corollary 62. Max Ind $(d, 2)$-CSP can be solved in $O(1.2025^{dn})$ time using polynomial space, and in $O(1.1889^{dn})$ time using exponential space.
9. Optimisation Problems

Algorithm 19 Algorithm for Max Ind \((d, 2)\)-CSP.

\[
\text{MaxInd} \ (d, 2)\text{-CSP} \ (\Theta = (X, D, C))
\]

1. \(s := \emptyset\)
2. repeat \((d/k)^n\) times
3. Randomly choose \(\Delta \in S_k\)
4. Let \(G\) be the microstructure graph of \(\Theta_\Delta\)
5. \(f := \text{MIS}(G)\)
6. if \(|s| < |f|\) then
7. \(s := f\)
8. end repeat
9. return \(s\)

Proof. Combining Lemma 61 with the Maximum Independent Set algorithms by Robson [102] gives the result. The algorithms have a time complexity of \(O\left(1.2025|V(G)|\right)\), using polynomial space, and \(O\left(1.1889|V(G)|\right)\), using exponential.

We will now construct a much faster probabilistic algorithm for this problem, and we begin by noting that Lemma 45 holds for Max Ind \((d, 2)\)-CSPs as well. The idea behind the algorithm is to use Corollary 62 to solve randomly chosen, restricted instances of the original CSP instance. In Algorithm 19, we assume the existence of an algorithm MIS for finding maximum independent sets in arbitrary graphs. (Of course, we can get an arbitrarily high probability of success through iteration.)

**Theorem 63.** Given an \(O\left(a|V(G)|\right)\) time algorithm for solving Maximum Independent Set for arbitrary graphs, there exists a probabilistic algorithm for Max Ind \((d, 2)\)-CSP which returns an optimal solution with probability at least \(1 - e^{-1}\), and runs in \(O\left(\left(\frac{d}{k} \cdot a^k\right)^n\right)\) time, for \(d \geq k\). Furthermore, either \([(\ln c)^{-1}]\) or \([(\ln c)^{-2}]\) is the best choice for \(k\).

Proof. Combining Lemma 45 with the fact that we can solve Max Ind \((k, 2)\)-CSP in \(O\left(c^{kn}\right)\) time using microstructures, we can solve
a MAX IND \((d, 2)\)-CSP instance where the variables are restricted to domains of size \(k\) in \(O\left(k^n\right)\) time.

For a randomly chosen \(\Delta \in S_k\), the probability that an optimal solution to \(\Theta\) is still in \(\Theta_\Delta\) is at least \((\frac{k}{d})^n\). No additional solution can be introduced by restricting the domains of the variables, and a solution \(f\) which is optimal for \(\Theta\), is still a solution (and hence optimal) to any \(\Theta_\Delta\) for which \(f(x_i) \in \Delta_i, i \in \{1, \ldots, n\}\).

It follows that the probability of not finding an optimal solution in \(t\) iterations is at most \((1 - (k/d)^n)^t < \exp(-t(k/d)^n)\). Therefore, by repeatedly selecting \(\Delta \in S_k\) at random and solving the induced MAX IND \((k, 2)\)-CSP problem, we can reduce the probability of error to \(\epsilon^{-1}\) in \((d/k)^n\) iterations. The result is Algorithm 19, and it has a running time of \(O\left(\left(\frac{d}{k} \cdot c^k\right)^n\right)\).

Now, let \(g(x) = cx/x\). The function \(g\) is convex and assumes its minimum at \(x' = \left(\ln c\right)^{-1}\). Thus, for a given \(c\), finding the minimum of \(g(\lfloor x' \rfloor)\) and \(g(\lceil x' \rceil)\) determines the best choice of \(k\).

This immediately gives us the following corollary:

**Corollary 64.** For \(d \geq 5\), there exists a probabilistic algorithm for solving MAX IND \((d, 2)\)-CSP returning an optimal solution with probability at least \(1 - \epsilon^{-1}\) in \(O((0.5029d)^n)\) time using polynomial space, and, for \(d \geq 6\), there exists an algorithm which runs in \(O((0.4707d)^n)\) using exponential space.

Similar to the case with MAX VALUE earlier, we can also apply the covering method in order to get a deterministic algorithm for this problem. Coincidentally, we can “reuse” the result from Theorem 63 to get the size of the instances to cover with, and arrive at:

**Corollary 65.** There exists a covering based algorithm for MAX IND \((d, 2)\)-CSP, \(d \geq 5\), which has a running time of \(O((0.5029d + \epsilon)^n)\).

### 9.3.2 Max Ind \((d, 2)\)-CSP Algorithm (Again)

Similar to the previous optimisation problems, we can also construct a split and list algorithm for the MAX IND problem. Since we need a
way to express that a variable has not been assigned, we add a value to the domain of the problem, which is then taken to mean “undefined.”

