Exact Exponential-Time Algorithms for Domination Problems in Graphs

Johan M. M. van Rooij

Johan M. M. van Rooij

Exact Exponential-Time Algorithms for Domination Problems in Graphs

ISBN: 978-90-8891-293-1

jmmrooij@gmail.com

Cover design: Proefschriftmaken.nl || Printyourthesis.com
Printed by: Proefschriftmaken.nl || Printyourthesis.com
Published by: Uitgeverij BOXPress, Oisterwijk

Exact Exponential-Time Algorithms for Domination Problems in Graphs

Exacte Exponentiële-Tijd Algoritmen voor Domineringsproblemen in Grafen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof. dr. G. J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op vrijdag 24 juni 2011 des middags te 12.45 uur

door

Johannes Matheus Maria van Rooij

geboren op 19 juni 1983
te Schaijk

Promotor: Prof. dr. J. van Leeuwen
Co-promotor: Dr. H. L. Bodlaender

Preface

What lies before you is the result of a four year period of PhD research. It is the product of creative ideas, hard labour, and many discussions and collaborations with other researchers and students. It also is a result of a process that includes many moments of joy, many moments of frustration, and many notable moments of fascination on the beauty of mathematics and computer science in general and algorithmic research in particular. But, maybe most of all, this thesis is the final product of a four year period of personal growth.

In this four year period, I explored a number of areas of research, the larger part of which was related to the topic of this thesis. Nonetheless, I have also performed research on slightly different topics. Some of this research led to concrete results and published papers as well. I would like to mention the research I did on role assignment problems together with Daniël Paulusma and Pim van 't Hof [311, 312], the joint results with Hans L. Bodlaender on a subexponential-time algorithm for intervalising coloured graphs [51], and the fruitful collaboration with Jesper Nederlof, Marek Cygan, Marcin Pilipczuk, Michał Pilipczuk and Jacub Onufry Wojtaszczyk on singly-exponential-time algorithms for connectivity problems on tree decompositions [88].

The research in this thesis was done while I was a PhD student at the Department of Information and Computing Sciences of Utrecht University under the guidance of Hans L. Bodlaender and Jan van Leeuwen. While most of this research was done in the department, many ideas also originated from my visits to other universities: my visit to the Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la DEcision of the Université Paris Dauphine, my visit to the School of Engineering and Computing Sciences of Durham University, and my visit to the Department of Informatics of the University of Bergen. Also, attending various conferences had a major impact on my research. Noteworthy are IWPEC 2008, ESA 2009, IPEC 2010, and Dagstuhl Seminars 08431 and 10441.

Acknowledgements. First and foremost, I would like to thank my advisor Hans Bodlaender, whom I experienced to be a great advisor. While Hans gave me a lot of freedom to pursue my own research, his door was nearly always open to me for discussions and conversations, both on research and on private matters. His enthusiasm and understanding have been of great value to me and are highly appreciated.

The next person who has most probably had the most influence on this thesis and whom I would like to thank next is Jesper Nederlof. Our collaboration, which started while he was a Master student supervised by Hans and me, has both been very enjoyable and very fruitful: it resulted in Chapters 8 and 9 (and [88]), and this research also planted the seeds in my mind that led to many of the results in Chapters 10-13.

I would also like to thank my promotor Jan van Leeuwen. His guidance on the

writing of this thesis has been very valuable; I learned much from his many comments, both on writing in general and on relating my research to the field in particular.

Furthermore, I thank my office mate Thomas van Dijk. Reflecting on research together has been very helpful, our joined work has been enjoyable, and, maybe most of all, his continued assistance in C++ programming has been invaluable in my first year as a PhD student.

The research presented in this thesis could not have taken place, and would not have been as enjoyable, without all the people I worked with. First of all, this includes my coauthors Hans Bodlaender, Nicolas Bourgeois, Marek Cygan, Bruno Escoffier, Jesper Nederlof, Vangelis Paschos, Daniël Paulusma, Marcin Pilipczuk, Michał Pilipczuk, Peter Rossmanith, Thomas van Dijk, Marcel van Kooten Niekerk, Erik Jan van Leeuwen, Pim van 't Hof, Martin Vatshelle, and Jacub Onufry Wojtaszczyk. Secondly, this includes many other people with whom I discussed research including Andreas Björklund, Henning Fernau, Fedor Fomin, Thore Husfeldt, Dieter Kratsch, Daniel Lokshtanov, Matthias Mnich, Igor Razgon, Saket Saurabh, and Jan Arne Telle. Of these people, I want to express my gratitude to Vangelis Paschos, Daniël Paulusma, and Jan Arne Telle for inviting me to visit their respective research groups.

My research benefited from the nice working environment in the Algorithmic Systems group of the department. I would like to thank the members of the group during this four year period. This includes Jan van Leeuwen, Hans Bodlaender, Han Hoogeveen, Marjan van den Akker, Gerard Tel, Marinus Veldhorst, Stefan Kratsch, Bart Jansen, Eelko Penninkx, and Guido Diepen. In particular, I enjoyed the discussions about politics with Han Hoogeveen and Hans Bodlaender, and the many friendly conversations at less productive moments with Marjan van den Akker and Thomas van Dijk. I would also like to thank the support staff, in particular Wilke Schram and Edith Stap, for helping me find my way in the department and their contribution to a positive working atmosphere. I would like to thank Stefan Kratsch and Erik Jan van Leeuwen for their help in setting up various parts of the LaTeX environment used to write this thesis.

To perform good research, I believe that it is important to be physically fit, and that it is even more important to have a place to relax and have fun. I would like to thank all my friends in the badminton club 'S. B. Helios' in Utrecht, in particular, the members of the teams I played in and everybody who organised various activities for this wonderful club with me, including some great tournaments. I would also like to thank my friends from my other badminton club 'DVS-Koto Misi'.

There are a few more people that I would like to thank for making the past four years a pleasant time. Firstly, the members of the 'politieke commissie' of the 'Jonge Democraten' afdeling Utrecht. I enjoyed thinking about topics related to society and politics, and I enjoyed the things we worked on together and the fun we had in the process. Secondly, my fellow residents of 'Oudwijk 21' who made sure that there was always some fun after a day of hard work. Thirdly, all my other friends, especially Daniël Heijnen; your support has been of great value to me.

Also, a word of thanks to my family, especially my two brothers who have always been there for me for support.

Last, but certainly not least, I would like to thank Lizette, whose support and comfort mean at lot to me.

Contents

1 Introduction

The famous *five queens puzzle* and *eight queens puzzle* are chess puzzles that are played on the usual 8-by-8 chessboard and that involve the placement of queens. According to the rules of chess, a queen attacks any square that it can reach by moving any number of squares in a horizontal, vertical, or diagonal direction, without passing over another piece. The objective of the five queens puzzle is to place five queens on the chessboard that together attack or occupy all squares of the chessboard, and the objective of the eight queens puzzle is to place eight queens on the chessboard such that no two queens attack each other. For possible solutions to these puzzles, see Figure 1.1.

The puzzles can be said to be the origin of the study of graph domination problems (see also [179]). They were introduced by chess enthusiasts in the 1850s, and many mathematicians, including C. F. Gauss (see [174]), have worked on these puzzles and

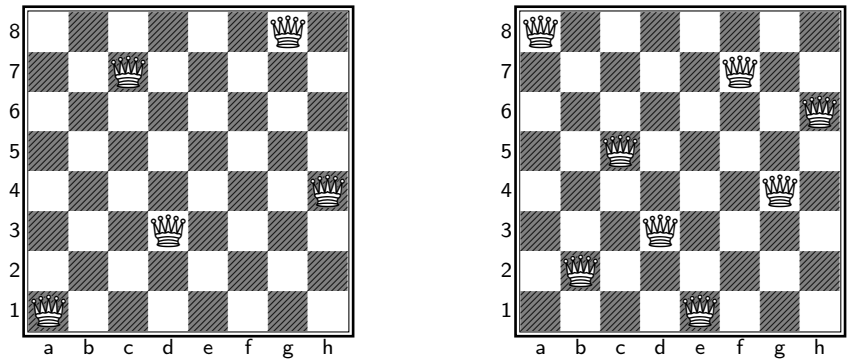


Figure 1.1. Solutions to the five queens puzzle and the eight queens puzzle.

their generalisations. In computer science, the puzzles have received much attention as well. For example, the famous Dutch Turing Award winner E. W. Dijkstra used the eight queens puzzle in 1972 to illustrate the power of structured programming [92]. He published a highly detailed description of the development of a backtracking algorithm for this puzzle. We will also present many backtracking algorithms in this PhD thesis, some solving generalisations of the two puzzles.

To exhibit the relation of the puzzles to graphs, we consider the queens graph of a chessboard. Note that an undirected *graph* $G = (V, E)$ consists of a set V, whose elements are called *vertices*, and a set E of unordered pairs of vertices from V, which are called *edges*. The *queens graph* of the usual 8-by-8 chessboard is the graph that has the 64 squares of the chessboard as vertices, and that contains an edge between a pair of distinct squares if and only if a queen can attack one square of the pair from the other. A solution to the five queens puzzle corresponds to a set D of five vertices in the queens graph G with the property that every vertex in V is either in D or connected to some vertex in D by an edge: the vertices in D correspond to the squares on which the queens are placed. Any square corresponding to a vertex that is not in D is being attacked by a queen since there exists an edge between the vertex and a vertex in D. Similarly, a solution to the eight queens puzzle corresponds to a set of eight vertices I in the queens graph G with the property that there is no edge between any two vertices in I: since there are no edges between the vertices in I, no two queens placed on the corresponding squares can attack each other.

In this PhD thesis, we will consider the problem of computing vertex sets similar to the solutions to the queens puzzles in arbitrary graphs. A set of vertices D in a graph G such that every vertex in V is either in D or adjacent to some vertex in D is called a *dominating set*, and a set of vertices I in a graph G such that there is no edge between any pair of vertices from I is called an *independent set*. The computational problem of computing a dominating set of minimum size is known as Dominating Set and the computational problem of computing an independent set of maximum size is known as Independent Set. Notice that the five and eight queens puzzles are special cases of these problems. We will focus on solving these, and other graph domination problems, on graphs in general[1] using algorithms that run in exponential time (unless we can do better). Our goal is to design such algorithms for which we can prove fast worst-case running times.

1.1. Exact Exponential-Time Algorithms

An *algorithm* is a step-by-step approach to solve a computational problem. The number of computation steps that are required to execute an algorithm is called the algorithm's *running time*. In algorithmic research, one often gives algorithms for which one can prove that the running time is below some guaranteed asymptotic bound. The goal is to give algorithms for which this asymptotic bound is as small as possible.

This asymptotic behaviour can have many forms. If we let n be a complexity parameter for a problem, for example the number of vertices of an input graph (see

[1] For simple exact exponential-time algorithms that solve the queens puzzles directly, i.e., without considering only the problem on the underlying graph, we refer the reader to a nice paper by Fernau [132].

Section 3.1 in Chapter 3 for the definition of complexity parameters), then we typically distinguish between the following types of running times for algorithms (partially based on [321]):

- Polynomial in n. For example: n^2, n^3, $n \log n$, n^{1000}.
- Quasi-polynomial in n. For example: $n^{\log n}$, $n^{\log^2 n}$, $2^{\log^3 n}$.
- Subexponential in n. For example: $2^{\sqrt{n}}$, $3^{n^{0.99}}$, or less than $2^{\epsilon n}$ for all $\epsilon > 0$.
- Exponential in n. For example: 2^n, 1.4969^n, $n!$, n^n.

For many computational problems, algorithms with a relatively fast type of asymptotic running time, e.g., algorithms with polynomial running times, are known. However, many other computational problems seem to be solvable only by algorithms with a slower type of running time, e.g., algorithms with exponential running times.

In this PhD thesis, we will mainly consider *exponential-time algorithms*. More specifically, the algorithms that we present will mostly have *singly-exponential* running times: running times of the form c^n. We refer to the constant c in running times of this form as the base of the exponent of the running time. Singly-exponential running times are the asymptotically fastest running times that are of the type exponential running time in the classification above. We will give algorithms with singly-exponential running times for problems for which most likely no algorithms with running times of a faster type (such as subexponential-time algorithms) exist.

We study algorithms with 'small' asymptotic running times. For singly-exponential-time algorithms, this means that we try to minimise the base of the exponent of the running time. A smaller base of the exponent can lead to much faster algorithms on instances of moderate size: see Table 1.1 and, for example, compare 2^n with 1.4969^n or even 1.0222^n. For practical situations, it is often more important to consider the maximum size of an instance that can be tackled within a given time frame, for example in a day. One can see from Table 1.1 that, using a 2^n-time algorithm, one can tackle instances with n up to roughly 50 in a day, while if we use a 1.4969^n-time algorithm, then this becomes roughly 90. Using an algorithm with a running time that has a smaller base of the exponent allows us to tackle problems that are a multiplicative factor larger, while using a twice-as-fast computer allows us only to tackle problems that are an additive factor larger.

In this PhD thesis, we care little for the polynomial factors involved in the exponential running times as they are always asymptotically dominated by any improvement in the base of the exponent of the exponential factor; for example, compare $n^2 1.7^n$ with 2^n in Table 1.1. However, as one can also see from the same comparison, polynomial factors are of great importance in practical situations. Although most of our algorithms can be used to solve the studied problems in practice, we will focus on the theoretical study of exponential-time algorithms and their asymptotic running times.

In the field of exact exponential-time algorithms, one aims at the best possible asymptotic running time. In other fields of algorithm design, one sometimes trades running time for other resources or properties of the algorithm. Examples of this include *heuristic algorithms*, where one has no a priori guarantee on the quality of a solution, and *approximation algorithms*, where the quality of the solution is guaranteed to be at most some constant times the exact optimal solution. Other approaches include randomised algorithms, the study of average-case complexity, or parameterised algorithms.

This table gives the number of operations and the associated execution times corresponding to different running times. The numbers given without brackets are the number of operations, and the given running times between brackets are the associated execution times based on a standard of 10^{11} operations per second; this equals the number of FLOPS (FLoating point OPerations per Second) of the Intel Core i7 980 XE processor at peak performance.

Behaviour	$n = 10$	$n = 30$	$n = 50$	$n = 70$	$n = 90$
n	10 $(< 1\mu s)$	30 $(< 1\mu s)$	50 $(< 1\mu s)$	70 $(< 1\mu s)$	90 $(< 1\mu s)$
n^4	10^4 $(< 1\mu s)$	10^6 $(8\mu s)$	10^7 $(63\mu s)$	10^7 $(0.2ms)$	10^8 $(0.7ms)$
1.02220^n	1.2 $(< 1\mu s)$	1.9 $(< 1\mu s)$	3 $(< 1\mu s)$	4.6 $(< 1\mu s)$	7.2 $(< 1\mu s)$
1.08537^n	2.3 $(< 1\mu s)$	12 $(< 1\mu s)$	60 $(< 1\mu s)$	309 $(< 1\mu s)$	10^3 $(< 1\mu s)$
1.4969^n	56 $(< 1\mu s)$	10^5 $(1.8\mu s)$	10^9 $(6ms)$	10^{12} $(18s)$	10^{16} $(16h)$
$n^2 1.7^n$	10^4 $(< 1\mu s)$	10^{10} $(74ms)$	10^{15} $(2h)$	10^{20} $(21y)$	10^{24} $(10^6 y)$
2^n	10^3 $(< 1\mu s)$	10^9 $(11ms)$	10^{15} $(3h)$	10^{21} $(374y)$	10^{27} $(10^8 y)$
$n!$	10^6 $(36\mu s)$	10^{32} $(10^{14} y)$	10^{64} $(10^{45} y)$	10^{100} $(10^{81} y)$	10^{138} $(10^{119} y)$

Some of the given running times are based on algorithms in this PhD thesis. For example, in Chapter 4 we give an $\mathcal{O}(1.02220^n)$-time algorithm, in Chapter 5 an $\mathcal{O}(1.4969^n)$-time algorithm, and in Chapter 7 an $\mathcal{O}(1.08537^n)$-time algorithm. We note that, in this table, the running times are used exactly, that is, without additional constant or polynomial factors that are suppressed in the notation.

Table 1.1. Different running times and the corresponding number of operations and execution times for different instance sizes.

Parameterised algorithms are closely related to exact exponential-time algorithms. The difference lies in the (complexity) parameters that are used in the analysis of the running times of the algorithms. There are roughly three kinds of parameter that are used for measuring the running time of an algorithm.

1. Parameters based on a size measure of the search space of the problem, for example, the number of squares n of an input chessboard in a queens puzzle.
2. Parameters based on the solution size of the required solution, for example, the number of queens k in a solution to a queens puzzle.
3. Parameters based on some other property of the input, for example, the treewidth of the input graph, or the size of the minimum vertex cover of the input graph[2].

Algorithms that give exact solutions to problems and whose analysis is based on a parameter of the first type are known as *exact algorithms*; if their analysis is based on a parameter of the second or third type, then they are known as *parameterised algorithms*. For the third type of algorithms, one can distinguish between the kind of parameter that is being used. If a parameter is based on a graph decomposition, for example treewidth, we call the algorithms *graph-decomposition-based algorithms*.

In this PhD thesis, we will mainly study exact exponential-time algorithms and graph-decomposition-based algorithms. Our motivation for also studying graph-decomposition-based algorithms comes from the fact that, for some graph decomposition parameters, improvements on the decomposition-based algorithms directly relate to improvements on exact algorithms. An example of such a graph decomposition parameter is the treewidth of the input graph. Exact exponential-time algorithms using

[2]See Section 2.2.2 for the definition of treewidth, and Section 1.6 for the definition of a vertex cover.

treewidth-based algorithms can, for example, be found in Section 2.2.2 in Chapter 2, or in Chapters 8-10.

1.2. A Short History of Exponential-Time Algorithms

Although the study of exact exponential-time algorithms was initiated in the nineteen sixties and nineteen seventies, the area remained a small subfield of theoretical computer science for a long time. Important and well-known early results from this period include the following[3]:

1. An $\mathcal{O}^*(2^n)$-time[4] algorithm for Travelling Salesman Problem by Held and Karp [180] (1962) and independently by Bellman [17] (1962).
2. An $\mathcal{O}(1.2852^n)$-time algorithm for Independent Set by Tarjan [291] (1972). This was later improved to $\mathcal{O}(1.2599^n)$ by Tarjan and Trojanowski [292] (1977).
3. An $\mathcal{O}(1.4143^n)$-time algorithm for Subset Sum and Binary Knapsack by Horowitz and Sahni [186] (1974).
4. An $\mathcal{O}(2.4423^n)$-time algorithm for Graph Colouring and an $\mathcal{O}(1.4423^n)$ time algorithm for 3-Colouring by Lawler [220] (1976).
5. An $\mathcal{O}^*(2^n)$-time algorithm for Hamiltonian Cycle by Kohn et al. [207] (1977). This algorithm was later rediscovered by Karp [200] in 1982, and by Bax [14] in 1993. This result improves the $\mathcal{O}^*(2^n)$ for Travelling Salesman Problem for the unweighted version of the problem in the sense that the algorithm of (1.) uses $\mathcal{O}^*(2^n)$ space while this algorithm uses only polynomial space.

Only a handful of results were obtained in the nineteen eighties (e.g., [195, 200, 239, 270, 283]) and the early nineteen nineties (e.g., [14, 16, 169, 272, 277, 278, 287, 328]).

In the late nineteen nineties, the field has grown tremendously. There are a number of reasons for this increased interest. First of all, it is now widely believed that no polynomial-time algorithms exist for many key problems; therefore, one needs to consider the best possible super-polynomial or even exponential-time algorithms when the need arises to solve these problems. Secondly, with the increase of the computational power of modern computers, exponential-time algorithms can be practicable for solving moderate-size instances. This is especially useful on problems where other approaches do not perform satisfactory, e.g., in applications that require exact solutions to a given problem, or on many problems like Independent Set and Dominating Set that are hard to approximate [127, 178, 329] and for which most likely no parameterised algorithms exist [116].