So, given a Max Ind $(d, 2)$-CSP instance $\Theta = (X, D, C)$, we begin by adding an element $\bot$ to $D$, and let $\bot$ represent the case when a variable has not been assigned a value. Next, we proceed with splitting the set $X$ into 3 parts, and list all possible assignments for each of these partitions. Using these lists, we then construct a graph exactly as described in Section 9.1.3. The vertices in the graph correspond to partial assignments of values to variables, so we define the weight function $w$ to be the number of assignments in a vertex (recall that $\bot$ corresponds to no assignment), i.e.

$$w(v) = |\{ x[a] \in v \mid a \neq \bot \}|$$

and, similarly to what we did for Max Value, we then construct an edge weight function from this:

$$w((u, v)) := w(u)/2 + w(v)/2.$$ 

However, unlike the case with Max Value, we now know the maximum weight of a triangle, which is $n$ and happens when all variables have been assigned a value, and condition 1 is always satisfied. Of course, the fast algorithm for finding 3-cliques only works for unweighted graphs (see Theorem 7), whereas we have a very much weighted graph, so we have to get around this somehow. Luckily, we can use the exact same reduction as in [114], and arrive at the following theorem:

**Theorem 66.** There exists a split and list algorithm for Max Ind $(d, 2)$-CSP, which has a time complexity of $O((d + 1)^{\omega n/3})$ and uses $O((d + 1)^{n/3})$ space, where $\omega < 2.376$ is the exponent of matrix multiplication (see Theorem 7.)

**Proof.** Follows from the previous discussion together with the reduction from weighted to unweighted graphs found in Williams [114, Theorem 3.1].
9.3.3 Max Ind k-COL Algorithm

If we restrict ourselves to CSPs where the only relation allowed is ‘$\neq$’, we get the problem Max Ind k-COL. Note the difference to the Max COL problem in Definition 57—we are now dealing with an induced subgraph.

**Definition 67.** Given a graph $G$ and a natural number $k$, the Max Ind $k$-COL problem is to find a subset $S \subseteq V(G)$ such that the induced subgraph $G(S)$ is $k$-colourable and $|S|$ is maximised.

Though the problem is still NP-complete for this case (see Jonsson & Liberatore [83]), we can improve the time complexity from Corollary 64. Theorem 18 is applicable to this problem as well. From Corollary 62 we know that Max Ind 2-COL and Max Ind 3-COL can be solved in $O(1.2025^{2n}) = O(1.4460^n)$ and $O(1.2025^{3n}) = O(1.7388^n)$ time, respectively, and we can combine these with the following theorem to get an algorithm for Max Ind $k$-COL.

**Theorem 68.** Given that we can solve Max Ind 2-COL and Max Ind 3-COL in time $O(\beta_2^n)$ and $O(\beta_3^n)$, respectively, there exists a partitioning based algorithm for solving Max Ind $k$-COL which has a running time of $O(\alpha_k^n)$, where

$$\alpha_k = \begin{cases} i - 1 + \beta_3 & \text{if } 2^i < k \leq 2^i + 2^{i-1} \\ i + \beta_2 & \text{if } 2^i + 2^{i-1} < k \leq 2^{i+1} \end{cases}$$

for $i \geq 4$.

**Proof.** Yet again, we use the partitioning $\lfloor k/2 \rfloor, \lceil k/2 \rceil$ recursively, and we get from Theorem 18 that a partitioning based algorithm will have a running time of $O(\alpha_k^n)$, where $\alpha_k$ is given by the solution to the following recurrence:

$$\alpha_k = \begin{cases} \beta_2 & \text{if } k = 2 \\ \beta_3 & \text{if } k = 3 \\ 1 + \alpha_{\lfloor k/2 \rfloor} & \text{otherwise} \end{cases}$$
9.4 The Maximum Hamming Distance Problem

Before we can discuss the problem, we need the following definitions:

**Definition 69 (Crescenzi & Rossi [42]).** Given a set of variables $X$, the Hamming distance between a pair of assignments $f_1$ and $f_2$ of values to the variables in $X$, denoted $d_H(f_1, f_2)$, is the number of variables on which the two assignments disagree.

For example, consider the 2-SAT formula $(x \lor y) \land (\neg x \lor z)$, together with the assignments $f_1 = \{ x \mapsto 0, y \mapsto 1, z \mapsto 0 \}$, $f_2 = \{ x \mapsto 1, y \mapsto 1, z \mapsto 1 \}$.

Clearly, $f_1$ and $f_2$ both satisfy the formula, and their Hamming distance is $d_H(f_1, f_2) = 2$, since they have different values for two variables, $x$ and $z$.

**Definition 70 (Angelsmark & Thapper [13]).** Let $\Theta = (X, D, C)$ be an instance of $(d, l)$-CSP. The **Max Hamming Distance** $(d, l)$-CSP problem is to find two satisfying assignments $f, g$ to $\Theta$ which maximises $d_H(f, g)$.

The reason we began studying this problem was that a naïve enumeration algorithm would require $O(d^{2n})$ time to solve this problem, and it was not immediately obvious how to improve this bound. (In contrast, almost all other CSP problems can be solved in $O(d^n)$ time using enumeration.) In the following section, we will describe a less naïve algorithm for solving the general $(d, l)$-CSP case, as well as a specialised version for $(2, l)$-CSPs. Additionally, a rather more complicated algorithm for the extra special case with $(2, 2)$-CSPs can be constructed by exploiting the microstructure of the problem.

9.4.1 Max Hamming Distance $(d, l)$-CSP Algorithm

Before we start, let us consider the following problem: Given a CSP instance $\Theta = (X, D, C)$, can we find a pair of solutions differing on $k$
variables? If we can solve this in a reasonable amount of time, then we can construct an algorithm for Maximum Hamming Distance by trying out all possible $k$. An obvious way of solving the problem is sketched in Algorithm 20.