Finally, the interest for exact exponential-time algorithms can also be explained from the fact that, from the scientific point of view, the field has many interesting unsolved problems. In particular, some hard problems seem to have faster exponential-time algorithms than others, while classical complexity theory cannot explain these differences. The study of whether worst-case running times for various problems are related, and whether progress on different problems is connected, has begun only very recently. Also, in many cases, we have little idea to how close known algorithms are to the best possible.

[3]For definitions of the mentioned problems, see the list of problems in Appendix B.

[4]The $\mathcal{O}^*$-notation is introduced in Section 1.6.

Currently the field of exact exponential-time algorithms is very active. This can be observed from the many recent surveys, see [99, 142, 191, 282, 319, 320, 321], and PhD theses, see [8, 24, 53, 66, 171, 223, 236, 264, 288, 315, 318], written on the topic. We note that the thesis of Liedloff also studies domination problems in graphs [223]. Recently, Fomin and Kratsch published the first comprehensive book dedicated to the field of exact exponential-time algorithms [149].

Over the last fifteen years, many new techniques have been developed or introduced to the field. Prominent newly-developed techniques are the *measure and conquer* technique due to Fomin et al. [144] and the invention of the *fast subset convolution* algorithm by Björklund et al. [28]. Examples of very successful introductions of known techniques to the field of exact exponential-time algorithms are the use of *inclusion/exclusion* by Björklund et al. [27, 33] and the use of *treewidth-based algorithms* by various authors [138, 147, 204]. See Chapter 2 for an introduction to all these approaches except for fast subset convolution. More on fast subset convolution can be found in Chapter 11.

1.3. Domination Problems in Graphs

This PhD thesis deals with exact exponential-time algorithms for graph domination problems, see e.g. the monograph 'Fundamentals of Domination in Graphs' by Haynes, Hedetniemi, and Slater [179]. Graph domination problems are an important class of combinatorial problems with many practical and theoretical applications. They model many relevant problems in fields such as optimisation, communication networks and network design, social network theory, computational complexity, and algorithm design. Many facility location, resource allocation, and scheduling problems, are variants of graph domination problems.

Let us start by formally introducing a prominent graph domination problem: the Dominating Set problem. A subset $D \subseteq V$ of the set of vertices V in a graph G is called a *dominating set* if every vertex $v \in V$ is either in D or adjacent to some vertex in D.

Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set $D \subseteq V$ in G of size at most k?

We will consider algorithms for this problem in Chapters 5, 8, 11, 12, and 13.

Next, we will introduce some of the most common problems that are related to Dominating Set. We first need some additional definitions. An *independent set* is a subset $I \subseteq V$ of which no two vertices are adjacent, a *total dominating set* is a subset $D \subseteq V$ such that every $v \in V$ (thus, also those in D) is adjacent to a vertex in D, and an *edge dominating set* is a subset $D \subseteq E$ such that every edge in E has an endpoint in common with an edge in D.

Independent Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an independent set $I \subseteq V$ in G of size at least k?

Total Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a total dominating set $D \subseteq V$ in G of size at most k?

Edge Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of size at most k?

We will consider algorithms for Independent Set in Chapter 7, for Total Dominating Set in Chapters 5, 11, 12, and 13, and for Edge Dominating Set in Chapter 6.

1.3.1. What is a Graph Domination Problem?

As no precise definition of the concept seems to occur in the literature, we let a *domination problem* be a combinatorial problem that involves at least the following three aspects:

1. *It is a subset problem.* That is, we are given some ground set U, and the problem asks whether there exists a subset $S \subseteq U$ that satisfies certain properties.

2. *The subset must dominate its complement*[5]. That is, every element of $U \setminus S$ must be dominated by an element from S. Some domination problems may also require that the elements in S are dominated.

3. *The domination criterion is based on a neighbourhood relation.* That is, a neighbourhood relation between elements from U determines which elements from U are dominated by a given element $e \in U$.

Similarly, we let a *graph domination problem* be a domination problem in which the required subset is part of a graph, for example, a subset of the vertices or edges.

We will illustrate the three given aspects below. We refer the reader to the list of problems in Appendix B for formal definitions of problems that we have not yet introduced.

Let us first consider the first aspect. For graph domination problems, the problem could ask for a subset of the vertices, such as in Dominating Set or Independent Set, or a subset of the edges, such as in Edge Dominating Set. In non-graph problems such as Matrix Dominating Set (see Chapter 6), the problem could ask for a subset of non-zero entries in a matrix, or in a generalisation of the five queens puzzle to arbitrary-size chessboards, the problem could ask for a subset of the squares to place queens. For other problems, more specific constraints are placed on a solution subset S. For example, the subset needs to be connected in the Connected Dominating Set problem, and it needs to be a clique in the graph in the Dominating Clique problem.

Next, we consider the second aspect: the domination criterion. In the Dominating Set problem, the vertex subset must dominate all other vertices, and in the Edge Dominating Set problem, the edge subset must dominate all other edges. Additional

[5]This corresponds to Telle who defines a $[\rho, \sigma]$-set (also known as $[\rho, \sigma]$-dominating set, see its definition in Section 1.6) to be dominating if $0 \notin \rho$. [295].

constraints can be placed on how often a vertex must be dominated or how often a vertex can dominate another vertex. For example, in p-Dominating Set (see Chapters 11 and 12 that, amongst others, consider this problem), each vertex not in the subset S must be dominated at least p times, or in Capacitated Dominating Set, each vertex in the subset S can dominate at most as many vertices as its capacity. Another variant is Total Dominating Set where each element, including those in the subset S, must be dominated at least once.

Finally, consider the third aspect: the neighbourhood relation that defines which elements can be dominated by a given element. For vertex subset graph domination problems, this neighbourhood relation can be the closed neighbourhood in the graph, such as in Dominating Set or Independent Set, or the open neighbourhood in the graph, such as in Total Dominating Set. Other options include that the neighbourhood relation is defined through the set of vertices up to distance ℓ as in Distance-r Dominating Set. We note that this neighbourhood relation does not need to be symmetric, see for example Directed Dominating Set in directed graphs, or Weak Dominating Set and Strong Dominating Set in undirected graphs; for algorithms for these problems see Chapter 5.

For edge-subset graph domination problems, an edge typically is in the neighbourhood of another edge if they have a common endpoint; this is the case in Edge Dominating Set. For matrix subset problems, an entry is typically in the neighbourhood of another entry if both entries are in the same row or column; this is the case in Matrix Dominating Set. Finally, for generalisations of the five queens puzzle, a typical neighbourhood relation between a pair of squares is that a queen on one square can attack the other square.

This description of a graph domination problem covers all graph domination problems for which we present new algorithms in this PhD Thesis.[6] Perhaps somewhat surprisingly, the notion of a graph domination problem as described above includes the Independent Set problem. This is because of the following well-known simple fact:

Proposition 1.1 ([18]). *An independent set I cannot be extended to a larger independent set by adding a vertex $v \in V \setminus I$ if and only if I is an independent set and a dominating set.*

For more examples of graph domination problems, we refer the reader to the $[\rho, \sigma]$-domination problems defined by Telle [294, 295, 296] which are defined in Section 1.6, or to the list of problems in Appendix B. We note that for most of the $[\rho, \sigma]$-domination problems, fast exact algorithms improving upon the simple brute force algorithms exist [139, 140].

1.3.2. Variants of Graph Domination Problems

For every graph domination problem there exist a number of variants[7]:

[6] Except Partial Dominating Set, but we consider this to be a partial domination problem as not all vertices have to be dominated.

[7] We note that some of these variants may be trivial, for example, consider computing a maximum size dominating set.

- *Decision variant.* In a decision variant, the question is whether there exists a subset of the given type. An example is Dominating Clique.
- *Minimisation variant.* In a minimisation variant, the question is whether there exists a subset of the given type of size at most k. An example is Dominating Set.
- *Maximisation variant.* In a maximisation variant, the question is whether there exists a subset of the given type of size at least k. An example is Independent Set.
- *Counting variant.* In a counting variant, the question is to count all subsets of a given type. An example is #Dominating Set where we must count all minimum size dominating sets. Alternatively, one could ask to count all dominating sets.
- *Enumeration variant.* In an enumeration variant, the question is to enumerate all subsets of a given type. For example, enumerate all dominating sets, or enumerate all minimum size dominating sets.

For the first three of these variants, one also distinguishes between constructive versions, where the required subset need to be constructed explicitly, and non-constructive versions, where one needs only to decide whether the required subset exists.

Note that these five variants do not need to be mutually exclusive, as one could, for example, ask to count minimum size solutions or enumerate maximum size solutions. Also, one often asks to count or enumerate maximal or minimal solutions where maximal and minimal mean that the solution subsets must be inclusionwise maximal and minimal.

1.4. Thesis Overview

This PhD thesis consists of five parts and fourteen chapters. Each part consists of three chapters except the last parts that contains only the conclusion. Except for these five main parts, the thesis contains an introductionary chapter (i.e., this chapter) and an appendix. Below, we give an overview of the contents of this PhD thesis.

Part I: Introduction to Exact Exponential-Time Algorithms. The first part is an introduction to the field of exact exponential-time algorithms.

Chapter 2 introduces a series of common algorithmic techniques in the field of exact exponential-time algorithms that will be used throughout this thesis. This chapter also contains many references to relevant literature and concludes with a short survey on other common techniques in the field that are less relevant to this thesis.

In Chapter 3, we give some background information required in order to understand some results in this thesis or their relevance. This chapter states basic results from classical complexity theory and their relation to the complexity parameters used to analyse algorithms. The chapter also states a few basic results from parameterised complexity and their relation to exact exponential-time algorithms, and surveys a few recent results in exponential-time complexity.

We conclude this introductionary part with a new result in Chapter 4. In this chapter, we study the Partition Into Triangles problem on graphs of maximum degree four for which we give an $\mathcal{O}(1.02220^n)$-time algorithm. This result shows that there exist exponential-time algorithms for $\mathcal{NP}$-hard problems that have an exponential

running time which is so small that the algorithms can be used in practice even on instances that are quite large.

Part II: Branching Algorithms and Measure Based Analyses. In the second part, we give the currently fastest algorithms for a number of standard graph domination problems. These algorithms are branch-and-reduce algorithms and their analyses involve measure-based techniques that give good upper bounds on their running times.

We give a faster algorithm for Dominating Set in Chapter 5. This algorithm is a branch-and-reduce algorithm with an analysis based on *measure and conquer*. In this chapter, we also display how to obtain such an algorithm by iteratively improving known algorithms. Furthermore, we give some results on related other problems such as Total Dominating Set and the parameterised problem k-Nonblocker.

In Chapter 6, we apply the same approach to the Edge Dominating Set problem and obtain the fastest known algorithm for this problem. Here, we also consider a number of related problems such a Matrix Dominating Set and the parameterised problem k-Minimum Weight Maximal Matching.

The Independent Set problem is the topic of Chapter 7. For this problem, we will give the currently fastest algorithms on bounded degree graphs. These algorithms are obtained by doing an extensive case analysis on graphs of average degree at most three (most of this case analysis is given in Appendix A.4). Hereafter, this result is used to obtain the currently fastest algorithms for Independent Set on graph classes with a larger maximum or average degree.

Part III: Inclusion/Exclusion Based Branching Algorithms. In the third part of this thesis, we obtain a number of different results based on a new approach that allows us to use *the principle of inclusion/exclusion* as a branching step.

First, we give the currently fastest polynomial-space and exponential-space algorithms for #Dominating Set in Chapter 8. These algorithms are also used to obtain faster algorithms for Dominating Set restricted to some graph classes and to obtain a faster polynomial-space algorithm for the Domatic Number problem.

In Chapter 9, we extend the approach of Chapter 8 to partial domination problems. Here, we will give the currently fastest polynomial-space and exponential-space algorithms for Partial Dominating Set. We will also use this approach to give the currently fastest algorithm for the parameterised problem k-Set Splitting.

Finally, we consider a completely different problem in Chapter 10, namely the Disjoint Connected Subgraphs problem. This chapter shows that the techniques from Chapter 8 also have applications in different problem settings. Moreover, this chapter shows that it is sometimes useful to consider algorithms that switch from considering a decision problem to considering the counting variant of the same problem at a convenient moment.

Part IV: Dynamic Programming Algorithms on Graph Decompositions. In the fourth part, we consider dynamic programming algorithms on a number of different types of graph decompositions. We give the currently fastest algorithms for a number of problems on these graph decompositions as a function of the associated graph-width parameter. These results are obtained by developing a generalisation of the fast subset convolution algorithm.

The first type of graph decomposition we consider is the tree decomposition. We give the currently fastest algorithms for Dominating Set, #Perfect Matching, $[\rho, \sigma]$-domination problems, and a number of clique covering, partitioning, and packing problems on tree decompositions in Chapter 11. These results have been used as a subroutine in several algorithms in Part III of this thesis.

The second type of graph decomposition we consider is the branch decomposition. Using these decompositions, we give the currently fastest algorithms for Dominating Set, #Perfect Matching, and $[\rho, \sigma]$-domination problems. These results are the topic of Chapter 12.

Finally, we consider clique decompositions, also called k-expressions, in Chapter 13. In this short chapter, we will give the currently fastest algorithms for Dominating Set, Total Dominating Set, and Independent Dominating Set on this type of graph decomposition.

Part V: Conclusion. We conclude this thesis with some concluding remarks, open problems, and directions for further research in Chapter 14.

1.5. Published Papers

This thesis is based on the following refereed journal papers and refereed conference proceedings papers.

[1] Hans L. Bodlaender, Erik Jan van Leeuwen, Johan M. M. van Rooij, and Martin Vatshelle. Faster algorithms on branch and clique decompositions. In P. Hliněný and A. Kucera, editors, *35th International Symposium on Mathematical Foundations of Computer Science, MFCS 2010*, volume 6281 of *Lecture Notes in Computer Science*, pages 174–185. Springer, 2010.

[2] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. A bottom-up method and fast algorithms for max independent set. In H. Kaplan, editor, *12th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2010*, volume 6139 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 2010.

[3] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Fast algorithms for max independent set. *Algorithmica*, 2010. Accepted for publication, to appear.

[4] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Maximum independent set in graphs of average degree at most three in $O(1.08537^n)$. In J. Kratochvíl, A. Li, J. Fiala, and P. Kolman, editors, *7th Annual Conference on Theory and Applications of Models of Computation, TAMC 2010*, volume 6108 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2010.

[5] Jesper Nederlof and Johan M. M. van Rooij. Inclusion/exclusion branching for partial dominating set and set splitting. In V. Raman and S. Saurabh, editors, *5th International Symposium on Parameterized and Exact Computation, IPEC 2010*, volume 6478 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2010.

[6] Daniël Paulusma and Johan M. M. van Rooij. On partitioning a graph into two connected subgraphs. In Y. Dong, D.-Z. Du, and O. H. Ibarra, editors, *20th International Symposium on Algorithms and Computation, ISAAC 2009*, volume 5878 of *Lecture Notes in Computer Science*, pages 1215–1224. Springer, 2009.

[7] Johan M. M. van Rooij. Polynomial space algorithms for counting dominating sets and the domatic number. In T. Calamoneri and J. Díaz, editors, *7th International Conference on Algorithms and Complexity, CIAC 2010*, volume 6078 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2010.

[8] Johan M. M. van Rooij and Hans L. Bodlaender. Design by measure and conquer, a faster exact algorithm for dominating set. In S. Albers and P. Weil, editors, *25th International Symposium on Theoretical Aspects of Computer Science, STACS 2008*, volume 1 of *Leibniz International Proceedings in Informatics*, pages 657–668. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[9] Johan M. M. van Rooij and Hans L. Bodlaender. Exact algorithms for edge domination. In M. Grohe and R. Niedermeier, editors, *3th International Workshop on Parameterized and Exact Computation, IWPEC 2008*, volume 5018 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2008.

[10] Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In A. Fiat and P. Sanders, editors, *17th Annual European Symposium on Algorithms, ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2009.

[11] Johan M. M. van Rooij, Jesper Nederlof, and Thomas C. van Dijk. Inclusion/exclusion meets measure and conquer. In A. Fiat and P. Sanders, editors, *17th Annual European Symposium on Algorithms, ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 554–565. Springer, 2009.

[12] Johan M. M. van Rooij, Marcel E. van Kooten Niekerk, and Hans L. Bodlaender. Partition into triangles on bounded degree graphs. In I. Cerná, T. Gyimóthy, J. Hromkovic, K. G. Jeffery, R. Královic, M. Vukolic, and S. Wolf, editors, *37th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2011*, volume 6543 of *Lecture Notes in Computer Science*, pages 558–569. Springer, 2011.

1.6. Basic Notation and Definitions

We conclude this introduction by giving some commonly used notations and basic definitions.

Graphs. An undirected *graph* $G = (V, E)$ consists of a set V of elements called *vertices* and a set E of unordered pairs of the vertices V called *edges*. We will generally use n for the number of vertices of a graph G, and m for its number of edges.

A *path* from a vertex v_1 to a vertex v_k in a graph G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ from V such that for each pair of vertices v_i, v_{i+1} there exists an edge $\{v_i, v_{i+1}\} \in E$. The *length* of such a path is the number of vertices in the sequence minus one: $k-1$. The *distance* between two vertices $u, v \in V$ in G is defined as the minimum length of any path from u to v in G.

A set of vertices $X \subseteq V$ is *connected* if any two vertices $u, v \in X$ are connected through a path in $G[X]$. A *connected component* of a graph G is an inclusionwise-maximal connected subset of the vertices V. A subset $S \subseteq V$ is called a *separator* of a connected graph G if $G[V \setminus S]$ is not connected. A vertex $v \in V$ is called an *articulation point* of G if $\{v\}$ is a separator in G.

The *open neighbourhood* $N(v)$ of a vertex $v \in V$ is defined as the set of vertices that are connected to v through an edge in E, i.e, $N(v) = \{u \in V \mid \{u, v\} \in E\}$. The *closed neighbourhood* $N[v]$ of v is defined as the set of vertices containing the vertex v and its open neighbourhood: $N[v] = \{v\} \cup N(v)$. The *degree* $d(v)$ of a vertex $v \in V$ is the number of neighbours of this vertex, i.e, $d(v) = |N(v)|$.

The definition of a neighbourhood can be extended to vertex subsets $Y \subseteq V$: $N(Y) = \bigcup_{v \in Y} N(v)$ and $N[Y] = \bigcup_{v \in Y} N[v]$. Also, $N^2(v)$ is the set of vertices at distance two from v, and $N^2[v]$ is the set of vertices at distances at most two from v.

The *maximum degree* of a graph G, denoted by $\Delta(G)$, is the maximum of the degrees of the vertices of G. The *minimum degree* of a graph is defined similarly and is denoted by $\delta(G)$. A graph G is called r-regular if $\Delta(G) = \delta(G) = r$, i.e., if every vertex in G has degree r.

For a vertex subset $X \subseteq V$, the *induced subgraph* $G[X]$ of G induced by X is defined to be the restriction of G to the vertices in X, that is, $G[X] = (X, F)$ where $F = (X \times X) \cap E$. By $N_X(v)$ and $N_X[v]$, we denote to the open, respectively closed, neighbourhood of a vertex $v \in X$ in $G[X]$. Similarly, the X-degree of a vertex $v \in X$ is denoted by $d_X(v)$: $d_X(v) = |N_X(v)|$. This notation of neighbourhoods in a subgraph induced by $X \subseteq V$ extends to neighbourhoods of subsets $Y \subseteq X$: $N_X(Y) = (\bigcup_{v \in Y} N_X(v)) \setminus Y$ and $N_X[Y] = \bigcup_{v \in Y} N_X[v]$.

A *bipartite graph* is a graph G whose vertices V can be partitioned into two subsets V_1 and V_2 such that all edges of G have one endpoint in V_1 and one endpoint in V_2. A *planar graph* is a graph G that can be drawn in the plane in such a way that the edges do not intersect except at their endpoints. The *line graph* $L(G)$ of a graph G is the graph $L(G) = (E, F)$ that has the edges of G as vertices and in which two vertices are connected by an edge in F if and only if the corresponding edges in G share an endpoint. That is, $L(G) = (E, \{\{e1, e2\} \mid \exists v \in V \ : \ v \in e_1 \wedge v \in e_2\})$.

Finally, we denote the graph that consists only of a single *path* on l vertices by P_l, and the graph that consists only of a *cycle* on l vertices by C_l. That is, if we let $V_l = \{v_1, v_2, \ldots, v_l\}$, then $P_l = (V_l, E_{P_l})$ where $E_{P_l} = \{\{v_i, v_{i+1}\} \mid 1 \leq i \leq l-1\}$, and $P_l = (V_l, E_{C_l})$ where $E_{C_l} = E_{P_l} \cup \{\{v_l, v_1\}\}$.

Vertex and Edge Subsets. A set of vertices $X \subseteq V$ is called a *clique* in a graph G if every pair of distinct vertices $u, v \in X$ is connected by an edge in G. An *independent set* $I \subseteq V$ in G is a set of vertices such that there is no edge between any two vertices in I. A *vertex cover* $C \subseteq V$ in G is a set of vertices such that for every edge $\{u, v\} \in E$, either $u \in C$ or $v \in V$. Cliques, independent sets, and vertex covers are closely related. That is, a set $C \subseteq V$ is a vertex cover if and only if $V \setminus C$ is an independent set.

ρ	σ	Standard description
$\{0,1,\ldots\}$	$\{0\}$	Independent Set
$\{1,2,\ldots\}$	$\{0,1,\ldots\}$	Dominating Set
$\{0,1\}$	$\{0\}$	Strong Stable Set/2-Packing/ Distance-2 Independent Set
$\{1\}$	$\{0\}$	Perfect Code/Efficient Dominating Set
$\{1,2,\ldots\}$	$\{0\}$	Independent Dominating Set
$\{1\}$	$\{0,1,\ldots\}$	Perfect Dominating Set
$\{1,2,\ldots\}$	$\{1,2,\ldots\}$	Total Dominating Set
$\{1\}$	$\{1\}$	Total Perfect Dominating Set
$\{0,1\}$	$\{0,1,\ldots\}$	Nearly Perfect Set
$\{0,1\}$	$\{0,1\}$	Total Nearly Perfect Set
$\{1\}$	$\{0,1\}$	Weakly Perfect Dominating Set
$\{0,1,\ldots\}$	$\{0,1,\ldots,p\}$	Induced Bounded Degree Subgraph
$\{p,p+1,\ldots\}$	$\{0,1,\ldots\}$	p-Dominating Set
$\{0,1,\ldots\}$	$\{p\}$	Induced p-Regular Subgraph

Table 1.2. Examples of $[\rho,\sigma]$-dominating sets (taken from [294, 295, 296]).

Furthermore, a set $C \subseteq V$ is a clique if and only if C is an independent set in the graph $\bar{G} = (V, (V \times V) \setminus E)$.

A *dominating set* $D \subseteq V$ in G is a set of vertices such that every vertex $v \in V$ is either in D or adjacent to some vertex in D. A *total dominating set* is a subset $D \subseteq V$ such that every $v \in V$ including those in D is adjacent to a vertex in D.

A *distance-r dominating set* is a set of vertices $D \subseteq V$ such that for every vertex $v \in V$ there exists a vertex $u \in D$ that lies at distance at most r from v. For a given integer $t \in \mathbb{N}$, a *partial dominating set* is a set of vertices $D \subseteq V$ such that there is a subset $U \subseteq V \setminus D$ of size at most t such that every vertex $v \in V \setminus U$ is adjacent to a vertex in D, i.e., D dominates all vertices in V except for the at most t vertices in U.

The notion of a $[\rho,\sigma]$-*dominating set*, introduced by Telle in [294, 295, 296], is a generalisation of many vertex subsets including an independent set, a dominating set, and a total dominating set.

Definition 1.2 ($[\rho,\sigma]$-Dominating Set). Let $\rho,\sigma \subseteq \mathbb{N}$, a $[\rho,\sigma]$-*dominating set* in a graph G is a subset $D \subseteq V$ such that:

- for every $v \in V \setminus D$: $|N(v) \cap D| \in \rho$.
- for every $v \in D$: $|N(v) \cap D| \in \sigma$.

See Table 1.2 for some example vertex subsets defined as $[\rho,\sigma]$-dominating sets. Note that although these vertex subsets are known as $[\rho,\sigma]$-dominating sets, we consider the corresponding combinatorial problems to be graph domination problems only if $0 \notin \rho$; see our description of a *domination problem* in Section 1.3.1.

A *matching* in G is a set of edges $M \subseteq E$ such that no two edges in M are incident to the same vertex. A vertex that is an endpoint of an edge in M is said to be *matched* to the other endpoint of this edge. A *perfect matching* is a matching in which every vertex $v \in V$ is matched. Computing a maximum-size matching in a graph, and

thus also deciding whether a graph has a perfect matching, can be done in polynomial time [120]. However, the following problem that asks to compute the number of perfect matchings is #$\mathcal{P}$-hard; see Chapter 3.

#Perfect Matching
Input: A graph $G = (V, E)$.
Question: How many perfect matchings exist in G?

We note that a vertex or edge subset of a certain type is called a minimal or maximal subset if it is inclusionwise minimal or maximal, respectively. This is different from subsets of the same type that are called minimum or maximum subsets: these are subsets of this type of minimum or maximum cardinality, respectively. For example, a maximal independent set in G is an independent set to which no other vertices of G can be added, while a maximum independent set is an independent set of maximum size in G. Combinations also exist, for example, a minimum maximal matching is an inclusionwise-maximum matching of minimum cardinality.

Satisfiability Problems. A *literal* of a variable x is a *positive occurrence* of x, also denoted by x, or a *negative occurrence* of x, denoted by $\neg x$. A *clause* is a set of literals. A truth assignment of X is an assignment of the values $True$ and $False$ to the variables of X. Given a truth assignment of X, the value of a positive literal equals the value of the corresponding variable, and the value of a negative literal equals the negation of the value of the corresponding variable. For a given set of clauses C, we denote the *frequency* of a variable x by $f(x)$, that is, $f(x)$ is the number of occurrences of the literals x and $\neg x$ in C.

In the Satisfiability problem, a clause is defined to be *satisfied* by a given truth assignment if it contains at least one literal set to $True$.

Satisfiability
Input: A set of clauses C using a set of variables X.
Question: Does there exist a truth assignment to the variables in X that satisfies all clauses in C?

The k-Satisfiability problem is defined similarly. In this variant of the Satisfiability problem the clauses C in the input can have maximum size k, that is, they may contain at most k literals.

Besides maximisation variants, where we are asked to satisfy a maximum number of clauses, satisfiability problems have counting variants and enumeration variants that are all defined similarly to the variants of graph domination problems in Section 1.3.2. Many other variants of satisfiability problems exist by using different definitions of when a clause is satisfied. For example, in the Exact Satisfiability problem, a clause is satisfied when *exactly* one literal in the clause is set to $True$, and in the Not-All-Equal Satisfiability problem, a clause is satisfied when at least one literal in it is set to $True$ and at least one literal in it is set to $False$.

Set Cover Problems. A multiset is a set that can contain the same element multiple times. Given a multiset of sets $\mathcal{S}$, a *set cover* $\mathcal{C}$ of $\mathcal{S}$ is a subset $\mathcal{C} \subseteq \mathcal{S}$ such that every element in any of the sets in $\mathcal{S}$ occurs in some set in $\mathcal{C}$. I.e., a set cover $\mathcal{C}$ of $\mathcal{S}$ is a collection of sets such that $\bigcup_{S \in \mathcal{C}} S = \bigcup_{S \in \mathcal{S}} S$. The universe $\mathcal{U}(\mathcal{S})$ of $\mathcal{S}$ is the set of all

elements that occur in any set in $\mathcal{S}$: $\mathcal{U}(\mathcal{S}) = \bigcup_{S \in \mathcal{S}} S$. A set cover $\mathcal{C}$ is called a *minimum set cover* if it is of minimum cardinality over all possible set covers. This leads to the following computational problem:

Set Cover
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}(\mathcal{S})$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a set cover $\mathcal{C} \subseteq \mathcal{S}$ of size at most k?

We often denote a set cover instance by the tuple $(\mathcal{S}, \mathcal{U})$ omitting the dependency of $\mathcal{U}$ on $\mathcal{S}$. We note that the *frequency* of an element e is denoted by $f(e)$, and the set of all elements in $\mathcal{S}$ that contain e by $\mathcal{S}(e) = \{S \in \mathcal{S} \mid e \in S\}$, thus, $f(e) = |\mathcal{S}(e)|$.

For a set S, we denote its *powerset* by 2^U, i.e, $2^U = \{S' \mid S' \subseteq S\}$. Also, a *partitioning* of a set S into k subsets is defined as a collection of subsets $S_1, S_2, \ldots, S_k$ such that $S = \bigcup_{i=1}^{k} S_i$, while for any two distinct S_i and S_j we have $S_i \cap S_j = \emptyset$. A partitioning of a set S into k subsets is also called a *k-partition*.

Additional Notation. In exact exponential-time algorithms and parameterised algorithms, one often uses the $\mathcal{O}^*$-notation introduced by Woeginger [319]. This notation is similar to the usual big-$\mathcal{O}$ notation but suppresses all polynomial factors. For exponential-time algorithms, this means that $f(n) = \mathcal{O}^*(g(n))$ if $f(n) = \mathcal{O}(p(n)g(n))$ for a polynomial $p(n)$. If $g(n)$ is at least c^n for some constant c, this is identical to the more common $\tilde{\mathcal{O}}$-notation defined as $f(n) = \tilde{\mathcal{O}}(g(n))$ if $f(n) = \mathcal{O}(g(n)\log^k(g(n)))$ for some $k \in \mathbb{N}$.

For parameterised algorithms, there is a difference between both notations. An input to a parameterised problem is measured by two complexity parameters, often denoted by n and k, where k is said to be the 'parameter' which is typically assumed to be significantly smaller than n; see Section 3.3 in Chapter 3. In this field, we say that a running time is $\mathcal{O}^*(f(k))$ if the running time is $\mathcal{O}(p(n,k)f(k))$ for some polynomial p in both k and n.

For many results in exact exponential-time algorithms, the base of the exponent c of the running time is a real number that is rounded up to display a bounded number of digits. In such a case, we can use the usual $\mathcal{O}$-notation, as any function $f(n)$ that is $\mathcal{O}^*(c^n)$ also is $\mathcal{O}((c+\epsilon)^n)$ for any $\epsilon > 0$ that we can take small enough to disappear in the numerical rounding.

Finally, we note that we use the Iverson bracket notation: $[\mathit{condition}] = 1$ if the condition is $True$ and $[\mathit{condition}] = 0$ if the condition is $False$.

I

Introduction to Exact Exponential-Time Algorithms

2

Common Techniques in Exact Exponential-Time Algorithms

The first non-trivial results in the field of exact exponential-time algorithms date back to the nineteen sixties and nineteen seventies. The field gained a lot of attention over the last fifteen years; see also the discussion in Section 1.2. In this chapter, we will survey a series of important techniques in the field that are relevant to this PhD thesis. We do not aim at giving an overview of all techniques that have been developed over the years. However, we will conclude this chapter by mentioning additional techniques and giving some interesting pointers to the literature.

As example algorithms, we will, amongst others, use a series of well-known results that can be explained quickly. These examples include a simple algorithm that enumerates all maximal independent sets in $\mathcal{O}(1.4423^n)$ time, Liedloff's $\mathcal{O}(1.4143^n)$-time algorithm for Dominating Set on bipartite graphs [224], Björklund's $\mathcal{O}^*(2^n)$-time algorithm for #Perfect Matching [27], the classical $\mathcal{O}^*(2^n)$-time algorithm for Hamiltonian Cycle by, independently, Kohn et al. [207], Karp [200], and Bax [14], algorithms for Dominating Set and Edge Dominating Set in graphs of maximum degree three based on Fomin and Høie's treewidth bound [147], and the algorithms for Graph Colouring of Lawler that runs in $\mathcal{O}(2.4423^n)$ time [220] and Björklund et al. that runs in $\mathcal{O}^*(2^n)$ time [33].

We begin by introducing branching algorithms in Section 2.1. Here, we also consider the memorisation technique. Next, we consider exponential-time dynamic programming algorithms in Section 2.2. This includes what is known as 'dynamic programming across subsets' in Section 2.2.1, and dynamic programming on graph decompositions (path decompositions and tree decompositions) in Section 2.2.2. In Section 2.3, we introduce one last technique, namely inclusion/exclusion. We conclude with some discussion on other techniques in Section 2.4.

2.1. Branch-and-Reduce Algorithms

Branch and reduce[1] certainly is the most basic technique in the field of exact exponential-time algorithms[2]. It is also the technique that is used the most in this PhD thesis: most algorithms in Chapters 5-10 are branch-and-reduce algorithms. The approach was pioneered in the nineteen sixties in the Davis-Putman-Logemann-Loveland procedure [101, 102].

A branch-and-reduce algorithm is a recursive *divide-and-conquer* algorithm that consists of a series of *reduction rules* and *branching rules*. The reduction rules are rules that can be applied in polynomial time to simplify, or even solve, instances with specific properties. Reduction rules that solve instances are sometimes also called *halting rules*. The branching rules are rules that solve the problem instance by recursively solving a series of smaller instances of the same problem. The recursive calls are also called *branches* of the algorithm.

To explain what a branch-and-reduce algorithm is, we give an example. Consider the Independent Set and #Independent Set problems. The first problem asks whether there exists an independent set of size at least k in a graph G, and the second problem asks to compute the number of maximum independent sets in G. Algorithm 2.1 is a simple branch-and-reduce algorithm for these two problems. It requires a graph G as input and returns solutions that are tuples (s, a) where s is the size of a maximum independent set in G and a is the number of such sets.

To improve the readability of our pseudo-code, we adopt the following convention:

Convention about Pseudo-Code. If values of variables are claimed to exist in an if-statement, then these variables are considered to be instantiated to the appropriate value in the subsequent then-statement.

Algorithm 2.1 has one branching rule: if G contains a vertex v of degree at least three, then it considers either taking v in a maximum independent set or discarding v; it recursively solves the two problems that correspond to each of these choices. The first branch corresponds to taking v in a maximum independent set. In this branch, $N[v]$ is removed because the vertices in $N(v)$ cannot be in an independent set with v. The second branch corresponds to discarding v. Here, only v is removed as we have decided not to take it in the maximum independent set.

After returning from the recursive calls, the algorithm compares the returned numbers from both branches. It adds one to the size s returned from the first branch accounting for the fact that we consider v to be in the maximum independent set. Then, the algorithm either returns the results from the branch corresponding to the largest-size independent set, or adds up the number of such sets from both branches if both independent sets have the same size.

Algorithm 2.1 also has one reduction rule: if v has only vertices of degree at most two, then G consists of a collection of paths and cycles and the problem can be solved

[1] Besides the name 'branch-and-reduce algorithm' many other names exist for the same type of algorithm. These include: branching algorithm, backtracking algorithm, search tree algorithm, pruning the search tree, DPLL algorithm, and splitting algorithm [149].

[2] Fomin and Kratsch state in [149] that 'It is safe to say that at least half of the published fast exponential-time algorithms are branching [branch-and-reduce] algorithms'.

Algorithm 2.1. A simple algorithm for Independent Set and #Independent Set.

Input: a graph $G = (V, E)$
Output: the size of a maximum independent set in G and the number of such sets in G

```
#MIS(G) :
if G contains a vertex v of degree at least three then
    Let (s_take, a_take) = #MIS(G[V \ N[v]])
    Let (s_discard, a_discard) = #MIS(G[V \ {v}])
    if s_take + 1 > s_discard then
        return (s_take + 1, a_take)
    else if s_take + 1 < s_discard then
        return (s_discard, a_discard)
    else
        return (s_discard, a_take + a_discard)
else
    Compute the solution to each connected component of G and combine them
    return (the size of a maximum independent set in G, the number of such sets)
```

in polynomial time. In this case, we can use the following proposition for which we omit the (simple) proof.

Proposition 2.1. *A path P_l on l vertices has a unique maximum independent set of size $(l+1)/2$ if l is odd, and has $1 + l/2$ maximum independent sets of size $l/2$ if l is even. A cycle C_l on l vertices has l maximum independent sets of size $(l-1)/2$ if l is odd, and two maximum independent sets of size $l/2$ if l is even.*

To analyse the running time of a branch-and-reduce algorithm, one looks at the corresponding *search trees*. The search tree of a branch-and-reduce algorithm applied to an instance is defined to be the tree consisting of an internal node for each subproblem on which the algorithm branches and a leaf node for each problem that the algorithm solves directly in polynomial time. Note that no nodes are created for applications of other reduction rules than the halting rules. Since all reduction rules are applied in polynomial time, an upper bound of the form c^n on the size of the search tree gives us a running time of $\mathcal{O}^*(c^n)$ for the branch-and-reduce algorithm. We note that often an upper bound of c^n on the number of leaves of the search tree is proven; since the number of leaves is always larger than the number of internal nodes, this also gives us a running time of $\mathcal{O}^*(c^n)$.

To bound the size of the search tree, one usually considers a set of recurrence relations containing a recurrence relation for each branching rule. Let $N(n)$ be the number of leaves of the search tree of Algorithm 2.1 when applied on an instance consisting of n vertices. Then, we have that:

$$\begin{aligned} N(n) &= 1 && \text{for } n \leq 3 \\ N(n) &\leq N(n-1) + N(n-4) && \text{for } n \geq 4 \end{aligned}$$

Here, the top equation follows from the fact that instances with at most three vertices are solved by the reduction rule. The inequality follows from the fact that a larger instance is either solved by the reduction rule, or solved by generating two subproblems:

one subproblem where at least four vertices are removed, namely v and its at least three neighbours, and one subproblem where one vertex is removed, namely v.

An upper bound on the solution to such a set of recurrence relations can be found by standard techniques. More specifically, the solution of a linear recurrence relation such as the above is of the form $p(n)c^n$ for some $c > 1$ and polynomial $p(n)$. If one substitutes the exponential factor c^n in the inequality of the recurrence relation, one finds an upper bound on the running time by letting c be a (minimum) solution to the equation:

$$c^n \geq c^{n-1} + c^{n-4}$$

Or, after dividing both sides by c^n and replacing the inequality by an equality, by letting c be the solution to:

$$1 = c^{-1} + c^{-4}$$

The unique positive real root of this equation is $c \approx 1.38028$.

Because polynomial factors in the running time are suppressed by the decimal rounding of c (see the discussion on the big-$\mathcal{O}$ notation in Section 1.6), one obtains the following result:

Proposition 2.2. *Algorithm 2.1 solves* Independent Set *and* #Independent Set *in* $\mathcal{O}(1.3803^n)$ *time.*

Notice that the precise bound on the size of the instances that are solved directly by reduction rules does not matter in the above analysis: this influences only the polynomial factors. Also, notice that the algorithm does not contain reduction rules that increase the complexity parameter of an instance: if a reduction rule would do so, an analysis along these lines would not necessarily be correct.

We can significantly shorten the above analysis using the following concepts. A *branching vector* associated with a branching rule that generates r subproblems is an r-tuple $(t_1, t_2, \ldots, t_r)$ such that each t_i equals the minimum decrease in the complexity parameter in subproblem i. In the above example, the branching rule has the associated branching vector $(1, 4)$. For such a branching vector, the associated *branching number* (also called branching factor) is the unique positive real root of the corresponding equation: $1 = \sum_{i=1}^{r} c^{-t_i}$. Given a series of branching rules, we can bound the running time of the corresponding algorithm by giving a branching vector for each branching rule of the algorithm. Then, the running time is $\mathcal{O}^*(c^n)$ where c is the largest associated branching number, which, in this case, is approximately 1.38028.

The branching number associated to a branching vector $(t_1, t_2, \ldots, t_r)$ is often denoted by $\tau(t_1, t_2, \ldots, t_r)$. Hence, the branching number associated to the branching vector $(1, 4)$ equals $\tau(1, 4) \approx 1.38028$.

Definition 2.3 (τ-function). The *τ-function* is the function that assigns the associated branching number to a given branching vector. That is, $\tau(t_1, t_2, \ldots, t_r)$ is the root of the equation: $1 = \sum_{i=1}^{r} c^{-t_i}$.