There are $2^n$ ways to choose $Y$ on the first line, so if we can solve the satisfiability problem for $\Theta$ in time $O(h(n))$, then, since the number of variables in $\Theta'$ is twice that of $\Theta$, we can find a pair of solutions with maximum Hamming distance in $O(2^n h(2n))$ time. For example, 2-SAT can be solved in linear time, thus we would, using this method, get a running time of $O(2^n)$ for Max Hamming Distance $(2, 2)$-CSP. We will see later that this is a poor bound for this problem, but it provides us with algorithms for Hamming distance problems with larger domains and higher arity constraints.

Now we note that it is not actually necessary to make a copy of all the variables in the instance. Since we add the constraint $x = x'$ for all
Algorithm 21 Algorithm for MAX HAMMING DISTANCE \((d,l)\)-CSP

\[
\text{MaxHammingDistance} \ (d,l)\text{-CSP} \ (\Theta = (X, D, C))
\]

1. for \(k := |X|\) downto 0 do
2. \hspace{1em} for each \(\xi \subseteq X\) s.t. \(|\xi| = k\) do
3. \hspace{2em} Let \(\Theta' = (X', D, C')\) be a copy of \(\Theta\).
4. \hspace{2em} Let \(\gamma \subseteq C\) be all constraints involving variables in \(\xi\).
5. \hspace{2em} Create \(\gamma'\) by exchanging all variables not in \(\xi\) with their counterparts from \(X'\).
6. \hspace{2em} \(C' := C' \cup \gamma'\)
7. \hspace{1em} for each \(x \in \xi\) do
8. \hspace{2em} \(C' := C' \cup \{x \neq x'\}\)
9. \hspace{1em} if \((X \cup X', D, C')\) is satisfiable with a solution \(\sigma\) then
10. \hspace{2em} \hspace{1em} Let \(f, g\) be the two assignments found in \(\sigma\)
11. \hspace{2em} \hspace{1em} return \((f, g)\)
12. \hspace{1em} end if
13. \hspace{1em} end for
14. end for

variables which we require to have the same value in both solutions, there is really no need to include these variables at all, since the copy will always behave exactly as the original. Thus if we want \(k\) variables to differ, we duplicate only these variables, leaving the remaining \(n-k\) variables unchanged. Consequently, we get Algorithm 21, and the following theorem.

**Theorem 71.** If we can solve \((d,l)\)-CSP in \(\mathcal{O}(a^n)\) time, then there exists an \(\mathcal{O}((a(1+a))^n)\) time algorithm for solving MAX HAMMING DISTANCE \((d,l)\)-CSP.

**Proof.** In Algorithm 21, the instance \((X \cup X', D, C')\) will contain \(n+k\) variables, and there are \(\binom{n}{k}\) ways of choosing \(\xi\). Consequently, given that we can solve the \((d,l)\)-CSP instance on line 9 in \(\mathcal{O}(a^n)\) time, the
9.4. The Maximum Hamming Distance Problem

The algorithm will have a running time of

\[ \sum_{k=0}^{n} \binom{n}{k} a^{n-k} = a^n \sum_{k=0}^{n} 1^{n-k} a^k \in \mathcal{O}(a^n(1 + a)^n). \]

There exists a number of algorithms, both specialised and general, for solving \((d, l)\)-CSPs, which can be used in conjunction with Theorem 71 in order to solve \textsc{Max Hamming Distance} \((d, l)\)-CSP. For example, the \((d, 2)\)-CSP algorithms by Eppstein [51] and Feder & Motwani [54] could be used, as well as the general \((d, l)\)-CSP algorithm by Schöning [105]. Even though these algorithms are probabilistic, they are only applied once each time line 9 of Algorithm 21, thus the probability of error is not increased. Consequently, if the algorithm used on line 9 has a probability of error given by \(\varepsilon\), then so will Algorithm 21.

If we restrict ourselves to domains of size 2, i.e. the \textsc{Maximum Hamming Distance} \((2, l)\)-CSP problem (which coincidentally was the problem which was studied by Crescenzi & Rossi [42]) we can improve the algorithm further. Again, consider the following formula:

\[ (x \lor y) \land (\neg x \lor z) \]  (9.2)

If we wanted to find a pair of assignments agreeing on the values of, say, \(y\) and \(z\), and disagreeing on \(x\) using Algorithm 21, we would try to solve the following formula:

\[ (x \lor y) \land (\neg x \lor z) \land (x' \lor y) \land (\neg x' \lor z) \land (x \lor x') \land (\neg x \lor \neg x') \]

(since \(x \neq x'\) would become \((x \lor x') \land (\neg x \lor \neg x')\) when written as a 2-SAT formula.) There is, however, a better way of doing this. Since there is only two possible domain values, and we force \(x'\) to always assume a different value than \(x\), we are actually imposing the constraint \(x' = \neg x\). Consequently, there is no need to create new variables; instead, we only duplicate the clauses containing the variables on which the solutions should differ, and then replace every literal containing one of these variables with its negation. We could then transform (9.2) into

\[ (x \lor y) \land (\neg x \lor z) \land (\neg x \lor y) \land (\neg x \lor z) \]
and a satisfying assignment for this formula is \(\{x \mapsto 1, y \mapsto 1, z \mapsto 1\}\).

From this we can derive two assignments \(\mathbf{f}_1 = \{x \mapsto 1, y \mapsto 1, z \mapsto 1\}\) and \(\mathbf{f}_2 = \{x \mapsto 0, y \mapsto 1, z \mapsto 1\}\), both of which satisfy (9.2).

The result is Algorithm 22, and while it certainly bears a striking resemblance to Algorithm 21, it does not add any variables to the problem, and thus it is able to solve MAX HAMMING DISTANCE \((2,l)\)-CSPs a lot faster than the more general algorithm.

**Theorem 72.** If we can solve \((2,l)\)-CSP in \(\mathcal{O}(a^n)\) time, then there exists an algorithm for solving MAX HAMMING DISTANCE \((2,l)\)-CSP which runs in \(\mathcal{O}((2a)^n)\) time.

**Proof.** Algorithm 22 considers all subsets of variables of the problem, as was described earlier. Consequently, it will deliver a solution within \(\mathcal{O}(b^n)\) time, where

\[
b^n = \sum_{k=0}^{n} \binom{n}{k} a^n = a^n \sum_{k=0}^{n} \binom{n}{k} 1^k 1^{n-k} = (2a)^n.\]

Combining this theorem with previously known algorithms for solving \((2,l)\)-CSPs, we arrive at the following corollary:
Corollary 73. There exists

\[ a) \text{ a probabilistic } \mathcal{O} \left( (4 - 4/l + \varepsilon)^n \right), \varepsilon > 0, \text{ time algorithm, and} \]
\[ b) \text{ a deterministic } \mathcal{O} \left( (4 - 4/(l + 1))^n \right) \text{ time algorithm} \]

for solving MAX HAMMING DISTANCE \((2, l)\)-CSP. Additionally, for MAX HAMMING \((2, 3)\)-CSP, there exists

\[ c) \text{ a probabilistic } \mathcal{O} \left( 2.6604^n \right) \text{ time algorithm, and} \]
\[ d) \text{ a deterministic } \mathcal{O} \left( 2.9450^n \right) \text{ time algorithm.} \]

Proof. Combining Theorem 72 with algorithms by a) Schöning [105], b) Dantsin et al. [45], c) Hofmeister et al. [75], and d) Brueggemann & Kern [30] gives the result. \(\square\)

9.4.2 Max Hamming Distance \((d, 2)\)-CSP Algorithm

We saw in Section 9.1.3 how to apply the split and list method on the MAX VALUE problem, and the approach was quite similar to the one originally used by Williams [114]. In this section we will see a rather inventive way of using it when we construct a MAX HAMMING DISTANCE algorithm.

Starting with a MAX HAMMING DISTANCE \((d, 2)\)-CSP instance \(\Theta = (X, D, C)\), we begin, as usual, by splitting \(X\) into three parts, but rather than proceeding with listing the assignments, this time we first make copies of the partitions. Consequently, we have partitions \(X_1, X_2\) and \(X_3\) as well as partitions \(X'_1, X'_2\) and \(X'_3\), all of equal size, \(n/3\). Next, we list all possible assignments in each of these, getting lists \(L_1, L_2, L_3\) and \(L'_1, L'_2, L'_3\), all of size \(d^{n/3}\), and construct two graphs, \(G\) and \(G'\), with vertices \(V(G) = L_1 \cup L_2 \cup L_3\) and \(V(G') = L'_1 \cup L'_2 \cup L'_3\), and edge sets

\[ E(G) := \{ (u, v) \mid u, v \in V(G), u \in L_i, v \in L_j, i \neq j \} \]

and

\[ E(G') := \{ (u, v) \mid u, v \in V(G'), u \in L'_i, v \in L'_j, i \neq j \} \].
We now have two identical graphs, and it is time to join them together: Let $H$ be a graph with vertex set $V(G) \cup V(G')$ and edge set $E(G) \cup E(G')$. Additionally, for each $v \in L_i$ and $v' \in L'_j$, we add an edge $\{v, v'\}$ to $H$. Next we construct a weight function, which looks as follows: For $v \in L_i$ and $v' \in L'_j$

$$w(v, v') = \begin{cases} d_H(v, v') & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Thus for two assignments $v, v'$, if they have no variables in common, the weight of the edge between them is 0, but if they contain the same variables, their weight is defined as the hamming distance between the assignments. (See Fig. 9.5)

Now that we have the graph, complete with weights, we need to define what we are searching for. First of all, we need two triangles, one from $G$ and one from $G'$, in order to get a solution from each of them. Furthermore, we want the distance between them to be as large as possible, i.e. we want to find a complete subgraph consisting of 6 vertices with maximum weight on its edges.