Extensive treatments of branching vectors, branching numbers, the τ-function, and their properties are given in [216, 217, 218]. For a shorter introduction, see also [149].

Algorithm 2.2. A simple algorithm for enumerating all maximal independent sets.

Input: a graph $G = (V, E)$, an independent set I, and a set of discarded vertices D
Output: a list containing all maximal independent sets in G

EMIS(G, I, D):
1: **if** G is the empty graph **then**
2: **return** $\{I\}$
3: **else if** there is a vertex $v \in D$ such that $N[v] \subseteq D$ **then**
4: **return** $\emptyset$
5: **else**
6: Let v_0 be a vertex of minimum degree in G with $N(v_0) = \{v_1, v_2, \ldots, v_{d(v_0)}\}$
7: **for** $i = 0$ **to** $d(v_0)$ **do**
8: **if** $v_i \notin D$ **then**
9: Let $P = \{v_0, v_1, \ldots, v_{i-1}\}$
10: Let $S_i = \text{EMIS}(G[V \setminus (P \cup N[v_i])], I \cup \{v_i\}, (D \cup P) \setminus N[v_i])$
11: **return** $\bigcup_{0 \leq i \leq d(v_0), v_i \notin D} \{S_i\}$

We note that one can easily improve Algorithm 2.1. For #Independent Set, there exist an $\mathcal{O}(1.3247^n)$-time algorithm by Dahllöf and Jonsson [93], which was later improved by considering algorithms for the more general #2-Satisfiability problem [95, 96, 119, 160, 316, 328] resulting in an $\mathcal{O}(1.2377^n)$-time algorithm by Wahlström [316]. For an overview of faster algorithms for Independent Set, we refer the reader to Chapter 7.

We will demonstrate the use of branching vectors and branching numbers on the next algorithm: Algorithm 2.2. This is a branch-and-reduce algorithm that enumerates all maximal independent sets, i.e., all independent sets I such that $I \cup \{v\}$ is not an independent set for any $v \in V$. This algorithm is given as input a graph G, an independent set I, and a set of discarded vertices D. Here, G is the graph from which we can still add vertices to the maximal independent set I that we are constructing. However, we may not add vertices from D: these vertices are not removed because they still require a neighbour in I in order for I to become a maximal independent set. The function of the set D is to prevent the algorithm from generating the same maximal independent set twice. In each branch of the search tree, the algorithm generates at most one maximal independent set, and it returns all maximal independent sets through its recursive calls.

Algorithm 2.2 has two reduction (halting) rules: if G is empty, it returns I as we are in a leaf of the search tree; and, if D contains a vertex v whose entire neighbourhood in G is contained in D, then it returns the empty set because no maximal independent set can be constructed from this subproblem (since we can always add v to whatever set we could end up with).

The algorithm has one branching rule: it selects a vertex v_0 of minimum degree in G and generates a subproblem for each vertex in $N[v_0] \setminus D$. If $v_0 \notin D$, then it starts by creating a subproblem that corresponds to taking v_0 in the independent set. Here, it puts v_0 in I and removes $N[v_0]$ from G since these vertices can no longer be added to I. It also removes $N[v_0]$ from D since all these vertices now have a neighbour in I. In the other subproblems, it considers not taking v_0 in I and instead taking at least one of its

neighbours. The algorithm considers these neighbours $v_1, v_2, \ldots, v_{d(v_0)}$ one at a time and generates a subproblem for each such neighbour $v_i \notin D$. In these subproblems, the algorithm takes v_i in I and forbids taking any of the previously considered neighbours to be taken in I. Therefore, the algorithm removes $N[v_i]$ from G and D and adds the remaining previously considered neighbours of v to D (the set P in the pseudocode of Algorithm 2.2). Notice that this use of D prevents the algorithm from generating the same maximal independent set twice because maximal independent sets containing multiple neighbours of v are now considered only in the branch that corresponds to taking the first such neighbour in I. Finally, the algorithm returns a set containing all maximal independent sets constructed from the recursive calls.

To prove an upper bound on running time of Algorithm 2.2, we consider the corresponding branching vectors. If $d \geq 1$ is the minimum degree in G, then the algorithm generates at most $d+1$ subproblems. In each such subproblem, at least $d+1$ vertices are removed since all the neighbours of v also have degree at least d. We stress that a vertex that is put in D is not considered to be removed. The branching numbers of these branchings correspond to $\tau(2,2), \tau(3,3,3), \tau(4,4,4,4), \tau(5,5,5,5,5)$, etc. It is easy to verify that $\tau(\vec{b}) = t^{\frac{1}{t}}$ if $\vec{b}$ is a vector of length t with value t at each coordinate. Therefore, the algorithm runs in $\mathcal{O}^*(3^{\frac{1}{3}n})$ time as $t^{\frac{1}{t}}$ is maximal for $t \in \mathbb{N} \setminus \{0\}$ if $t = 3$. This gives a running time of $\mathcal{O}(1.4423^n)$. Thus, we have obtained the following result.

Proposition 2.4. *Algorithm 2.2 enumerates all maximal independent sets in a graph G in $\mathcal{O}(1.4423^n)$ time.*

As each leaf of the search tree of Algorithm 2.2 generates at most one maximal independent set, this analysis directly proves the following proposition.

Proposition 2.5 ([235, 241]). *An n-vertex graph contains at most $3^{\frac{1}{3}n}$ maximal independent sets.*

We note that the running time of Algorithm 2.2 is optimal for enumerating maximal independent sets in the sense that there exist graphs that contain $3^{\frac{1}{3}n}$ maximal independent sets [235, 241]. However, if one would be interested only in counting the number of maximal independent sets instead of enumerating them, then one can do so in $\mathcal{O}(1.3642^n)$ time using an algorithm by Gaspers et al. [164].

2.1.1. Memorisation

The running time of branch-and-reduce algorithms is often improved using memorisation[3]. This technique was introduced to the field of exact exponential-time algorithms by Robson in 1986 in an algorithm for Independent Set [270]. It is often used to improve the running time of branch-and-reduce algorithms at the expense of using exponential space, e.g., see [142, 144, 229, 270, 271].

Algorithms using this technique store the solutions to all subproblems generated by the recursive calls in the search tree in an exponential-size database. Before computing a solution to a generated subproblem, the algorithm checks whether the solution has already been computed in another branch of the search tree, and if so, find this solution

[3]Memorisation is often also called *memoisation*.

Algorithm 2.3. A modification of Algorithm 2.1 using memorisation.

Input: a graph $G = (V, E)$
Output: the size of a maximum independent set in G and the number of such sets in G
$\#\mathrm{MIS}(G, D)$:

```
1: if the database D contains an entry for the key G then
2:    return the tuple (s, a) associated with G in the database D
3: else if G contains a vertex v of degree at least three then
4:    Let (s_take, a_take) = #MIS(G[V \ N[v]], D)
5:    Let (s_discard, a_discard) = #MIS(G[V \ {v}], D)
6:    if s_take + 1 > s_discard then
7:       return (s_take + 1, a_take) and add (G; (s_take + 1, a_take)) to D
8:    else if s_take + 1 < s_discard then
9:       return (s_discard, a_discard) and add (G; (s_discard, a_discard)) to D
10:   else
11:      return (s_discard, a_take + a_discard) and add (G; (s_discard, a_take + a_discard)) to D
12: else
13:   Compute the solution to each connected component of G and combine them
14:   return (the size of a maximum independent set in G, the number of such sets)
```

by querying the database. Since such a database can be implemented such that both querying and adding solutions can be done in time logarithmic in the database size, these actions can be executed in polynomial time.

To show how this works, we modify Algorithm 2.1 using memorisation: this leads to Algorithm 2.3. This algorithm uses a database D that contains key-value pairs where the keys are generated subproblems, i.e., graphs, and the values are the solution to these subproblems. Algorithm 2.3 first queries the database D before branching, and returns the solution to the current subproblem if it is found in D. Otherwise, the algorithm branches, and after returning from the recursive calls, it adds the solution to the current subproblem to the database D. Notice that the algorithm does not need to add solutions to D that are computed in polynomial time by the reduction rule since the algorithm does not branch on these subproblems anyway.

Proposition 2.6. *Algorithm 2.3 solves* Independent Set *and* #Independent Set *in* $\mathcal{O}(1.3424^n)$ *time and space.*

Proof. Let $N_h(n)$ be the number of subproblems consisting of h vertices generated when the algorithm is applied to an n-vertex instance. We can bound $N_h(n)$ in two ways. Firstly, by using a recurrence relation similar to the proof of Proposition 2.2.

$$\begin{aligned} N_h(n) &\leq 1 && \text{for } n \leq h \\ N_h(n) &\leq N_h(n-1) + N_h(n-4) && \text{for } n > h \end{aligned}$$

Solving this recurrence relation yields an upper bound on $N_h(n)$ of c^{n-h} where $c = \tau(1, 4) \approx 1.38028$ as in the proof of Proposition 2.2.

Secondly, we can bound $N_h(n)$ by noticing that each input graph on n vertices has at most $\binom{n}{h}$ induced subgraphs consisting of h vertices. Hence, $N_h(n)$ can be bounded

in the following way:

$$N_h(n) \leq \min\left\{1.3803^{n-h}, \binom{n}{h}\right\}$$

We will prove the claimed bound on running time of Algorithm 2.3 by using the first bound if $h > \alpha n$ and the second bound if $h \leq \alpha n$, for some $0 < \alpha < \frac{1}{2}$. This leads to the following bound on the number of leaves of the search tree $N(n)$:

$$N(n) \leq \sum_{h=0}^{n} N_h(n) \leq \sum_{h=0}^{\lfloor \alpha n \rfloor} \binom{n}{h} + \sum_{h=\lceil \alpha n \rceil}^{n} 1.3803^{n-h} \leq \alpha n \binom{n}{\lfloor \alpha n \rfloor} + (1-\alpha)n \cdot 1.3803^{(1-\alpha)n}$$

To minimise the upper bound on the running time, we choose α such that both exponential factors in the upper bound on the running time are equal, i.e, such that $1.3803^{(1-\alpha)n} = \binom{n}{\alpha n}$. Using Stirling's formula and ignoring polynomial factors, both terms are equal if α is the solution to the following equation; for a detailed derivation, e.g., see [299].

$$c^{1-\alpha} = \frac{1}{\alpha^{\alpha}(1-\alpha)^{(1-\alpha)}} \qquad \text{with } c = \tau(1,4) \approx 1.38028$$

By solving this equation numerically, we obtain $\alpha \approx 0.08652$. This leads to $N(n) \leq 1.3424^n$ from which the claimed running time follows. □

We note that the bound of $\binom{n}{h}$ on the number of induced subgraphs that appear as subproblems can be improved significantly if one requires these induced subgraphs to be connected and of some maximum degree [270]. Further improvements are possible if one additionally requires that the induced subgraphs that appear as subproblems are of minimum degree two [271]. See [142] for an overview of these improvements.

2.2. Dynamic Programming

Another well-known paradigm for algorithm design is *dynamic programming*. Similar to branch-and-reduce algorithms, dynamic programming algorithms are based on recursive formulations where the solution to the problem is constructed by recursively combining partial solutions from suitable subproblems. In this paradigm, the considered subproblems are stored in memory. As such, the memorisation technique from Section 2.1.1 can be seen as a top-down variant of a dynamic programming algorithm.

More interesting than top-down dynamic programming algorithms, are bottom-up variants. These algorithms begin by solving small subproblems, and they combine solutions to considered subproblems to obtain solutions to larger subproblems. In the field of exact exponential-time algorithms, two variants of this approach are prominent. The first of these variants is based on dynamic programming across subsets. We consider this approach in Section 2.2.1. The second variant is based on dynamic programming on graph decompositions. This is the topic of Section 2.2.2.

2.2.1. Dynamic Programming Across Subsets

Dynamic programming across subsets is a form of dynamic programming where a subset structure is used to formulate the recursive structure of the algorithm. This approach dates back to the classical $\mathcal{O}^*(2^n)$-time algorithm for Travelling Salesman Problem due to Bellman [17] and independently Held and Karp [180]. The algorithm for Travelling Salesman Problem is often given as the first example of an exact exponential-time algorithm; for example in [149, 319].

We choose to consider two different examples that are more closely related to the topic of this thesis, namely the Graph Colouring algorithm of Lawler [220], and the algorithm for Dominating Set on bipartite graphs of Liedloff [224].

We will begin by introducing the Graph Colouring problem.

Graph Colouring
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exists a colouring of the vertices of G using at most k colours such that no two vertices of the same colour are adjacent?

Because, in a solution to above problem, each set of vertices with the same colour contains no adjacent vertices, the problem is equivalent to the following problem: given a graph $G = (V, E)$, can we cover G with at most k independent sets in G? The equivalence follows directly by taking, for each of the at most k colours c, the independent set consisting of the vertices with colour c.

This leads to the following classical result due to Lawler:

Proposition 2.7 ([220]). Graph Colouring *can be solved in* $\mathcal{O}(2.4423^n)$ *time and* $\mathcal{O}(2^n)$ *space.*

Proof. Let $G = (V, E)$ be the input graph. Let C be a set of vertices with the same colour in a solution of the problem on G. Observe that, in a solution, we can change the colour of any vertex v to the colour of the vertices in C as long as the vertex v does not have a neighbour in C. By repeating this action, C becomes a maximal independent set. Hence, we can assume, without loss of generality, that any solution contains a colour whose corresponding vertices form a maximal independent set I in G.

Let $C(S)$ be the number of colours required to colour $G[S]$ for any subset $S \subseteq V$. Because the above observation does not only hold for G, but also for any induced subgraph of G, $C(S)$ satisfies the following recurrence:

$$C(S) = 1 + \min\{C(S \setminus I) \mid I \text{ is a maximal independent set in } G[S]\}$$

As a consequence, we can compute $C(V)$ by setting $C(\emptyset) = 0$, and evaluating $C(S)$ for all $S \subseteq V$ in order of increasing size of S.

This approach requires us to enumerate all maximal independent sets in $G[S]$ for all $S \subseteq V$. We can do so for a fixed set S in $\mathcal{O}(1.4423^{|S|})$ time and polynomial space using Proposition 2.4. This leads to the following running time:

$$\sum_{i=0}^{n} \binom{n}{i} \mathcal{O}(1.4423^i) = \sum_{i=0}^{n} \binom{n}{i} \mathcal{O}(1.4423^i 1^{n-i}) = \mathcal{O}(2.4423^n)$$

Namely, we enumerate all maximal independent sets for all $\binom{n}{i}$ induced subgraphs on i

vertices for all i, $1 \leq i \leq n$. The last equality follows from Newton's binomial theorem. The space requirement comes from the 2^n subsets $S \subseteq V$ for which we store $C(S)$. □

This algorithm has been improved often and many algorithms for Graph Colouring that use only polynomial space have been developed; see [9, 27, 46, 65, 78, 123]. The currently fastest algorithm is the $\mathcal{O}^*(2^n)$-time, $\mathcal{O}(1.2916^n)$-space algorithm by Björklund et al. [31]. For an $\mathcal{O}^*(2^n)$-time-and-space algorithm see [33] and Corollary 2.21. When restricted to using polynomial space, then the currently fastest algorithm is an $\mathcal{O}(2.2377^n)$-time algorithm that can be obtained by combining the results of Björklund et al. [33] with a result due to Wahlström [316]. We note that there also exist many exponential-time algorithms for deciding, for a fixed $k \in \{3, 4, \ldots\}$, whether a graph G can be coloured using k colours; see for example [16, 65, 122, 137, 220, 278].

The second example we give is Liedloff's $\mathcal{O}(1.4143^n)$-time-and-space algorithm for Dominating Set on bipartite graphs [224]. Let $G = (A \cup B, E)$ be a bipartite graph with vertex partitions A and B, with $|A| \leq |B|$ without loss of generality. Liedloff's algorithm considers all subsets $X \subseteq A$. It uses the fact that every dominating set D in G with $D \cap A = X$ can be partitioned into three disjoint subsets X, Y, Z with $Y, Z \subseteq B$ [224]:

1. The vertex set $X = D \cap A$.
2. The vertex set $Y = B \setminus N(X)$. All these vertices must be in D as they are not dominated by X and cannot be dominated by any other vertices from B except by the vertices themselves.
3. The vertex set $Z = D \cap N(X)$. This vertex set is a minimum-size set of vertices that dominates all vertices that are not dominated by either X or Y. That is, Z is a minimum-size set from $N(X)$ that dominates all vertices in A that are neither in X nor dominated by $N(X)$.

Using this partitioning, Liedloff obtains the following result.

Proposition 2.8 ([224]). *There exists an algorithm for* Dominating Set *on bipartite graphs running in* $\mathcal{O}(1.4143^n)$ *time and space.*

Proof. We use dynamic programming across subsets to compute, for each $X \subseteq A$, the size of a minimum vertex set from $N(X)$ that dominates all vertices in $A \setminus (X \cup N^2(X))$. This is the size of the set Z associated with X as defined above.

To do so, we number the vertices in B, $B = \{v_1, v_2, \ldots, v_{|B|}\}$, and for each $A' \subseteq A$, we let $D(A', i)$ be the size of a minimum vertex set from $\{v_1, v_2, \ldots, v_i\}$ that dominates the vertices in A'. For $i = 0$, we have $D(\emptyset, 0) = 0$ as zero vertices suffice to dominate nothing, and $D(A', 0) = \infty$, for any $A' \neq \emptyset$, as no such sets exists. For increasing i, we can then compute $D(A', i)$ for all $A' \subseteq A$ using the following recurrence that corresponds to either taking v_i in the dominating set or not:

$$D(A', i) = \min\{1 + D(A' \setminus N(v_i), i - 1),\ D(A', i - 1)\}$$

Obtaining $D(A', |B|)$ for all $A' \subseteq A$ clearly costs $\mathcal{O}^*(2^{|A|})$ time and space.

Having computed these numbers, the algorithm now considers each $X \subseteq A$. Because X fixes all three parts of the dominating set as defined above, the algorithm directly sees which choice of X leads to the smallest dominating set in G. That is, the

algorithm returns the following value which gives the size of a minimum dominating set in G:

$$\min\{|X| + |B \setminus N(X)| + D(A \setminus (X \cup N^2(X)), |B|) \mid X \subseteq A\}$$

The three terms of the sum under the minimum correspond to the sizes of the three parts X, Y, and Z of the dominating set D.

Computing this minimum clearly cost $\mathcal{O}^*(2^{|A|})$ time. Since $|A| \leq |B|$, the execution of this algorithm requires $\mathcal{O}^*(2^{\frac{n}{2}}) = \mathcal{O}(1.4143^n)$ time and space. □

We note that this algorithm is sometimes also considered as an algorithm using *preprocessing*. The preprocessing technique [319] considers restructuring a given input, in this case using dynamic programming, such that the problem can be solved easily thereafter. Other examples of algorithms that are considered to be preprocessing algorithms are the algorithms based on sorting in Section 2.4.

2.2.2. Dynamic Programming on Graph Decompositions

The second type of structure for dynamic programming commonly used in exponential-time algorithms is based on graph decompositions. In this PhD thesis, we consider four types of graph decompositions, namely, path decompositions, tree decompositions, branch decompositions, and clique decompositions. In exact exponential-time algorithms, mostly path and tree decompositions are used.

The notions of pathwidth and treewidth and the related notions of a path decomposition and a tree decomposition were introduced by Robertson and Seymour [267]. The use of these concepts in the field of exact exponential-time algorithms can be attributed to Fomin et al. [147] and independently to Kneis et al. [204].

Definition 2.9 (Path Decomposition). A *path decomposition* of a graph $G = (V, E)$ is a sequence of sets of vertices (called *bags*) $X = (X_1, X_2, \ldots, X_r)$ with $\bigcup_{i=1}^{r} X_i = V$ with the following properties:

1. for each $\{u, v\} \in E$, there exists an X_i such that $\{u, v\} \subseteq X_i$.
2. if $v \in X_i$ and $v \in X_k$, then $v \in X_j$ for all $i \leq j \leq k$.