**Lemma 74.** In the graph $H$, as described above, a clique of size 6 with edge weight $W$ corresponds to two satisfying assignments with hamming distance equal to $W$ in the original CSP.
Algorithm 23 Main algorithm for \textsc{Max Hamming Distance} \((2,2)\)-CSP.

\begin{algorithm}
\caption{$\text{MaxHamming}_1(f,G,\Theta)$}
\begin{algorithmic}[1]
\State 1. if $\delta(x) \in \{(3,1), (2,2), (2,1), (1,1)\}$ for all variables $x$ in $G$ then
\State 2. \quad return $\text{MaxHamming}_2(f,G,\Theta)$
\State 3. \quad end if
\State 4. Choose a variable $x$ in $G$ with $\delta(x) \in \{(\geq 3, \geq 2), (\geq 4, 1)\}$
\State 5. $(f_0, g_0) = \text{MaxHamming}_1(\alpha \cup \{x[0]\}, G - N_G(x[0]) - \{x[0]\}, \Theta)$
\State 6. $(f_1, g_1) = \text{MaxHamming}_1(\alpha \cup \{x[1]\}, G - N_G(x[1]) - \{x[1]\}, \Theta)$
\State 7. return $(f_i, g_i), i \in \{0,1\}$ maximising $d_H(f_i, g_i)$
\end{algorithmic}
\end{algorithm}

\textbf{Proof.} By construction, there are no edges within between any assignments in the same list $L_i$. Consequently, in order to find a clique of size 6, we need to pick one assignment from each of the lists; one from $L_1$, one from $L'_1$, etc. Now the edge weights in such a clique is, again by construction, 0 on edges between assignments not containing the same variables. Only on the edges between assignments containing the same variables is the weight non-zero. Consequently, there are three edges in the subgraph the sum of which give the sum of the entire clique, and this sum is the distance between the two assignments.

Unfortunately, the algorithm for finding cliques of size 6 (see Theorem 7) does not work for weighted graphs. However, in \cite{114}, a rather elaborate way of getting around this is described. While we cannot use this construction as-is, it gives some guidance when searching for one that works for us.

Assume we want to find a solution, i.e. a 6-clique, with weight $W$, and consider all triples $(t_{11}, t_{22}, t_{33})$, where $t_{ii} \in \{0,1,\ldots,W\}$ and $t_{11} + t_{22} + t_{33} = W$. For each tuple $t$, we construct a graph $H_t$ from the two graphs $G$ and $G'$, described earlier, by adding edges according to the following rule: For each $v \in L_i$ and $v' \in L'_i$, add the edge $\{v, v'\}$ if $w(v, v') = t_{ii}$. Let $\mathcal{H}_t$ be the set of all these graphs. Any clique of
size 6 in a graph in $\mathcal{H}_t$ will have weight $W$ by construction, and since the graph is unweighted and undirected, we can apply Theorem 7 to find such a 6-clique. Since we know $W \leq n$, it will take at most $n$ iterations to find the maximum $W$, and thus a solution to MAX HAMMING DISTANCE $(d, 2)$-CSP. Consequently, we get the following theorem:

**Theorem 75.** There exists a split and list algorithm for solving MAX HAMMING DISTANCE $(d, 2)$-CSP in $O\left(d^{2\omega n/3}\right) \in O\left(d^{1.584n}\right)$ time and $O\left(d^{n/3}\right)$ space.

**Proof.** The graph $H$ constructed earlier has $6 \cdot d^{n/3}$ vertices, giving the space requirement, and, from Theorem 7, we know that a clique of size 6 can be found in $O\left(|V(H)|^{2\omega}\right) = O\left(d^{2\omega n/3}\right)$ time. 

9.4.3 Max Hamming Distance $(2, 2)$-CSP Algorithm

In this section we will discuss and analyse our algorithm for MAX HAMMING DISTANCE $(2, 2)$-CSP. Since the formulae for the time complexity of the algorithm can be rather lengthy, the final step, that of calling a weighted 2-SAT solver for every leaf in the search tree, has been left out unless otherwise noted. Consequently, when, in the discussion, we say “$O(a^n)$,” this should be read as “$O(a^n \cdot b^n)$ where $O(b^n)$ is the running time of a weighted 2-SAT solver.”

Before we start the discussion of the algorithms, we will need some additional definitions.

The *degree* of a vertex $v$ in a graph, usually denoted $\text{deg}(v)$, is the size of its neighbourhood, i.e. $|N_G(v)|$. However, we are not really interested in the degree of a single vertex, but rather in the degrees of the two vertices that make up a variable. Thus let $\Theta = (X, D, C)$ be a $(2, 2)$-CSP and, for $x \in X$, define the variable degree $\delta(x)$ as a tuple $(\text{deg}(x[i]), \text{deg}(x[1 - i]))$, where $x[i]$ is the vertex with highest degree. If we are interested in variables with degrees higher than a certain number, we write $\delta(x) = (\geq i, \geq j)$.

We now focus our attention on the algorithm. The main algorithm, $\text{MaxHamming}_1$ (Algorithm 23), takes as input a partial assignment $f$, a microstructure graph $G$, and the original problem instance.
Algorithm 24 Helper function MaxHamming2.

MaxHamming2 (f, G, Θ)
1. if \( \delta(x) \in \{(2, 1), (1, 1)\} \) for all variables \( x \) in \( G \) then
2. return MaxHamming3(f, G, Θ)
3. end if
4. if \( G \) contains a cycle then
5. if all variables \( x \) has \( \delta(x) = (2, 2) \) in a cycle then
6. Choose \( x \) in this cycle
7. else if there is a variable \( z \) in a cycle s.t. \( \delta(z) = (2, 2) \) then
8. Choose \( x \) in a cycle s.t \( \delta(x) = (3, 1) \) and \( x[i] \) has a neighbour \( y \) with \( \delta(y) = (2, 2) \)
9. else
10. Choose \( x \) with \( \delta(x) = (3, 1) \) in a cycle
11. end if
12. else \% \( G \) is cycle-free
13. If possible, choose \( x \) two variables from the end of a chain, otherwise, choose \( x \) one variable from the end (of a chain.)
14. end if
15. \((f_0, g_0) = \text{MaxHamming2}(f \cup \{x[0]\}, G - N_G(x[0]) - \{x[0]\}, \Theta)\)
16. \((f_1, g_1) = \text{MaxHamming2}(f \cup \{x[1]\}, G - N_G(x[1]) - \{x[1]\}, \Theta)\)
17. return \((f_i, g_i), i \in \{0, 1\}\) maximising \( d_H(f_i, g_i) \)