The *width* of a path decomposition X is $\max_{1 \leq i \leq r} |X_i| - 1$, and the pathwidth $\mathrm{pw}(G)$ of G is the minimum width over all possible path decompositions of G.

Definition 2.10 (Tree Decomposition). A *tree decomposition* of a graph $G = (V, E)$ consists of a tree T in which each node $x \in T$ has an associated set of vertices $X_x \subseteq V$ (called a *bag*) such that $\bigcup_{x \in T} X_x = V$ and the following properties hold:

1. for each $\{u, v\} \in E$, there exists an X_x such that $\{u, v\} \subseteq X_x$.
2. if $v \in X_x$ and $v \in X_y$, then $v \in X_z$ for all nodes z on the path from node x to node y in T.

Similar to pathwidth, the *width* of a tree decomposition T is $\max_{i \in T} |X_i| - 1$, and the treewidth $\mathrm{tw}(G)$ of G is the minimum width overall possible tree decompositions of G.

If a graph G has small pathwidth/treewidth and is given with a path/tree decomposition of G of width k, for some small value of k, then we can efficiently solve many

combinatorial problems on G. This can be done by dynamic programming on the path/tree decomposition [11, 19, 34]. Below, we give two examples of dynamic programming algorithms on path decompositions, one for Dominating Set and one for Edge Dominating Set. After giving these algorithms, we will show how they can be used to design fast exact exponential-time algorithms. For dynamic programming algorithms on tree decompositions, see Chapter 11.

For the two algorithms below, we first need the concept of a nice path decomposition introduced by Kloks [202].

Definition 2.11 (Nice Path Decomposition). A *nice path decomposition* is a path decomposition $X = (X_1, X_2, \ldots, X_r)$ with $X_1 = \{v\}$ for some $v \in V$, with $X_r = \emptyset$, and with each bag X_i with $2 \leq i \leq r$ of one of the following two types:

- *Introduce bag*: $X_i = X_{i-1} \cup \{v\}$ for some $v \notin X_i$. This bag is said to introduce the vertex v.
- *Forget bag*: $X_i = X_{i-1} \setminus \{v\}$ for some $v \in X_i$. This bag is said to forget the vertex v.

It is easy to transform any path decomposition into a nice path decomposition of equal width in polynomial time by adding or removing bags.

Proposition 2.12. *There is an algorithm that, given a path decomposition of a graph G of width k, solves* Dominating Set *on G in $\mathcal{O}^*(3^k)$.*

Proof. First, transform the path decomposition into a nice path decomposition $X = (X_1, X_2, \ldots, X_r)$ of equal width.

Let $V_i = \bigcup_{j=1}^{i} X_j$. For each bag X_i in X, the algorithm computes a table A_i that is indexed by a partitioning of the vertices of X_i into three sets C, D and O. In this table, an entry $A_i(C, D, O)$ stores the size of a minimum vertex set of $G[V_i]$ that satisfies the following four properties:

1. the vertex set dominates all vertices in $V_i \setminus X_i$.
2. the vertex set contains all vertices in C.
3. the vertex set does not contain the vertices in D, but it does dominate all vertices in D.
4. the vertex set does not contain the vertices in O, and it may, but is not required to, dominate some vertices in O.

If we compute these tables A_i for each bag X_i in the path decomposition X, then we can find the size of a minimum dominating set in G in $A_r(\emptyset, \emptyset, \emptyset)$.

Notice that, by definition, the tables A_i have the following property that is called *monotonicity* [2, 3]: for any set $S \subseteq D$, $A_{i-1}(C, D \setminus S, O \cup S) \leq A_{i-1}(C, D, O)$. This inequality is satisfied because $A_{i-1}(C, D \setminus S, O \cup S)$ is the size of a minimum vertex set that needs to satisfy less requirements compared to the vertex set whose size is stored in $A_{i-1}(C, D, O)$, namely less vertices need to be dominated.

Computing the table A_1 for the bag $X_1 = \{v\}$ is straightforward:

$$A_1(\{v\}, \emptyset, \emptyset) = 1 \qquad A_1(\emptyset, \{v\}, \emptyset) = \infty \qquad A_1(\emptyset, \emptyset, \{v\}) = 0$$

If, for $i \geq 2$, X_i is an introduce bag that introduces the vertex v, then we can compute $A_i(C, D, O)$ for each partitioning of X_{i-1} into C, D, O in the following way:

$$A_i(C \cup \{v\}, D, O) = 1 + A_{i-1}(C, D \setminus N(v), O \cup (N(v) \cap X_{i-1}))$$
$$A_i(C, D \cup \{v\}, O) = \begin{cases} A_{i-1}(C, D, O) & \text{if } N(v) \cap C \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$
$$A_i(C, D, O \cup \{v\}) = A_{i-1}(C, D, O)$$

The first recurrence uses the monotonicity property: because all neighbours of v in D that are now dominated by v could either be dominated or not before adding v, the monotonicity property makes sure that we use the corresponding dominating set of minimum size. Correctness of the other two recurrences can easily be verified.

If X_i is a forget bag that forgets the vertex v, then we can use the following recurrence to compute $A_i(C, D, O)$ for each partitioning of X_{i-1} into C, D, O:

$$A_i(C, D, O) = \min\{A_{i-1}(C \cup \{v\}, D, O), A_{i-1}(C, D \cup \{v\}, O)\}$$

This recurrence is correct because we continue with the minimum solution that also dominates the forgotten vertex v, which is now part of $V_i \setminus X_i$.

Since a nice path decomposition has $\mathcal{O}(n)$ bags, each table that is computed for such a bag has $\mathcal{O}(3^k)$ entries, and the computation of each entry can be done in constant time, we conclude that the algorithm runs in $\mathcal{O}^*(3^k)$ time and space. □

Proposition 2.13. *There is an algorithm that, given a path decomposition of a graph G of width k, solves* Edge Dominating Set *on G in $\mathcal{O}^*(3^k)$.*

Proof. The proof is analogous to the proof of Proposition 2.12 using a different set of recurrences for dynamic programming.

In this algorithm, we again use a table A_i for each bag X_i in X that is indexed by a partitioning of the vertices of X_i in three sets C, R and O. In this table, an entry $A_i(C, R, O)$ stores the size of a minimum edge set in $G[V_i]$ that satisfies the following three properties:

1. the edge set dominates all edges in $G[V_i \setminus X_i]$.
2. the edge set precisely has the vertices in C as endpoints on X_i.
3. if we would add edges (possibly outside $G[V_i]$) with endpoints in R to the edge set, then the edge set also dominates all edges between $V_i \setminus X_i$ and X_i.

The vertices in R are required to become an endpoint of an edge dominating set later on. This part of the 3-partition exists because a minimum edge dominating set may include edges that have an endpoint outside of V_i that it needs in order to dominate all edges in $G[V_i]$. The vertices in O are the other vertices (not in C or R).

To simplify the dynamic programming recurrence, we assume that the edges in the minimum edge dominating set do not share any endpoints. We can safely assume this because any minimum edge dominating set whose edges share endpoints can be transformed into one whose edges do not share endpoints by using a standard replacement trick from [177] whose details can be found in Corollary 6.3.

Below, we give the set of dynamic dynamic programming recurrences to compute A_i for each bag in the path decomposition X. We notice that all edges are considered to

be taken into the edge dominating set at some point because, by definition of a path decomposition, each edge is in some bag X_i. In the recurrences below, we consider taking an edge e into the edge dominating set when the algorithm considers the bag where one endpoint of e is already present and the other endpoint is being introduced.

For A_1 we have:

$$A_1(\{v\},\emptyset,\emptyset) = \infty \qquad A_1(\emptyset,\{v\},\emptyset) = 0 \qquad A_1(\emptyset,\emptyset,\{v\}) = 0$$

For an introduce bag X_i that introduces the vertex v, we have:

$$\begin{aligned}
A_i(C \cup \{v\}, R, O) &= 1 + \min\{A_{i-1}(C \cup \{u\}, R \setminus \{u\}, O \setminus \{u\}) \mid u \in N(v) \cap (R \cup O)\} \\
A_i(C, R \cup \{v\}, O) &= A_{i-1}(C, R, O) \\
A_i(C, R, O \cup \{v\}) &= A_{i-1}(C, R, O)
\end{aligned}$$

The recurrence in which v is put in C considers taking an edge between v and a neighbour of v in X_i that is in R or O. This is correct as it does not need to consider neighbours in C nor picking multiple edges because we have assumed that no two edges in the minimum edge dominating set share endpoints. Correctness of the other two recurrences can easily be verified.

For a forget bag X_i that forgets the vertex v, we have:

$$A_i(C,R,O) = \min \begin{cases} A_{i-1}(C \cup \{v\}, R, O) & \text{and} \\ A_{i-1}(C, R, O \cup \{v\}) & \text{if } (N(v) \cap X_i) \subseteq C \cup R \end{cases}$$

This recurrence makes sure that only edge sets in which the edges incident to v are dominated are considered after forgetting v. We note that, for these edge sets to dominated all the required edges, edges containing the vertices in R as endpoints may only be added. This is as required by the definition of A_i.

This algorithm runs in $\mathcal{O}^*(3^k)$ time because of the same reasons as in the proof of Proposition 2.12. □

Most applications of path decompositions in exact exponential-time algorithms are based on the following theorem due to Fomin and Høie [147].

Theorem 2.14 ([147]). *For any $\epsilon > 0$, there exist an integer n_ϵ such that if G is an n-vertex graph of maximum degree at most three with $n > n_\epsilon$, then $\mathrm{pw}(G) < \frac{1}{6}n + \epsilon n$. Moreover, a path decomposition of this width can be computed in polynomial time.*

This theorem is based on a graph-theoretic result by Monien and Preis [238] who show that, for any $\epsilon > 0$, there exists an integer n_ϵ such that the vertices of any 3-regular graph of size at least n_ϵ can be partitioned into two vertex sets V_1, V_2 that differ at most one in size, and such that the number of edges in G between V_1 and V_2 is at most $\frac{1}{6}n + \epsilon n$.

Corollary 2.15. Dominating Set *and* Edge Dominating Set *can be solved in* $\mathcal{O}(1.2010^n)$ *time and space on graphs of maximum degree three.*

Proof. Fix any $\epsilon > 0$ that is small enough such that $3^{\frac{1}{6}+\epsilon} < 1.2010$, and let n_ϵ be the associated value from Theorem 2.14. Since the time and space bound of $\mathcal{O}(1.2010^n)$ is an asymptotic bound, we can assume that the input graph G has at least n_ϵ vertices.

Use Theorem 2.14 to compute a path decomposition of G of width k with $k < \frac{1}{6}n + \epsilon n$, and use Proposition 2.12 or Proposition 2.13 to solve the problem instance by dynamic programming on the computed path decomposition.

The claimed time and space bounds follow since $3^k \leq 3^{\frac{1}{6}n+\epsilon n} < 1.2010^n$ by the choice of ϵ. □

Corollary 2.15 gives the currently fastest algorithms for these problems on graphs of maximum degree three. If we restrict ourselves to using polynomial space, Dominating Set on graphs of maximum degree three can be solved in $\mathcal{O}(1.3161^n)$ time [27], and Edge Dominating Set on graphs of maximum degree three can be solved in $\mathcal{O}(1.2721^n)$ time [323].

Fomin et al. have extended the result of Theorem 2.14 to graphs with vertices of larger degrees [138]. This has led to the following result that we will use a number of times in this thesis.

Proposition 2.16 ([138]). *For any $\epsilon > 0$, there exists an integer n_ϵ such that if G is an n-vertex graph with $n > n_\epsilon$ then*

$$\mathrm{pw}(G) \leq \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 + \frac{23}{45}n_6 + n_{\geq 7} + \epsilon n$$

where n_i is the number of vertices of degree i in G, for any $i \in \{3, 4, 5, 6\}$, and $n_{\geq 7}$ is the number of vertices of degree at least 7. Moreover, a path decomposition of the corresponding width can be constructed in polynomial time.

We note that the coefficients for n_4, n_5, and n_6 in formula of Proposition 2.16 are computed based on the coefficient for n_3. By using the same computation, we could in principle also compute coefficients for n_i for $i \geq 7$. These coefficient will all be smaller than one, but converge to one as i goes to infinity.

Corollary 2.17. Dominating Set *can be solved in $\mathcal{O}(1.4423^n)$ time and space on graphs of maximum degree four.*

Proof. Identical to the proof of Corollary 2.15, now using Proposition 2.16 and the fact that $3^{\frac{1}{3}} < 1.4423$. □

Not all graphs have small pathwidth or treewidth. A clique on n vertices, for example, has pathwidth and treewidth $n - 1$. Therefore, path/tree-decomposition-based approaches are useful only on restricted graph classes, especially on sparse graphs. An example is the following useful result by Kneis et al.

Proposition 2.18 ([204]). *For any graph G with n vertices and m edges, $\mathrm{tw}(G) \leq m/5.769 + \mathcal{O}(\log n)$. Moreover, a tree decomposition of the corresponding width can be constructed in polynomial time.*

For more information on path decomposition and tree decompositions, we refer the interested reader to [36, 38, 44]. For more path/tree-decomposition-based results in exact exponential-time algorithms, see for example [27, 138, 165, 204].

We conclude this section by noting that path and tree decompositions of minimum width can be computed in $\mathcal{O}^*(2^n)$ time by dynamic programming across subsets [41].

The currently fastest algorithm to compute a path decomposition of minimum width is due to Suchan and Villanger and requires $\mathcal{O}(1.9657^n)$ time and space [290]. A tree decomposition of minimum width can be computed in $\mathcal{O}(1.7347^n)$ time and space, or $\mathcal{O}(2.6151^n)$ time and polynomial space [155, 156]; see also [150].

2.3. Inclusion/Exclusion

The last common technique that we will demonstrate is *inclusion/exclusion*. Inclusion/exclusion is a simple combinatorial counting principle often identified with the formula in Proposition 2.19 below. This principle is based on the idea that one can often count certain terms too often, then subtract those terms that were counted too often, and then again add terms that were subtracted too often, etc. This then results in a formula that counts each term in which we are interested exactly once. We will make this approach more concrete below.

Inclusion/exclusion was introduced in the field of exact exponential-time algorithms on the Travelling Salesman Problem with bounded integer weights through an $\mathcal{O}^*(2^n)$-time and polynomial-space algorithm. This result was discovered three times independently: by Kohn et al. [207], by Karp [200], and by Bax [14]. The approach was popularised by a series of results by Björklund et al. who used it to solve many covering and partitioning problems [27, 33]. Recently, this approach has found many new applications [5, 25, 26, 28, 29, 30, 31, 32, 244, 245, 307].

Let us start by considering the following proposition. The formula in this proposition is also known as the *inclusion/exclusion formula*. The presented formulation and proof below are based on [27].

Proposition 2.19. *Let $\mathcal{S}$ be a collection of sets over the universe $\mathcal{U}$, and let $a(X)$ denote the number of sets in $\mathcal{S}$ that are disjoint from X, i.e., $a(X)$ is the number of sets S in $\mathcal{S}$ with $S \cap X = \emptyset$. Also, let c_k be the number of k-tuples of sets $(S_1, S_2, \ldots, S_k)$ with $S_i \in \mathcal{S}$ for $1 \leq i \leq k$ that form a set cover of $(\mathcal{S}, \mathcal{U})$, i.e., with $\bigcup_{i=1}^{k} S_i = \mathcal{U}$. Then:*

$$c_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a(X)^k$$

Proof. Notice that $a(X)^k$ counts the number of k-tuples $(S_1, S_2, \ldots, S_k)$ with $S_i \in \mathcal{S}$ and $S_i \cap X = \emptyset$ for each $1 \leq i \leq k$. We will prove that the contribution of such a k-tuple to the expression at the right hand side of the formula is exactly one if the k-tuple is a set cover, and zero otherwise. This will prove the proposition.

If such a k-tuple is a set cover of $(\mathcal{S}, \mathcal{U})$, then its sets contain all elements of $\mathcal{U}$. Therefore, it is counted only in the term of the above formula with $X = \emptyset$, and its contribution to the summation will be exactly one as required.

Assume that such a k-tuple is not a set cover, and let Y be the set of uncovered elements from $\mathcal{U}$, i.e., $Y = \mathcal{U} \setminus (\bigcup_{i=1}^{k} S_i)$ and $Y \neq \emptyset$. Now, this k-tuple is counted in the terms of the summation that correspond to every $X \subseteq Y$. However, the total contribution of this k-tuple over all relevant terms of the summation is:

$$\sum_{X \subseteq Y} (-1)^{|X|} \cdot 1 = \sum_{i=0}^{|Y|} \binom{|Y|}{i} (-1)^i 1^{|Y|-i} = (1-1)^{|Y|} = 0$$

where the second equality above follows from Newton's binomial theorem. □

We will use the above proposition to prove that #Perfect Matching can be solved in $\mathcal{O}^*(2^n)$ time and polynomial space, and that Graph Colouring can be solved in $\mathcal{O}^*(2^n)$ time and space. The first result is due to Björklund and Husfeldt [27], the second to Björklund et al. [33].

Corollary 2.20 ([27]). *There is an algorithm that counts the number of perfect matchings in a graph G in $\mathcal{O}^*(2^n)$ time and polynomial space.*

Proof. We assume that n is even, otherwise the graph has no perfect matchings.

We consider counting $\frac{n}{2}$-tuples $(e_1, e_2, \ldots, e_{\frac{n}{2}})$, where each e_i is an edge from G for $1 \leq i \leq \frac{n}{2}$. We use Proposition 2.19 to count all such tuples in which each vertex in G is an endpoint of at least one edge e_i in the tuple. Notice that the set of edges in such a tuple forms a perfect matchings in G because, if the tuples contains $\frac{n}{2}$ edges, and if all n vertices are endpoint of some edge in the tuple, then every vertex is an endpoint of exactly one edge. Also, notice that every perfect matching corresponds to exactly $(\frac{n}{2})!$ such tuples by permuting the order of the edges in the tuple.

In this way, we can count the number of perfect matchings in G by evaluating the 2^n terms of the formula in Proposition 2.19 and dividing the resulting value c_k by $(\frac{n}{2})!$. To do so, we have to compute the values $a(X)$ that, in this case, express the number of edges in G that do not contain the vertices in X as endpoints. This can be done by simple counting.

Since the involved numbers are bounded by $2^n m^{\frac{n}{2}}$, they can be represented by a polynomial number of bits as $\log(2^n m^{\frac{n}{2}}) = n\log(2) + \frac{n}{2}\log(m)$. Hence, the operations on these numbers can be preformed in polynomial time. This proves the time bound of $\mathcal{O}^*(2^n)$. □

We note that this is the currently fastest polynomial space algorithm for #Perfect Matching. Using exponential space, this result was first improved to $\mathcal{O}(1.7315^n)$ time and space by Björklund and Husfeldt [27] and later to $\mathcal{O}(1.6181^n)$ time and space by Koivisto [208]. The given algorithm uses ideas that date back to Ryser, who used inclusion/exclusion to compute the number of perfect matchings in a bipartite graph [274].

Corollary 2.21 ([33]). *There is an algorithm that solves* Graph Colouring *in $\mathcal{O}^*(2^n)$ time and space.*

Proof. We will describe an algorithm that decides in $\mathcal{O}^*(2^n)$ time and space whether G can be coloured with k colours. The result then follows by repeating this algorithm for increasing values of k.

As already noted in Section 2.2.1, a colouring of a graph with k colours such that no two vertices with the same colour are adjacent equals a covering of the vertices of the graph by independent sets. Therefore, we consider counting k-tuples $(I_1, I_2, \ldots, I_k)$, where each I_i is an independent set in G for $1 \leq i \leq k$. If such a k-tuple of independent sets contains all vertices in G in some independent set I_i, then we can construct a colouring from this tuple by assigning to every vertex v in G some colour i such that $v \in I_i$. Notice that the resulting colouring has no adjacent vertices with the same colour because each set I_i is an independent set. Also, if a vertex exists in multiple

independent sets in the tuple, then we can choose the colour of the vertex to equal that of any of the independent sets that contain v.