\( \Theta \). If every variable in the microstructure is involved in less than 3 constraints, the helper function MaxHamming2, Algorithm 24, is called. In the graph, this translates to every variable \( x \) having \( \delta(x) \) in the set \( \{(3, 1), (2, 2), (2, 1), (1, 1)\} \). Otherwise, a variable without this property is chosen, and the algorithm branches on the two possible values. We note that for \( \delta(x) = (3, 2) \), there will be at least 3 variables less in one branch and 2 variables less in the second branch, and for \( \delta(x) = (4, 1) \), there are at least 4 and 1 variables less, respectively. Thus if we let \( T_{(3,2)}(n) \) and \( T_{(4,1)}(n) \) denote the time complexities of the respective cases, then they are described by the following two
Algorithm 25 Helper function $MaxHamming_3$.

$MaxHamming_3 (f,G,\Theta)$
1. Let $w$ be a vector of weights, initially all set to 0
2. for each $x[i] \in f$ do
3.   add weight $w(x[1-i]) := 1$
4. for each connected component of $G$ do
5.   Add weights to $w$, as shown in Fig. 9.7.
6. $(g,W) := 2-SAT_w(\Theta,w)$
7. for each variable $x$ in $G$ do
8.   if $x[i]$ in $g$ then
9.     If possible, add $x[1-i]$ to $f$, otherwise add $x[i]$.
10. end if
11. end for
12. return $(f,g)$

recurrences:

$T_{(3,2)}(n) = T(n-3) + T(n-2) + p(n)$

and

$T_{(4,1)}(n) = T(n-4) + T(n-1) + p'(n)$

where $p(n)$ and $p'(n)$ are some polynomials. Thus the two cases have running times of $T_{(3,2)} \in O(\tau(3,2)^n)$ and $T_{(4,1)} \in O(\tau(4,1)^n)$, respectively. Of these two, the latter grows faster, and will dominate the time complexity.

The first helper function, $MaxHamming_2$ (Algorithm 24), takes over when no variable is involved in more than 2 constraints. Apart from Lines 1 to 3, which we will come back to later, the algorithm starts with checking for cycles. If there is a cycle in the graph, we need to break it, and this is done on lines 4 to 11. First of all, if there is a cycle where every variable has a degree of $(2,2)$, then selecting one value for a variable in this cycle will propagate through the entire
9.4. The Maximum Hamming Distance Problem

Figure 9.6: Branching on $x[i]$ will remove the shaded values and force the black values.

cycle, as is shown in the upper portion of Fig. 9.6. On line 8, by choosing a variable $x$ with $\delta(x) = (3, 1)$ with a neighbour $y$ with $\delta(y) = (2, 2)$, one of the values for $x$ will propagate to $y$. (See lower part of Fig. 9.6.) Consequently, 4 variables are removed in one branch, and one in the other, giving a running time of $O(\tau(4, 1)^n)$ for this case. The obvious exception to this case is when the cycle contains only 3 variables, as is shown in Fig. 9.8a. Note that the coloured vertex $x[i]$ is the only possible choice—the other assignment would lead to an inconsistency.

Now if every variable $x$ in the cycle has $\delta(x) = (3, 1)$, we get a number of different possibilities, but before we discuss them, we need to make some observations. The first one is that once a variable has no neighbours, see Fig. 9.7, we can choose any of the two values for it. Additionally, we get a similar situation when a component has been reduced to two variables with one edge between them, a “hurdle.” This means that when a component of $(3, 1)$ variables has at most 3 variables, as in Fig. 9.8b, when choosing such a variable, in effect, the entire component is removed from the problem and need
no longer be considered — in one case we get a unique assignment for the remaining (black) vertices in the other case we get a hurdle. If there are no cycles in the component, e.g. we have a “comb-like” structure (see Fig. 9.9), then choosing any of the three variables to branch on will, again, remove the entire component, giving a running time of $O(\tau(3, 3)^n)$. This also holds for cycle-free components of size 4 and 5. When there are more than 5 variables in the component, by choosing a variable which is two variables removed from the end of the comb (marked in Fig. 9.9), the chain is broken and we remove 3 variables in one branch and 4 in the other. As was seen in the case for cycles where all variables have degree (2, 2), the number of removed variables increases if a neighbour of the branching variable has this property. Consequently, we will focus on the combs and merely note that the time complexity will not be worse if we have more variables with degree (2, 2).

Getting back to discussing cycles; when we reach line 10 of algorithm MaxHamming_3, every cycle consists exclusively of variables with degree (3, 1), and since no vertex in the graph has degree higher than 3, there can be at most one cycle in a component. The case with cycles containing 3 variables was discussed earlier, and for the case with 4 variables we get one branch where the entire component is removed, and one where we get a comb with 3 variables, which can be removed in its entirety when we branch.

There can be no more than $n/4$ cycles with 4 variables in the graph at this point. For each of these cycles, we choose one variable to branch on, and in one branch the entire component is removed,
while in the other, we get a component with 3 variables. Since we want to look at all of these cycles, and both branches, this is equivalent to selecting $k$ cycles where we remove the entire component, and then examine the remaining $n/4 - k$ components. In other words, it will require

$$\sum_{k=0}^{n/4} \binom{n/4}{k} \left( 1^k \cdot \tau(3,3)^{3(n/4-k)} \right)$$

steps to examine all the cycles. Using the binomial theorem, this can be simplified to $(1 + \tau(3,3)^3)^{n/4}$.

For cycles with 5 variables, the situation is similar, but for 6 we no longer remove the entire component in one of the branches. Instead, we get one branch with 5 variables, and one with 3, which, using the same reasoning as above, gives

$$\sum_{k=0}^{n/6} \binom{n/4}{k} \left( \tau(3,3)^{3k} \cdot \tau(5,5)^{5(n/6-k)} \right) = (\tau(3,3)^3 + \tau(5,5)^5)^{n/6}.$$ 

Similarly, for cycles of length 7, we get $(\tau(4,4)^4 + \tau(3,4)^6)^{n/7}$. In general, if we have cycles of length $c$, one branch will have one variable less, and the other three variables less, giving the following general running time:

$$\sum_{k=0}^{n/c} \binom{n/c}{k} \left( \tau(3,4)^{(c-3)k} \cdot \tau(3,4)^{(c-1)(n/c-k)} \right) = \left( \tau(3,4)^{c-1} + \tau(3,4)^{(c-3)n/c} \right)^{n/c} < (2\tau(3,4)^c)^{n/c} = (2^{1/c}\tau(3,4))^n$$

Finally, when algorithm $\text{MaxHamming}_3$ (Algorithm 25) is called, the graph $G$ only contains variables involved in zero or one constraint, i.e. every variable will be of one of the forms found in Fig. 9.7. The weights shown in the figure is now added to the corresponding assignments in the original problem, $\Theta$, and the resulting weighted 2-SAT problem is given to a 2-$\text{SAT}_w$ solver. If the solution $g$ returned by the solver has weight $W$, this means that we can add assignments (i.e.
vertices) to \( f \) to create a solution which differs from \( g \) on \( W \) assignments. First of all, since all assignments in \( f \) are given weight 0, if any of these are chosen, they will not add anything to the distance, while the other possible value for all these variables will add one to the distance (and are consequently given a weight of 1 on line 3.) For the free variables in \( G \), i.e. all vertices \( x \) with \( \delta(x) = (1, 1) \), we can choose freely which value they should assume, and thus we can always find an assignment which adds one to the distance from \( g \) by choosing the other value for \( f \). The remaining components then consist of pairs of variables with one edge between them, i.e. hurdles. If \( g \) contains both assignments with weight 1/2, then obviously, we have to add one of them to \( f \), since not both assignments with weight 3/2 are allowed simultaneously — and thus we get a distance of 1, which is the sum of the weights in \( g \). On the other hand, if \( g \) contains one 3/2 and one 1/2 assignments, then we can choose the opposing value for both of these and get a distance of 2. Consequently, the pair returned on line 12 will have a Hamming distance equal to the weight of \( g \), and with \( f \) and \( G \) given, no pair with greater Hamming distance can exist.

Except for the call to 2-SAT\(_w\) on line 6, every step of Algorithm 25 can be carried out in polynomial time, thus the time complexity is fully determined by that of the 2-SAT\(_w\) algorithm.

**Figure 9.8:** Cycle with 3 variables.
9.4. The Maximum Hamming Distance Problem

Figure 9.9: Variable in a comb with more than 4 variables.

To summarise this section, we state the following theorem:

**Theorem 76.** Algorithm MH correctly solves MAX HAMMING DISTANCE $(2, 2)$-CSP and has a running time of $O((a \cdot 1.3803)^n)$, where $n$ is the number of variables in the problem, and $O(a^n)$ is the time complexity of solving a weighted 2-SAT problem.

*Proof.* The correctness follows from the previous discussion. Among the steps in the algorithm, $O(\tau(4,1)^n) \in O(1.3803^n)$ dominates, and since the 2-SAT$_w$ algorithm is called for every leaf in the search tree, we get a total time complexity of $O((a \cdot 1.3803)^n)$.

**Corollary 77.** MAX HAMMING DISTANCE $(2, 2)$-CSP can be solved in time $O(1.7338^n)$.

*Proof.* Dahllöf et al. [44] presents an algorithm for solving weighted 2-SAT in $O(1.2561^n)$ time, and this together with Theorem 76 gives the result.
Chapter 10

Future Work

Carefully study the well-being of your men, and do not overtax them. Concentrate your energy and hoard your strength. Keep your army continually on the move, and devise unfathomable plans.

Sun Tzu, The Art of War

There are, of course, an abundance of open questions regarding the applications. As was mentioned in Chapter 6, none of the algorithms we discuss have been implemented, thus it remains to be seen exactly how successful they would be in practice. Apart from this, however, there are a number of topics that could benefit from further study.