We use Proposition 2.19 to count all such k-tuples $(I_1, I_2, \ldots, I_k)$ that contain all vertices in G in some independent set I_i. To be able to do so, we need the values $a(X)$ that now represent the number of independent sets in $G[V \setminus X]$. We can precompute these values $a(X)$ for all $X \subseteq V$ by dynamic programming across the subsets of V. This can be done by computing the number of independent sets $\iota(Y)$ in $G[Y]$ for any $Y \subseteq V$. The numbers $a(X)$ then follow by $a(X) = \iota(V \setminus X)$.

We compute $\iota(Y)$ by dynamic programming using the following recurrence that corresponds to either taking a vertex $v \notin Y$ in an independent set or not:

$$\iota(Y \cup \{v\}) = \iota(Y) + \iota(Y \setminus N(v)) \qquad \text{with:} \quad \iota(\emptyset) = 1$$

If we consider the subsets $Y \subseteq V$ in order of increasing size, the values $\iota(Y)$ for all $Y \subseteq V$ are computed using $\mathcal{O}^*(2^n)$ time and space.

Now, we can evaluate the formula in Proposition 2.19 in $\mathcal{O}^*(2^n)$ time to find the number c_k that equals the number of k-tuples $(I_1, I_2, \ldots, I_k)$ that correspond to a colouring. Clearly, if $c_k > 0$, then G is k-colourable, and if $c_k = 0$ G is not. This completes the proof. □

For references to the literature on other Graph Colouring results, see the discussion below Proposition 2.7.

We conclude this introduction to inclusion/exclusion algorithms by giving the classical result on Hamiltonian Cycle of Kohn et al. [207], Karp [200], and Bax [14].

A Hamiltonian Cycle in a graph G is a collection of n edges from G that form a single cycle which goes through every vertex exactly once when traversed. In other words, every vertex in G is incident to exactly two edges of the cycle.

Hamiltonian Cycle
Input: A graph $G = (V, E)$.
Question: Does G have a Hamiltonian cycle?

Proposition 2.22 ([14, 200, 207]). *There is an algorithm that solves* Hamiltonian Cycle *in* $\mathcal{O}^*(2^n)$ *time and polynomial space.*

Proof. The algorithm will fix a vertex $v \in V$ and repeat the procedure below for each of its neighbours $u \in N(v)$. This procedure will decide whether there exists a path of length $n-1$ in G that starts in v, ends in u, and visits every vertex in G exactly once. Clearly, G has a Hamiltonian cycle if and only if such a path exists between v and one of its neighbours u.

Define a *walk* of length k in a graph G to be a sequence of vertices $(v_1, v_2, \ldots, v_{k+1})$ from V with $\{v_i, v_{i+1}\} \in E$ for every consecutive pair of vertices in the sequence v_i, v_{i+1}. Notice that if such a walk has length $n-1$ and visits all vertices in G, then it must visit all vertices exactly once. Hence, we can decide if the required path of length $n-1$ in G that starts in v, ends in u, and visits every vertex in G exactly once exists by counting walks from v to u of length $n-1$ that visit all vertices in G.

For $X \subseteq (V \setminus \{u, v\})$, let $a(X)$ be the number of walks of length $n-1$ from v to u that do not contain any vertex in X. Now, the number of paths $p_{u,v}$ of length $n-1$

in G that start in v, end in u, and visit every vertex in G (exactly once) equals:

$$p_{u,v} = \sum_{X \subseteq (V \setminus \{u,v\})} (-1)^{|X|} a(X)$$

This formula is a variation of the inclusion/exclusion formula in Proposition 2.19.

We prove the above formula in the same way as in the proof of Proposition 2.19. If a walk visits all vertices, then it is counted only in the term of the sum corresponding to $X = \emptyset$. If a walk does not visit the vertices in some set $Y \neq \emptyset$ while it does visit all vertices in $V \setminus Y$, then it is counted in all terms of the formula that correspond to any $X \subseteq Y$. However, the total contribution of this walk to the value $p_{u,v}$ equals $\sum_{X \subseteq Y} (-1)^{|X|} \cdot 1 = \sum_{i=0}^{|Y|} \binom{|Y|}{i} (-1)^i 1^{|Y|-i} = (1-1)^{|Y|} = 0$ by Newton's binomial theorem. We conclude that $p_{u,v}$ equals the number of walks that visit every vertex (exactly once) as required.

We can compute $p_{u,v}$ in $\mathcal{O}^*(2^n)$ time and polynomial space by evaluating the above formula if we can compute the values $a(X)$ in polynomial time. We will complete the proof of this proposition by showing how to compute these values $a(X)$ next.

Let $X \subseteq (V \setminus \{u,v\})$ be fixed. For any $w \in V \setminus X$, let $a_X(w,l)$ be the number of walks of length l in $G[V \setminus X]$ that start in v and end in w. Clearly, $a_X(v,0) = 1$ and $a_X(w,0) = 0$ for any $w \in V \setminus (X \cup \{v\})$. For $l \geq 1$, we can compute $a_X(w,l)$ for any $w \in V \setminus X$ by dynamic programming over the following recurrence:

$$a_X(w,l) = \sum_{x:\ (w,x) \in E, x \notin X} a_X(x, l-1)$$

This recurrence considers all possible edges that can be added to the walks of length $l-1$ to construct a walk of length l.

We find the required value $a(X)$, by noticing that $a(X) = a_X(u, n-1)$. Because the dynamic programming requires the computation of $\mathcal{O}(n^2)$ values using a summation over $\mathcal{O}(n)$ terms, we conclude that the values $a(X)$ can be computed in polynomial time as required. This completes the proof. □

We note that this result has very recently been improved by Björklund who gives a randomised $\mathcal{O}(1.6576^n)$-time and polynomial-space algorithm for Hamiltonian Cycle with an exponentially small probability of failure [25].

This result based on counting walks to count Hamiltonian paths has been generalised by Nederlof to counting what he defines to be branching walks (treelike walks) to count Steiner trees [244]. This Steiner Tree algorithm has applications in algorithms for graph domination problems. For example, it is used as a subroutine in recent results by Abu-Khzam et al. on Connected Red-Blue Dominating Set and Connected Dominating Set [1].

2.4. Other Techniques

In the previous sections, we have introduced a series of common techniques in the design of exact exponential-time algorithms. The techniques that we have presented were chosen because we either use these techniques, or want to be able to name them without further reference. However, many more interesting techniques have been developed in the field. In this last section, we survey some of these techniques.

Solution-Driven Approaches. The most prominent category of techniques that we have not covered are solution-driven approaches such as local search and iterative compression. These techniques explore the search space of a problem by moving from one candidate solution to another. In this search, we know that if a solution exists, then it is found (or, with high probability in randomised variants). Searching for a solution in this way can be beneficial to constructing a solution from scratch, for example, by using a branch-and-reduce or dynamic programming algorithm.

Local search is a well-known technique that can be used to obtain heuristics for optimisation problems. It was introduced in the field of exact exponential-time algorithms by Schöning [280] who used it for algorithms for 3-Satisfiability. We illustrate this technique by giving a simple algorithm for 3-Satisfiability from [281], which is also due to Schöning.

Let the *Hamming distance* between two truth assignments to the variables in the instance be the number of variables on which the two assignments differ. For a given truth assignment one can check whether there exists a satisfying truth assignment at Hamming distance at most k from the current assignment in $\mathcal{O}^*(3^k)$ time in the following way. If the current assignment is a satisfying assignment, output Yes. Otherwise, the instance contains an unsatisfied clause containing at most three literals (l_1, l_2, l_3). To satisfy this clause, we need to change the value of at least one of these at most three literals. Therefore, we can branch into at most three subproblems where, in each generated subproblem, we change the value of a different literal from this set of three literals. If we repeat this branching step at most k times, we generate at most 3^k subproblems and find a satisfying assignment at Hamming distance at most k from our initial assignment if such an assignment exists.

We can turn this approach into an algorithm for 3-Satisfiability by noticing that any assignment lies at Hamming distance at most $\frac{n}{2}$ from the all-$True$ or the all-$False$ assignment. By searching for solutions up to distance at most $\frac{n}{2}$ from these two points, we obtain an $\mathcal{O}^*(3^{\frac{n}{2}}) = \mathcal{O}(1.7321^n)$-time and polynomial-space algorithm for 3-Satisfiability.

The currently fastest algorithms for 3-Satisfiability consider searching in the neighbourhood of an exponential number of starting points. This can either be done by choosing these starting points at random, or by defining some exponentially large set of starting points from which any assignment has Hamming distance at most some number k. A set of starting points of this second type is also called a *covering code* [98].

There exist many papers giving algorithms for 3-Satisfiability based on this approach: both deterministic algorithms [61, 98, 219, 276] and randomised algorithms [13, 185, 192, 193, 273, 280]. The currently fastest deterministic algorithm is due to Moser and Scheder [242] and runs in $\mathcal{O}(1.3334^n)$ time, and the currently fastest randomised algorithm is due to Hertli et al. [181] and runs in $\mathcal{O}(1.3210^n)$ time.

Iterative compression is another solution-driven technique. This technique comes from the field of parameterised algorithms (see Section 3.3) where it was introduced by Reed et al. [262]. In this field, this approach has many applications; see for example [76, 88, 105, 175, 187].

Iterative compression is a technique for optimisation (often minimisation) problems. The main idea is that if we are given a solution of size $k+1$ for a given problem, then we use an algorithm that either compresses it to better solution of size at most k,

or proves that no solution of size k exists. In exact exponential-time algorithms, this approach has been used by Fomin et al. on a number of problems among which a series of variants of Hitting Set [135]. Very recently, we have also used this approach to give the currently fastest parameterised algorithm for Feedback Vertex Set: a randomised $\mathcal{O}^*(3^k)$-time and polynomial-space algorithm [88].

Algebraic Approaches. Another important category of techniques that we have not covered are algebraic approaches. These approaches have recently shown to be successful on a series of problems.

For example, Björklund has shown how to encode candidate solutions to k-Dimensional Matching and Exact Cover by k-Sets into polynomials over a finite field of characteristic two [26]. This, combined with the evaluation of determinants of matrices which entries are these polynomials allows him to solve k-Dimensional Matching in $\mathcal{O}^*(2^{n(k-2)/k})$ time, and Exact Cover by k-Sets in $\mathcal{O}(1.496^n)$, $\mathcal{O}(1.642^n)$, $\mathcal{O}(1.721^n)$, $\mathcal{O}(1.771^n)$, $\mathcal{O}(1.806^n)$ time for $k = 3, 4, 5, 6, 7$, respectively [26]. All these algorithms are randomised algorithms with an exponentially small probability of failure that use polynomial space. We note that he also obtains fast algorithms for Exact Cover by k-Sets for larger values of k.

Very recently, Björklund has also shown how to use this approach to give a randomised $\mathcal{O}(1.6576^n)$-time and polynomial-space algorithm for Hamiltonian Cycle [25].

Another recent result using algebraic techniques is due to Lokshtanov and Nederlof. They show how to transform standard exponential-time and pseudo-polynomial-time dynamic programming algorithms for various problems such as Subset Sum into polynomial-space algorithms using the discrete Fourier transform [228].

Using Fast Polynomial-Time Algorithms on Exponentially Large Instances. The last category of techniques that we will treat in this section are those that use fast polynomial-time algorithms on exponentially-large instances to obtain fast exponential-time algorithms.

Fast matrix multiplication is an example of such a fast polynomial-time algorithm. One can multiply two $n \times n$ matrices in $\mathcal{O}(n^\omega)$ time, where ω is the *matrix multiplication constant*. Currently, $\omega < 2.376$ due to an algorithm by Coppersmith and Winograd [83] who improved the first non-trivial algorithm for the problem by Strassen [289].

Using this fast matrix-multiplication algorithm, Itai and Rodeh have shown that one can detect whether a graph has a triangle, and count the number of triangles, in $\mathcal{O}(n^\omega)$ time [190]. Williams uses this for an $\mathcal{O}^*(2^{\frac{\omega n}{3}}) = \mathcal{O}(1.7315^n)$-time algorithm for Maximum Cut by constructing instances of size $2^{\frac{n}{3}}$ in which triangles need to be detected [317]. Björklund also uses this approach for #Perfect Matching [27], although more recent, faster algorithms exist.

Fast matrix multiplication is also used by Dorn for a series of dynamic programming algorithms on branch decompositions [110]. The currently fastest subexponential-time algorithms for many problems on planar graphs and some related graph classes are based on this approach [113, 114]. We will also use this approach in Chapter 12.

Sorting is another example of a fast polynomial-time algorithm that is often used in exponential-time algorithms on exponential-size instances. We illustrate this approach by considering the example due to Woeginger [319] on the Subset Sum problem with

numbers $a_1, a_2, \ldots, a_n$ and target sum b (see the list of problems in Appendix B for a precise definition). This algorithm works as follows. Construct a $2^{\lfloor \frac{n}{2} \rfloor}$ size table with all possible sums corresponding to subsets of the numbers $a_1, a_2, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$. Since sorting can be done in $\mathcal{O}(n \log n)$ time, we can sort this list of sums in $\mathcal{O}^*(2^{\frac{n}{2}})$ time. Then, for each of the $2^{\lceil \frac{n}{2} \rceil}$ possible sums of the remaining numbers $a_{\lceil \frac{n}{2} \rceil}, a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n$, one can check in polynomial time whether this sum can be extended to the target value b by binary searching in the sorted table.

This technique is also known as *split and list* or *sort and search*. Examples of other algorithms based on this approach are those for Binary Knapsack in [186, 283], and Exact Hitting Set in [118]. Also, Fomin et al. use this approach to solve a number of $[\rho, \sigma]$-domination problems [139].

3 Complexity of Exact Exponential-Time Algorithms

An asymptotic upper bound on the running time of an exact exponential-time algorithm is usually expressed as an exponential function of a complexity parameter. The complexity parameter is a parameter that usually depends on the input, e.g., the number of vertices in the input graph, and that determines the size of the search space of the problem. The parameters are sometimes called the 'size' of the input, which can be regarded as a (slight) abuse of terminology.

In this chapter, we introduce the notion of a complexity parameter and other basic concepts from the complexity theory of exact exponential-time algorithms. We first introduce complexity parameters, certificates, and the definitions of the complexity classes $\mathcal{P}$, $\mathcal{NP}$, and $\#\mathcal{P}$ in Section 3.1. Then, in Section 3.2, we will give some standard conjectures and hypotheses for the theory of exponential-time algorithms that are used to show that subexponential-time algorithms do not likely exist for certain problems. Next, we treat the relation between exact exponential-time algorithms and parameterised algorithms and complexity in Section 3.3. In Section 3.4, we conclude with a few issues that relate to which exponential-time algorithms can be used to solve problems in practical situations.

We note that, in this chapter, all algorithms are considered deterministic algorithms when considering the existence or non-existence of algorithms with a certain type of upper bound on the running time for a given problem.

3.1. Complexity Parameters

P versus NP. Maybe the most interesting questions in complexity theory come from the perceived difference between the computational complexity of finding a solution to

a given problem and the computational complexity of checking that a given solution indeed is a solution. Most prominent among these questions is the '$\mathcal{P}$ versus $\mathcal{NP}$' question introduced in 1971 by Cook [80]. This problem is considered to be the most important question in the field by many researchers[1].

Slightly informally, as we omit the definitions of languages and Turing machines, the complexity classes $\mathcal{P}$ [79, 120] and $\mathcal{NP}$ [80] can be defined as follows.

Definition 3.1 (Polynomial Time). The complexity class $\mathcal{P}$ (*polynomial time*) is the class of all decision problems that can be solved in time polynomial in the number of bits required to represent a problem instance.

An example of a prominent problem in $\mathcal{P}$ that is used in this PhD thesis is the problem of deciding whether a graph contains a matching of size at least k [120].

Definition 3.2 (Non-Deterministic Polynomial Time). The complexity class $\mathcal{NP}$ (*non-deterministic polynomial time*) is the class of all decision problems which can be formulated in the following way:

$$\text{Given } x, \text{ decide whether there exists a } y \text{ with } |y| \leq \chi(x) \text{ such that } R(x, y).$$

where x is the input instance, y is a binary string, χ is a polynomial-time computable function with the property that $\chi(x)$ polynomially bounded in the size of x, and $R(x, y)$ is a polynomial-time computable relation. The binary string y is also called a *certificate* for the instance x.

Notice that a problem in $\mathcal{NP}$ has the property that there exists a polynomial-size 'proof' that shows that an instance is a YES-instance, namely the certificate y, whose correctness can be verified in polynomial time.

We also use the name $\mathcal{NP}$*-problem* for a problem in $\mathcal{NP}$. Many natural problems are $\mathcal{NP}$-problems. For example, consider the DOMINATING SET problem where we need to decide whether there exists a dominating set of size at most k in a graph G. This problem is in $\mathcal{NP}$ as a given dominating set $D \subseteq V$ of size at most k is a certificate for this problem. Finding such a certificate can be hard. For example, the currently fastest algorithm for this problem requires $\mathcal{O}(1.4969^n)$ time (Chapter 5). However, checking whether D actually is a dominating set of size at most k is easy and can be done in polynomial time.

Definition 3.3 (Polynomial-Time Many-One Reduction). A *polynomial-time many-one reduction* from a problem P to a problem Q is a polynomial-time transformation that takes an instance x of P and outputs an instance y of Q of size polynomial in the size of x with the property that x is a YES-instance of P if and only if y is a YES-instances of Q.

Cook [80], and independently Levin [222], proved that there exist polynomial-time many-one reductions from any problem in $\mathcal{NP}$ to SATISFIABILITY. As a result, all problems in $\mathcal{NP}$ can be solved in polynomial time if SATISFIABILITY can be solved in polynomial time.

[1] It is one of the Millennium Prize Problems of the Clay Mathematics Institute.

A subclass of $\mathcal{NP}$ is the class of $\mathcal{NP}$-*complete* problems introduced by Cook [80]. The $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$ from the viewpoint of polynomial-time algorithms.

Definition 3.4 (NP-complete). A problem P is $\mathcal{NP}$-complete if P is in $\mathcal{NP}$ and there exist polynomial-time many-one reductions from all problems in $\mathcal{NP}$ to P.

Thus, Satisfiability is $\mathcal{NP}$-complete. Moreover, if some $\mathcal{NP}$-complete admits a polynomial-time algorithm, then all problems in $\mathcal{NP}$ admit polynomial-time algorithms (in other words $\mathcal{NP} = \mathcal{P}$). Although $\mathcal{P} \subseteq \mathcal{NP}$, many researchers believe that $\mathcal{P} \neq \mathcal{NP}$ which means that $\mathcal{NP}$-complete problems cannot be solved in polynomial time; see also Section 3.2.

Many natural problems are $\mathcal{NP}$-complete. This includes many graph domination problems such as Dominating Set, Independent Set, and Edge Dominating Set, but also many other problems such as Set Cover, 3-Satisfiability and Hamiltonian Cycle [162, 199]. A superclass of the class of $\mathcal{NP}$-complete problems is the class of $\mathcal{NP}$-hard problems. This class of $\mathcal{NP}$-*hard* problems contains all problems that are at least as hard as the problems in $\mathcal{NP}$. If an $\mathcal{NP}$-hard problem can be solved in polynomial time, then all problems in $\mathcal{NP}$ can be solved in polynomial time also. As $\mathcal{NP}$-complete problems are decision problems, their (non-trivial[2]) maximisation or minimisation variants are often $\mathcal{NP}$-hard. This includes problems like 'find a dominating set of minimum size in a given graph' or 'what is the size of the maximum independent set in a given graph'. For more on $\mathcal{NP}$-completeness and $\mathcal{NP}$-hardness and many examples of $\mathcal{NP}$-complete problems, see for example the book by Garey and Johnson [162].

Complexity Parameters of NP-problems. We are now ready to introduce the concept of a complexity parameter. This concept was introduced by Impagliazzo et al. who observed that it is important to include a parameter (that they call the *complexity parameter*) in the problem description when considering exponential-time algorithms for $\mathcal{NP}$-problems [189].

Definition 3.5 (Complexity Parameter). Given a description of an $\mathcal{NP}$-problem as in Definition 3.2, the function $\chi(x)$ is called the associated *complexity parameter*.