Reuse. A theme common to all our work is the reuse of existing algorithms. Sometimes we have used algorithms designed to solve restricted versions of the problem we want to solve, for example the use of #2- and #3-colouring algorithms to solve the general #k-colouring problem, and sometimes the algorithms have been for problems which are only distantly related, such as using an algorithm for weighted 2-SAT when solving the MAX HAMMING DISTANCE problem. The key here is of course the methods, and how we apply them. The partitioning and covering methods let us focus on special, restricted cases of a problem, leaving the generalisation to the methods, and microstructures allow us to view almost any combinatorial problem as some kind
of independent set problem. The #2-SAT algorithm we have used in our algorithms, recently published in [44], is based on an extensive case analysis. It is very likely that the bounds on the running time can be squeezed even more through further analysis, thus it deserves additional study.

Quite recently Fürer & Kasiviswanathan [59] suggested improved algorithms for weighted #2SAT which runs in $O(1.2461^n)$ time, and for #3-COL, which runs in $O(1.7702^n)$. While these results alone would give us improved algorithms for several of the problems we discuss in this thesis — for example lowering the running time of the MAX HAMMING DISTANCE (2, 2)-CSP algorithm from $O(1.7338^n)$ to $O(1.7200^n)$ — it is even more interesting to note that the paper includes an explicit application of the partitioning method described in Chapter 5. Using the improved algorithms for #2SAT and #3-COL, the partitioning method is applied to get improvements in the algorithms for $(d, 2)$-CSP and $(k)$-COL. (The results has not yet been subject to peer-review, thus we have not included them in the discussion, but, nevertheless, it is an interesting development.)

**Derandomisation.** One of the driving forces behind much of our research has been the derandomisation of successful CSP algorithms. As it happened, our results turned out to be useful in a number of different areas, but nonetheless, derandomisation was always present in the background. Consequently, one could say we have failed miserably, since we have only managed to derandomise for domains of sizes 2 to 10, inclusive; the algorithm of Feder & Motwani [54] has so far withstood every attempt at derandomisation. The methods we developed cannot handle this kind of algorithm, and while we have made some preliminary work on it, we are still far from a derandomisation. With Eppstein’s algorithms, there is a deterministic ‘base case’ to which we can apply the covering method (say) and thus get a deterministic algorithm for any domain size. Unfortunately, there is no such base case in the algorithm by Feder & Motwani, so some other approach, quite probably very different, is needed. Of course, saying we have failed might be stretching it a bit. After all, as a result of
our work we now have deterministic algorithms for domains of size 2 to 10, inclusive, so even though it was not a roaring success, it is a rather good start.

$k$-colourability. The $k$-colourability problem suggests another open area for research. Why is it that for $k \geq 7$, the general Chromatic Number algorithm remains the fastest algorithm for determining $k$-colourability? Other than the obvious lack of success by the research community in finding a faster algorithm for, say 7-colourability, there exists no evidence to suggest that the number 7 is anything but arbitrary. We give a polynomial space algorithm for this problem, but since it has a running time which is a function of $k$, it cannot compete with the general algorithm. Consequently, this remains a most interesting open problem.

Other problems. One of the main reasons for studying the Max Hamming Distance problem was that neither the covering nor the partitioning method were applicable to it. For this problem, the reason was quite obvious; both our methods are based on the idea of taking small chunks of the domain and considering them in isolation. This is not possible for the Max Hamming Distance problem since we have to build two different solutions simultaneously — but they are not independent of each other. Consequently, it would be interesting to see if there are other problems where we cannot successfully apply our methods, and, if possible, characterise them.

The work we have done on the Max Ind problem is, as far as we know, the first of its kind. Since we have so far restricted our study to the case with binary constraints, this leaves the field wide open for further research. Our algorithm is based on analysis on the microstructure of the problem, which in its current form is limited to such constraints. Consequently, one possible extension would be to generalise the microstructure and try to adapt our work for this setting. If not, then a completely new approach will have to be applied.

Looking at a more detailed level, some of the algorithms we present, notably the Max Hamming $(2,2)$-CSP and Max Value $(3,2)$-CSP
algorithms, are based on a case analysis. With an open-ended activity such as research, one always reaches a point where it has to be decided whether to push on or to draw the line. While we managed to achieve quite good improvements on both algorithm during our investigations, it is almost certain that further analysis of them would yield even faster algorithms. Apart from the improvement in efficiency, it is conceivable, even likely, that as the work progresses, some further insight into how microstructures work and how they can be exploited could be gained. Indeed, much of what we discovered in one problem carried over to other problems with similar structure.
Bibliography


[37] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Reg-


[45] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2 - 2/(k+1))^n$ algo-


[56] Richard P. Feynman. There’s plenty of room at the bottom, December 1959. *Talk given at the annual meeting of the American Physical Society at the California Institute of Technology (Caltech).*


[79] Philippe Jégou. Decomposition of domains based on the microstructure of finite constraint-satisfaction problems. In Proceed-


<table>
<thead>
<tr>
<th>No</th>
<th>Title</th>
<th>Author</th>
<th>ISBN</th>
</tr>
</thead>
</table>
No 774  

No 779  

No 793  

No 785  

No 800  

No 808  

No 821  

No 823  

No 828  

No 833  

No 852  

No 867  

No 872  

No 887  

No 889  

No 893  

No 910  

No 918  

No 920  

No 929  

No 933  

No 937  

No 938  

No 945  

No 946  

No 947  