Before discussing some important properties of a complexity parameter, we first consider an example. We have already seen that a concrete dominating set is a possible certificate for the Dominating Set problem. Such a set can be encoded as a binary string in which each bit corresponds to a vertex and encodes whether this vertex is in the dominating set or not. In this case, the relation $R(x, y)$ checks whether y encodes a dominating set and whether this dominating set is of size at most k. The resulting certificate will have length equal to the number of vertices in the graph. Hence, the number of vertices n is a complexity parameter $\chi(x)$ for Dominating Set.

Complexity parameters are used to express the running times of exact exponential-time algorithms. This is useful because a running time expressed in this way can

[2]With trivial maximisation or minimisation variants of $\mathcal{NP}$-complete problems, we mean problems like finding a maximum size dominating set or a minimum size independent set.

directly be related to the trivial brute-force algorithms for this problem and complexity parameter combination. This in the following sense. Given an $\mathcal{NP}$-problem P and a complexity parameter χ, the *trivial brute-force algorithm* for this problem and complexity parameter combination is defined to be the algorithm that solves a problem instance x in the following way: it enumerates all strings y of length up to $\chi(x)$, and it checks whether $R(x, y)$ for each such string y, i.e., whether y is a certificate for the instance x. Clearly, this trivial brute-force algorithm runs in $\mathcal{O}^*(2^{\chi(x)})$ time. One of the main goals in exact exponential-time algorithms is to design algorithms that improve upon this straightforward time bound.

Because a complexity parameter is an upper bound on the size of a certificate for a given problem, this complexity parameter follows directly from the way in which the certificates are encoded as binary strings. For the purpose of designing exact exponential-time algorithms, however, the exact details of this encoding are not important. Therefore, the details of the encoding are almost always omitted, and only the complexity parameters used to compare running times are mentioned.

Complexity parameters are not unique. That is, for many graph problems one could use the number of vertices n, the number of edges m, or a parameter χ that is proportional to the number of bits used to represent an input graph in some given representation. To illustrate this, we give two examples.

Example 3.1. Consider the Vertex Cover problem (see Appendix B). For this problem, we could use the number of vertices n as a complexity parameter and encode a subset of the vertices in a certificate, or we could use the number of edges m as a complexity parameter and let a certificate encode, for each edge, which endpoint of the edge is used to cover it.

Example 3.2. Consider the Edge Dominating Set problem (see Section 1.3). The obvious complexity parameter for this problem is the number of edges m because an edge dominating set is an edge subset that can straightforwardly be encoded using m bits. However, we can also use the number of vertices n as a complexity parameter for this problem. We can do so by letting a certificate encode a subset of the vertices that represents the set of endpoints Y of the edge dominating set that is a solution.

To see that this works, first recall that we can assume that no edges in a minimum edge dominating set share endpoints. We have already used this fact in the algorithm of Proposition 2.13; it is based on a standard replacement trick from [177] whose details can be found in Corollary 6.3. Second, notice that the neighbourhood relation that defines the domination criterion in the Edge Dominating Set problem is based on the set of endpoints of the edge dominating set: it does not require knowledge of the actual dominating edges themselves. Using these properties, the relation $R(x, y)$ can verify in polynomial time whether the vertices in Y are the endpoints of a sufficiently small edge dominating set. It does so by checking whether whether $\frac{|Y|}{2} < k$, whether all edges in G have an endpoint in Y, and by checking whether there exists a perfect matching in $G[Y]$. Such a perfect matching can be computed in polynomial time [120] and corresponds to a solution edge dominating set. See Section 6.2 for further details.

We conclude this discussion of complexity parameters of $\mathcal{NP}$-problems by pointing out that, although complexity parameters are defined similar to Definition 3.5 in the

literature (e.g., see [149, 319]), often complexity parameters are used that do not correspond to this definition. Consider, for example, the Hamiltonian Cycle problem. For this problem, one could use the number of edges m as a complexity parameter and encode edge sets, or one could use $\log(n!)$ as a complexity parameter and encode an ordering of the vertices. However, it is not clear that the number of vertices n can be used as a complexity parameter, while often this parameter n is used for analysing algorithms for this problem; see for example Proposition 2.22. In any case, the complexity parameter used should be a function of a complexity parameter as defined in Definition 3.5.

For graph domination problems, this ambiguity in the precise definition of a complexity parameter is often not an issue. This is because graph domination problems are subset problems for which one can usually use a bound on the number of elements of the ground set as a complexity parameter. That is, the number of vertices n for a vertex-subset problem, and the number of edges m for an edge-subset problem.

Counting Problems and Complexity Parameters. Let us now consider counting variants of $\mathcal{NP}$-problems. The complexity class of these problems is the class $\#\mathcal{P}$ introduced by Valiant [298].

Definition 3.6 (Sharp-P). The complexity class $\#\mathcal{P}$ (*sharp-*$\mathcal{P}$) is the class of all counting problems corresponding to $\mathcal{NP}$-problems. That is, all counting problems which can be formulated in the following way:

$$\text{Given } x\text{, compute how many } y \text{ exist with } |y| \leq \chi(x) \text{ such that } R(x, y).$$

where x is the input instance, y is a binary string, χ is a polynomial-time computable function with the property that $\chi(x)$ polynomially bounded in the size of x, and $R(x, y)$ is a polynomial-time computable relation.

A problem in $\#\mathcal{P}$ is often denoted by placing the $\#$-sign in front of the corresponding problem from $\mathcal{NP}$. For example, the $\#\mathcal{P}$-problem of computing the number of minimum dominating sets in a graph G is denoted by #Dominating Set, and the $\#\mathcal{P}$-problem of computing the number of perfect matchings in a graph G is denoted by #Perfect Matching.

Valiant also introduced the class $\#\mathcal{P}$-complete [298]. We will not give a precise definition of $\#\mathcal{P}$-complete problem here. The intuition behind this class of problems is that it contains the hardest problems from $\#\mathcal{P}$ from the viewpoint of polynomial-time computability. If a $\#\mathcal{P}$-complete problem has a polynomial-time algorithm, then all problems in $\#\mathcal{P}$ have polynomial-time algorithms. This is similar to the way in which $\mathcal{NP}$-complete problems are the hardest problems from the class $\mathcal{NP}$.

If a problem is $\#\mathcal{P}$-complete, then it most likely does not have a polynomial-time algorithm. This is because the counting variants of many $\mathcal{NP}$-complete problems are $\#\mathcal{P}$-complete, and thus a polynomial-time algorithm for a $\#\mathcal{P}$-complete problem would imply polynomial-time algorithms for all problems in $\mathcal{NP}$. Maybe somewhat surprisingly, the #Perfect Matching problem is $\#\mathcal{P}$-complete [298], while the problem of deciding whether there exists a perfect matching in a given graph G is a problem in $\mathcal{P}$ [120]. For more details and some examples of reductions that prove $\#\mathcal{P}$-completeness of certain problems see [194, 298].

In this PhD thesis, we will also consider exact exponential-time algorithms for problems in $\#\mathcal{P}$. To express the running times of these algorithm, we again use a complexity parameter. In the case of $\#\mathcal{P}$-problems, this is the function $\chi(x)$ defined in Definition 3.6. This complexity parameter also allows us to compare the running time of an algorithm to a trivial brute-force algorithm, namely, the algorithm that enumerates all binary strings y of length up to $\chi(x)$ and counts the number of such strings that satisfy the relation $R(x, y)$.

3.2. Hypotheses in Exponential-Time Complexity

Since we aim at designing algorithms whose asymptotic running time is as small as possible, the first question one could ask when considering exponential-time algorithms is whether exponential time is really necessary. That is, do we have any evidence that no faster algorithms exist? I.e., no faster algorithms such as polynomial-time algorithms or subexponential-time algorithms. This section will survey some complexity-theoretic assumptions on which such claims can be based.

We have already mentioned the following widely believed hypothesis in the previous section.

Complexity Theoretic Hypothesis 3.7. $\mathcal{P} \neq \mathcal{NP}$

A consequence of this widely believed hypothesis is that we may assume that no polynomial-time algorithms exists for $\mathcal{NP}$-complete problems and $\#\mathcal{P}$-complete problems [80, 298].

This hypothesis, however, is insufficient for our purposes as it excludes only polynomial-time algorithms while subexponential-time algorithms also exist for many $\mathcal{NP}$-complete and $\#\mathcal{P}$-complete problems. Examples of such problems include many problems on planar graphs [110, 111, 114]. For an example problem on general graphs, see [51]. We also note that the classical $\mathcal{O}^*(2^n)$-time algorithm of Kohn et al. [207] (also [14, 200]) for Hamiltonian Cycle that can be found in Proposition 2.22 is a subexponential time algorithm when we use the complexity parameter $\chi(x) = \log(n!)$.

The following hypothesis can be used to give evidence for the non-existence of subexponential-time algorithms for specific problems. It was first formulated by Impagliazzo and Paturi in [188].

Complexity Theoretic Hypothesis 3.8 (Exponential-Time Hypothesis - ETH). There exists a constant $c > 1$ such that there exists no algorithm for 3-Satisfiability that uses only $\mathcal{O}(c^n)$ time.

This hypothesis is equivalent to the statement that there exist no algorithm for 3-Satisfiability that, for all $\epsilon > 0$, runs in $\mathcal{O}(2^{\epsilon n})$ time. That is, under this hypothesis, the worst case running time of an algorithms for 3-Satisfiability is at least singly exponential in n.

The Exponential-Time Hypothesis, or ETH, is not merely related to the complexity of the 3-Satisfiability problem. It is related to the syntactically defined problem class $\mathcal{SNP}$ (Strict $\mathcal{NP}$) [250]. Similar to the class $\mathcal{NP}$, whose hardest problems from the viewpoint of polynomial-time algorithms form the subclass of $\mathcal{NP}$-complete prob-

lems, the class $\mathcal{SNP}$ has a subclass of $\mathcal{SNP}$-*complete* problems that are the hardest problems in $\mathcal{SNP}$ from the viewpoint of subexponential-time algorithms. This class $\mathcal{SNP}$-complete is defined under SERF-reductions [189].

Definition 3.9 (SERF-reduction). A *SubExponential Reduction Family* from a problem P with complexity parameter χ_p to a problem Q with complexity parameter χ_q is an algorithm[3] that does the following for every $\epsilon > 0$: given an instance x_p of the problem P, the algorithm produces a series of instances $x_{q_1}, x_{q_2}, \ldots, x_{q_k}$ of problem Q in $\mathcal{O}(2^{\epsilon \cdot \chi_p(x_p)})$ time such that the algorithm can compute the solution to x_p given the solutions to the instances $x_{q_1}, x_{q_2}, \ldots, x_{q_k}$ within the given time bound. In this reduction, each instance x_{q_i} of Q in the series must satisfy two requirements:

- the complexity parameter $\chi_q(x_{q_i})$ must satisfy $\chi_q(x_{q_i}) \leq c_\epsilon \chi_p(x_p)$ for some constant c_ϵ that may depend arbitrarily on ϵ.
- the size of x_{q_i} is polynomial in the size of x_p.

We now show that SERF-reductions preserve subexponential-time solvability.

Lemma 3.10 ([189]). *If there exist a SERF-reduction from a problem P with complexity parameter χ_p to a problem Q with complexity parameter χ_q, then the problem P is solvable in subexponential time in χ_p if the problem Q is solvable in subexponential time in χ_q.*

Proof. Since Q is solvable in time subexponential in χ_q, there exists an algorithm for Q that given an instance x_q of Q, for any $\delta > 0$, solves x_q in $\mathcal{O}(2^{\delta \cdot \chi_q(x_q)})$ time.

An instance x_p of P can be solved in $\mathcal{O}(2^{\epsilon \cdot \chi_p(x_p)})$ time in the following way. Apply the algorithm of the SERF-reduction using any $\epsilon' < \epsilon$. This generates at most $\mathcal{O}(2^{\epsilon' \cdot \chi_p(x_p)})$ instances of Q with complexity parameters at most $c_{\epsilon'} \cdot \chi_p(x_p)$. These instances can be solved within the required time bound by using the subexponential-time algorithm for Q in time $\mathcal{O}(2^{\delta \cdot c_{\epsilon'} \cdot \chi_p(x_p)})$ for some small enough $\delta > 0$ such that $2^{\epsilon' \cdot \chi_p(x_p)} 2^{\delta \cdot c_{\epsilon'} \cdot \chi_p(x_p)} < 2^{\epsilon \cdot \chi_p(x_p)}$. □

Using these SERF-reductions, we can now define the classes $\mathcal{SNP}$-complete and $\mathcal{SNP}$-hard. A problem P with complexity parameter χ_p is $\mathcal{SNP}$-hard under SERF-reductions if all problems in the class $\mathcal{SNP}$ are SERF-reducible to P with complexity parameter χ_p. We note that this is a correct definition since $\mathcal{SNP}$ is a syntactically defined subclass of $\mathcal{NP}$, and thus problems in $\mathcal{SNP}$ are provided with complexity parameters based on their description as an $\mathcal{NP}$-problem.

Definition 3.11 (SNP-complete). A problem P with complexity parameter χ_p is $\mathcal{SNP}$-complete if P with complexity parameter χ_p is in the class $\mathcal{SNP}$ and P with complexity parameter χ_p is $\mathcal{SNP}$-hard.

It is easy to see from these definitions that if there exists a subexponential-time algorithm for any $\mathcal{SNP}$-complete or $\mathcal{SNP}$-hard problem P with complexity parameter χ_p, then all problems in $\mathcal{SNP}$ admit subexponential-time algorithms with the complexity parameters that arise from their syntactical definitions. As 3-Satisfiability with the

[3] Turing reduction.

number of variables n as complexity parameter is $\mathcal{SNP}$-complete [189], the Exponential-Time Hypothesis says that no subexponential-time algorithm exists for any $\mathcal{SNP}$-hard (and thus also for any $\mathcal{SNP}$-complete) problem and complexity parameter combination.

Assuming the Exponential-Time Hypothesis, Impagliazzo et al. show that no subexponential-time algorithms exist for a large number of problem and complexity parameter combinations by showing that these problems are (size-constrained[4]) $\mathcal{SNP}$-complete [189]. Such results are obtained for k-Colouring with $k \geq 3$ and the number of vertices n as complexity parameter, Independent Set with the number of vertices n as complexity parameter, and k-Set Cover with $k \geq 2$ and the number of sets n as complexity parameter. Impagliazzo et al. also show that Hamiltonian Cycle with the number of edges m as complexity parameter is $\mathcal{SNP}$-hard [189].

One of the most notable SERF-reductions is the following result by Impagliazzo et al. known as the *sparsification lemma* [189].

Lemma 3.12 (Sparsification Lemma). *k-Satisfiability with the number of variables n as complexity parameter is SERF-reducible to k-Satisfiability with the number of clauses m as complexity parameter.*

This lemma is a useful tool in proving the $\mathcal{SNP}$-hardness of many problem and complexity parameters. This because, for any fixed k, k-Satisfiability with the number of variables n as complexity parameter is $\mathcal{SNP}$-complete since it contains the 3-Satisfiability problem. It then directly follows from the sparsification lemma that k-Satisfiability with complexity parameter m or complexity parameter $n+m$ is also $\mathcal{SNP}$-hard under SERF-reductions. As a result, a SERF-reduction used to prove $\mathcal{SNP}$-hardness for some problem P can be a SERF-reduction to k-Satisfiability with any non-negative linear combination of n and m as a complexity parameter. We will, for example, use this in Section 4.4 when we prove that, under the Exponential-Time Hypothesis, no subexponential-time algorithm exists for Partition Into Triangles on graphs of maximum degree four.

In some cases, a standard polynomial-time many-one reduction used to prove $\mathcal{NP}$-hardness for a problem can also be used as a SERF-reduction. Such a reduction takes an instance x_p of an $\mathcal{NP}$-hard problem P and transforms it in polynomial time into an instance x_q of the problem Q in such a way that x_p is a Yes-instance if and only if x_q is a Yes-instance. Notice that such a polynomial-time reduction often does not preserve subexponential-time solvability as $\chi_p(x_p)$ can have super-linear, yet still polynomial, size in $\chi_q(x_q)$. In cases in which the reduction guarantees that $\chi_p(x_p) \leq c \cdot \chi_q(x_q)$ for some constant c, however, the reduction does indeed preserve subexponential-time solvability. In such a case, it is a SERF-reduction that generates only one instance and that runs in polynomial time. An example of such a simple reduction is the reduction from 3-Satisfiability to Vertex Cover in [162].

We now consider an important consequence of the Exponential-Time Hypothesis. Let c_k be defined in the following way:

$$c_k = \inf\{c \mid \text{there exists an } \mathcal{O}(c^n)\text{-time algorithm for } k\text{-Satisfiability}\}$$

[4]Size-constrained $\mathcal{SNP}$ is a syntactically slightly different class that is a small generalisation of $\mathcal{SNP}$. From the viewpoint of the existence of subexponential-time algorithms, they are equivalent. For the technical details, see [189].

Impagliazzo and Paturi [188] have proved that, assuming the ETH, the values c_k form a sequence that increases infinitely often.

Since the trivial brute algorithm for SATISFIABILITY solves this problem in $\mathcal{O}^*(2^n)$ time, we know that $c_k \leq 2$ for all $k \geq 3$. A natural question is whether this increasing sequence converges to 2, or to some smaller number $c_\infty < 2$.

This leads us to the following hypothesis due to Dantsin and Wolpert [100] based on the above question raised by Impagliazzo et al. [68, 188].

Complexity Theoretic Hypothesis 3.13 (Strong Exponential–Time Hypothesis). There is no algorithm that solves SATISFIABILITY in $\mathcal{O}((2-\epsilon)^n)$ time, for any $\epsilon > 0$.

This assumption is often used in parameterised complexity. Under this hypothesis, Pătraşcu and Williams [257] have shown that, for every $\epsilon > 0$, no algorithm exists for the parameterised problem k-DOMINATING SET that runs in time $\mathcal{O}(n^{k-\epsilon})$. Lokshtanov et al. [227] have also used this hypothesis to show that, among others, the bases in the exponents of the running times of some of our algorithms in Chapter 11 are optimal.

3.3. Relations to Parameterised Algorithms

A branch of algorithmic research that is strongly related to exact exponential-time algorithms is that of *parameterised algorithms*. In this area of research, part of the input of a problem is a non-negative number k that is called the *parameter*. In general, the value of this parameter k will be significantly smaller compared to a complexity parameter for the problem (as defined in Section 3.1). The computational complexity of a so-called parameterised problem is studied with respect to this parameter k. We will make this more clear below. For an introduction into parameterised algorithms and their complexity, we refer the reader to [117, 134, 247].

Definition 3.14 (Parameterised Problem). A *parameterised problem* is a decision problem in which the input has two parts: the first part is the instance x, and the second part is a non-negative number k that is called the *parameter*.

In the study of parameterised algorithms, one designs algorithms for parameterised problems and expresses the worst-case running times of these algorithms by functions of two parameters: the problem parameter k and a parameter n that is proportional to the size of an instance x.

Definition 3.15 (Fixed-Parameter Tractable). The complexity class $\mathcal{FPT}$ (*fixed-parameter tractable*) contains all parameterised problems which can be solved in time $\mathcal{O}(f(k)p(n))$ where k is the parameter, n is the size of an instance, f is any computable function, and p is a polynomial.

An algorithm for a parameterised problem that runs in $\mathcal{O}(f(k)p(n))$ time as in Definition 3.15 is called a *fixed-parameter-tractable algorithm* or $\mathcal{FPT}$*-algorithm*.

In many studies, the parameter k that is used is a bound on the size of a solution; for example, the k-DOMINATING SET problem asks to find a dominating set of size at most k. Examples of problems that are fixed-parameter tractable when parameterised

by the size of a solution include[5]: k-Vertex Cover [64, 74], where one asks for a vertex cover of size at most k, k-Nonblocker [104] (see also Section 5.5), where one asks for a dominating set of size at most $n - k$, and k-Edge Dominating Set [23, 256], where one asks for an edge dominating set of size at most k. Other parameterisations, however, are also common. Examples include the size of a given vertex cover in the graph [129], or the number of leaves of a spanning tree in the graph [128]. Other parameterisations are based on graph decompositions such as the treewidth of the graph; we also call these parameterised algorithms *graph-decomposition-based algorithms*; see for example [2, 44, 86, 110, 296] or the algorithms in Chapters 11-13.

The fact that $\mathcal{FPT}$-algorithms and exact exponential-time algorithms are related can be seen from the numerous results in which an algorithm of one of both types is used as a subroutine to produce an algorithmic result of the other type.

Examples of exact exponential-time algorithms based on $\mathcal{FPT}$-algorithms parameterised by the solution size include algorithms for a large series of problems by Raman et al. [259] and algorithms for 3-Hitting Set by Wahlström [314]. Other examples based on different parameters include the treewidth-based algorithms in [138, 147, 165, 204], in Section 2.2.2, and in Chapters 8-10.

Many results for parameterised algorithms that use exact exponential-time algorithms also exist. Most examples of such algorithms are $\mathcal{FPT}$-algorithms that exploit the existence of a small (linear-size) *kernel* of the parameterised problem.

Definition 3.16 (Kernel). A *kernel* or *kernelisation algorithm* for a parameterised problem P is a polynomial-time algorithm that, given an instance x of P with parameter k, transforms x into an instance x' of P with parameter k' such that:

- x is a Yes-instance of P if and only if x' is a Yes-instance of P.
- the size of x' is bounded by a computable function $f(k)$.
- the parameter k' is bounded by a function of k.

The function $f(k)$ in Definition 3.16 is called the *size* of the kernel. A kernel of size $f(k)$ is also called an $f(k)$-kernel. The size of a kernel can be an upper bound on various size-parameters of an instance such as the number of vertices in a graph or the number of bits required to represent the instance x'. We note that if we want to use small kernels in combinations with exact exponential-time algorithms to obtain a fixed-parameter-tractable algorithm, then the size of the kernel should be an upper bound on a complexity parameter for P. In this case, the exact exponential-time algorithm can solve the instance x' in time bounded by an (exponential) function in $\chi(x')$, which now satisfies $\chi(x') \leq f(k)$.

One can often use small linear kernels in combination with fast exact exponential-time algorithms to create fast $\mathcal{FPT}$-algorithms. That is, to create $\mathcal{FPT}$-algorithms where the function $f(k)$ in the bound on the running time grows slowly. Consider a problem for which an $f(k)$-kernel and an $\mathcal{O}(c^n)$-time algorithm are known, where f is a linear function, $f(k) = \alpha k$, and n is the corresponding complexity parameter with $n \leq \alpha k$. The combination of the kernelisation algorithm and the exact exponential-time algorithm now runs in $\mathcal{O}^*(c^{\alpha k})$ time. If α and c are small enough, then this construction can give the fastest known $\mathcal{FPT}$-algorithms for a problem. We will use

[5]For each of these problems, we give only a reference to the first $\mathcal{FPT}$-algorithm for this problem and a reference to the currently fastest parameterised algorithm.

this construction in this PhD thesis to construct fast parameterised algorithms for k-Nonblocker, k-Vertex Cover on graphs of maximum degree three, and k-Set Splitting; see Section 5.5, Section 7.2, and Section 9.3, respectively.

Although fixed-parameter-tractable algorithms exist for many parameterised problems, there also exist parameterised problems which most likely do not belong to $\mathcal{FPT}$, i.e., parameterised problems for which $\mathcal{FPT}$-algorithms most likely do not exist. To show this, Downey and Fellows identified a chain of complexity classes that seem to extend $\mathcal{FPT}$ [115]:

$$\mathcal{FPT} \subseteq \mathcal{W}[1] \subseteq \mathcal{W}[2] \subseteq \cdots \subseteq \mathcal{W}[P] \subseteq \mathcal{XP}$$

Well-known examples of problems in these complexity classes are k-Independent Set that is in $\mathcal{W}[1]$ and k-Dominating Set that is in $\mathcal{W}[2]$.

We will not provide the details of this $\mathcal{W}$-hierarchy, but mention only some properties. The complexity classes are defined under $\mathcal{FPT}$-reductions: reductions that preserve fixed-parameter tractability and that run in $\mathcal{O}(f(k)p(n))$ time. Under these $\mathcal{FPT}$-reductions, there exist the notions of $\mathcal{W}[1]$-completeness and $\mathcal{W}[1]$-hardness (or $\mathcal{W}[i]$-completeness or $\mathcal{W}[i]$-hardness for any $i \geq 1$) defined in roughly the same way as $\mathcal{NP}$-completeness for polynomial-time many-one reductions, or $\mathcal{SNP}$-completeness under SERF-reduction.

If a problem is $\mathcal{W}[1]$-hard, then it is most likely not solvable by an $\mathcal{FPT}$-algorithm due to the following complexity-theoretic hypothesis.

Complexity Theoretic Hypothesis 3.17. $\mathcal{FPT} \neq \mathcal{W}[1]$

Because k-Independent Set is in $\mathcal{W}[1]$-complete and k-Dominating Set is $\mathcal{W}[2]$-complete [116], these two problems are most likely not in $\mathcal{FPT}$.

We conclude this section by mentioning a relation between the above hypothesis and the Exponential-Time Hypothesis that is due to Chen et al. [71].

Theorem 3.18 ([71]). *If the ETH holds, then* $\mathcal{FPT} \neq \mathcal{W}[1]$.

3.4. Practical Issues: Time, Space, and Parallelisation

We conclude this introductionary chapter by briefly discussing some practical complexity issues that are important when *implementing* an exponential-time algorithm. Although these issues are important, we point out that this PhD thesis primarily is a theoretical search for asymptotically-fast exact exponential-time algorithms.

Time versus Space. It is a well-known phenomenon that an algorithm stops being effective in practice when it uses more space than the amount of main memory available on the machine. In this case, the computer starts using hard disk space, and because of the time required for the resulting swapping between both memory sources, the algorithm's execution will slow down tremendously. Consequently, Woeginger states that "algorithms using exponential space are absolutely useless in real-life applications" [321]. However, other papers consider using $\mathcal{O}^*(s^n)$ space and $\mathcal{O}^*(t^n)$ time not to be problematic in practice when s is significantly smaller than t; for example [89].

We want to stress that algorithms that use exponential space can be used in practice. A good example is the $\mathcal{O}^*(2^n)$-time-and-space algorithm that computes the treewidth of a graph [41].

In practical situations, the amount of available main memory space is usually known before running the algorithm. We think that a good exponential-space algorithm for practical purposes must be able to use all the space available to it to speed up its execution. This type of algorithm is usually based on combinations of techniques: it first uses a polynomial-space approach (such as branch and reduce, or the more general divide and conquer) that generates smaller subproblems until these subproblems are small enough to be solved by an exponential-space approach (such as dynamic programming). A nice example of such a result is obtained for the Travelling Salesman Problem and other permutation problems[6] by Koivisto and Parviainen [209]. They consider algorithms running in time $\mathcal{O}^*(t^n)$ and space $\mathcal{O}^*(s^n)$ where one can choose t and s on, or for some specific values even below, the curve defined by $ts = 4$. For more examples, see [149].

Another practical consideration is that the precise value of a proven upper bound on the running time of an algorithm is not interesting as long as the algorithm terminates in reasonable time. This consideration applies, for example, to many of the exponential-space algorithms that we give in Chapters 8-10. These algorithm are based on carefully balancing a polynomial-space branch-and-reduce approach with an exponential-space dynamic programming approach. If these algorithms are modified such that they only switch to the dynamic programming approach when there is sufficiently many main memory available, and otherwise continue branching, then these algorithms certainly qualify as practical algorithms.

Parallelisation. Another interesting practical issue is whether an exponential-time algorithm can benefit from using a number of parallel processors. This is of particular interest because over the last few years multi-core processors have become standard. Most surveys or PhD theses do not address this issue while discussing practical issues of exact exponential-time algorithms.

While we will not elaborate on which algorithms can benefit from parallelisation and which algorithms are inherently difficult to run in parallel, we do want to mention that parallelisation is possible for most algorithms given in this thesis. First of all, most algorithms given in Chapters 4-10 are branch-and-reduce algorithms (that do not use memorisation); see also Section 2.1. These algorithms are easily run in parallel because different subproblems that are generated by the branching rules can be solved independently on different processors. Secondly, the algorithms given in Chapters 11-13 are dynamic programming algorithms that need to compute exponential-size tables at each step. For these algorithms, the entries in the tables can be divided into equal-sized parts that can be computed independently in most cases.

[6] Problems for which the corresponding certificates are a permutation of a given set of elements, for example, a permutation of the vertices of a graph.

4

A Very Fast Exponential-Time Algorithm for Partition Into Triangles on Graphs of Maximum Degree Four

We conclude Part I of this thesis with a result to illustrate the fact that exponential-time algorithms do not necessarily need to be slow: exponential-time algorithms can be practical algorithms. This fact is expressed beautifully in the following quote by Alan J. Perlis, the first Turing Award winner:

> *"For every polynomial-time algorithm you have, there is an exponential algorithm that I would rather run."*

This quote is illustrated by Richard J. Lipton in his weblog [226] in the following way. "His point is simple: if your algorithm runs in n^4 time, then an algorithm that runs in $n2^{n/10}$ time (alternatively denoted as $n1.07178^n$ time) is faster if for example $n = 100$."

Woeginger made the same observation for $\mathcal{NP}$-hard problems instead of polynomial time solvable problems in his well-known survey on exact exponential-time algorithms [319]. Woeginger considers the fact that algorithms for $\mathcal{NP}$-hard problems with exponential running times may actually lead to practical algorithms: he compares the running times of $\mathcal{O}(n^4)$ with $\mathcal{O}(1.01^n)$.

[†]This chapter is joint work with Marcel E. van Kooten Niekerk and Hans L. Bodlaender. This research started in Marcel E. van Kooten Niekerk's research project for the course "Seminar Exact algorithms for hard problems" Hans L. Bodlaender and I taught in first period of 2008/2009. The results have changed much since that time. The chapter contains results of which a preliminary version has been presented at the 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011) [309]. The full version is also available as technical report UU-CS-2010-005 [308].

However, we are not aware of any results on natural $\mathcal{NP}$-hard problems with exponential-time algorithms with running times anywhere near $\mathcal{O}(1.01^n)$ without involving huge polynomial factors (either visible, or hidden in the notation, or hidden in the decimal rounding of the exponent in the big-$\mathcal{O}$). 'Very fast' exponential-time algorithms exist for problems such as Independent Set restricted to graphs in which 99% of the vertices have degree at most two. However, we do not consider this to be a natural problem because we can reduce an instance of this artificial problem in polynomial time to an equivalent instance of Independent Set in which only 1% of the vertices remain (for the polynomial time reductions see Chapter 7). Then, the trivial brute-force $\mathcal{O}(n2^n)$ algorithm for Independent Set gives an algorithm for this artificial problem running in $\mathcal{O}(n2^{n/100}) = \mathcal{O}(1.0070^n)$ time. We note that for the problem studied in this chapter, no polynomial time transformation that greatly reduce the instance size are known from the problem on graphs of maximum degree four to the problem on general graphs (and most likely none are possible).

In this chapter, we will give a very fast exponential-time algorithm for the Partition Into Triangles problem restricted to graphs of maximum degree four, running in $\mathcal{O}(1.02445^n)$ or $\mathcal{O}(2^{n/28.69})$ time. This result is further improved to $\mathcal{O}(1.02220^n)$ or $\mathcal{O}(2^{n/31.58})$ time by a further case analysis in Appendix A.1. Both algorithms use an interesting and powerful relation between this problem and Exact 3-Satisfiability. We will use this relation not only to give fast exponential-time algorithms, but also to prove that, assuming the *Exponential-Time Hypothesis*, no subexponential-time algorithms for this problem can exist.

We first introduce the Partition Into Triangles and Exact 3-Satisfiability problems and survey known results in Section 4.1. Then, we give a linear-time algorithm for Partition Into Triangles on graphs of maximum degree three in Section 4.2. Thereafter, we focus on the relation between Partition Into Triangles on graphs of maximum degree four to and Exact 3-Satisfiability in Section 4.3. We use this relation to prove our hardness results in Section 4.4 and to give a simple $\mathcal{O}(1.02445^n)$-time algorithm for Partition Into Triangles in Section 4.5. In the appendix, one can find the slightly faster $\mathcal{O}(1.02220^n)$-time algorithm.

4.1. Partition Into Triangles and Exact Satisfiability

The Partition Into Triangles problem is a classical $\mathcal{NP}$-complete problem [162]. Let us first define the problem and then survey some previous results. Recall that a *triangle* in a graph is a set of three vertices that are pairwise joined by an edge.

Partition Into Triangles
Input: A graph $G = (V, E)$.
Question: Can V be partitioned into 3-element sets $S_1, S_2, \ldots, S_{|V|/3}$ such that for each S_i the graph $G[S_i]$ is a triangle?

A partitioning of the vertices of G into 3-element vertex set $S_1, S_2, \ldots, S_{|V|/3}$ that each form a triangle is also called a *triangle partition* of G.

On general graphs, Partition Into Triangles can be solved using inclusion/exclusion [33] in $\mathcal{O}(2^n n^{\mathcal{O}(1)})$ time and polynomial space (similar to Corollary 2.20). This was recently improved by Koivisto [208] who has given a general covering al-

gorithm that can be used to solve the problem in $\mathcal{O}(1.7693^n)$ time and space. Also, Björklund [26] has given a general randomised partitioning algorithm that can be used to solve the problem in $\mathcal{O}(1.496^n)$ time and polynomial space while having a probability of failure that is exponentially small in n. On bounded-degree graphs, we do not know of any results besides the hardness result of Kann: he proved that the optimisation variant (find a packing consisting of a maximum number of triangles in G) is *Max*-$\mathcal{SNP}$-complete on graphs of maximum degree at least six [198].

The second problem that we consider in this chapter is Exact 3-Satisfiability. This problem is a variant of 3-Satisfiability where a clause is satisfied if and only if exactly one literal in the clause is set to $True$.

Exact 3-Satisfiability
Input: A set of clauses C with each clause of size at most three using a set of variables X.
Question: Does there exist a truth assignment of the variables X such that each clause in C contains *exactly* one true literal?

The problem Exact Satisfiability is defined similarly by omitting the requirement on the input that clauses must have size at most three.

For both the Exact Satisfiability and the Exact 3-Satisfiability problem there exists a long series of papers giving fast exponential-time algorithms. The first non-trivial algorithm for Exact Satisfiability is due to Schroeppel and Shamir and runs in $\mathcal{O}^*(2^{\frac{n}{2}})$ time and $\mathcal{O}^*(2^{\frac{n}{4}})$ space [283]. This was already improved in 1981 by Monien et al. to $\mathcal{O}(1.1844^n)$ [240]. However, many authors seem to have missed this paper as they published algorithms with slightly worse upper bounds on the running time [94, 118]. The currently fastest algorithm for this problem is due to Byskov et al. and runs in $\mathcal{O}(1.1749^n)$ time. When the number of clauses m is used as the complexity parameter, there exists an unpublished algorithm by Skjernaa using $\mathcal{O}^*(2^m)$ time and space, and an $\mathcal{O}^*(m!)$-time and polynomial-space algorithm by Madsen [232]. These results were improved by Björklund and Husfeldt who gave an $\mathcal{O}(2^m)$-time and polynomial-space algorithm [27].

The first improvement for Exact 3-Satisfiability is an $\mathcal{O}(1.1545^n)$-time algorithm due to Drori and Peleg [118]. This was later improved by Porschen et al. [255], by Kulikov [215], and Byskov et al. [67]. The currently fastest algorithm is due to Wahlström and runs in $\mathcal{O}(1.0984^n)$ time and polynomial space [315].

4.2. A Linear-Time Algorithm on Graphs of Maximum Degree Three

We begin by considering Partition Into Triangles on graphs of maximum degree three. We will prove that this problem is polynomial time solvable on this class of graphs by giving a linear-time algorithm: Algorithm 4.1.

Lemma 4.1. *Let $G = (V, E)$ be an instance of* Partition Into Triangles *restricted to graphs of maximum degree d containing a vertex v of degree at most two. In constant time, we can either decide that G is a* No*-instance, or transform G into an equivalent smaller instance.*

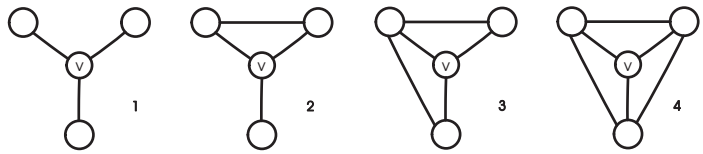


Figure 4.1. Possible edges within the neighbourhood of a vertex in a cubic graph.

Proof. If v has degree at most one, then this vertex cannot be in any triangle and the instance is a No-instance. Otherwise, let u, w be the neighbours of v. As G is of constant maximum degree, we can test in constant time whether $(u, w) \in E$. If $(u, w) \in E$, then $\{u, v, w\}$ is the unique triangle containing v, and we remove this triangle from G to obtain a smaller equivalent instance. If $(u, w) \notin E$, then v is not part of any triangle, and we again have a No-instance. □

Theorem 4.2. *Algorithm 4.1 solves* Partition Into Triangles *on graphs of maximum degree three in linear time.*

Proof. For correctness, we note that the number of vertices must be a multiple of three in order to partition G into triangles. Consider the tree cases in the if-statement in the main loop of the algorithm. Correctness of the first case follows from Lemma 4.1. For the other two cases, we observe that any local neighbourhood of v must equal one of the four cases in Figure 4.1. In Case 1, no triangle containing v exists, and, in Cases 3 and 4, the fact that G is cubic would mean that removing any triangle leads to vertices of degree at most 1 which can no longer be in a triangle. Hence, these are all No-instances. In Case 2, v can only be part of one triangle, which Algorithm 4.1 determines.

Each iteration of the main loop requires constant time, since inspecting a neighbourhood in a cubic graph can be done in constant time. In each iteration, Algorithm 4.1 either terminates, or removes three vertices from G. Hence, there are at most a linear number of iterations and Algorithm 4.1 runs in linear time. □

Algorithm 4.1. A linear-time algorithm for graphs of maximum degree three.

Input: a graph $G = (V, E)$ of maximum degree three
Output: a triangle partition T of G or **false**if no such partition exists

```
1: if |V| is not a multiple of three then return false
2: while G is non-empty do
3:    Take any vertex v ∈ V
4:    if N[v] contains a vertex of degree at most two then
5:       Reduce the graph using Lemma 4.1; if a triangle is determined, add it to T
6:    else if N[v] corresponds to Cases 1, 3, or 4 of Figure 4.1 then
7:       return false
8:    else // Case 2 of Figure 4.1
9:       Add the triangle in N[v] to T and remove its vertices from G
10: return T
```

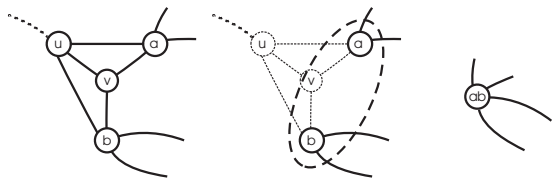


Figure 4.2. Reducing an instance with a degree three vertex by merging its neighbours.

4.3. The Relation Between Partition Into Triangles on Graphs of Maximum Degree Four and Exact 3-Satisfiability

When we restrict the Partition Into Triangles problem to graphs of maximum degree four, an interesting relation with Exact 3-Satisfiability can be observed. This relation will be the topic of this section.

We will first give three lemmas used to either decide that an instance of Partition Into Triangles on graphs of maximum degree four is a No-instance, or that it can be reduced to an equivalent smaller instance. These lemmas will apply to any instance unless all vertices in the instance have a local neighbourhood that is identical to one of two possible options. If we cannot reduce an instance in this way, pairs of vertices with one of these local neighbourhoods can be interpreted as a clause of size three in which exactly one variable must be set to $True$. The variables are then represented by connected series of vertices that each have the other remaining local neighbourhood. These variable can be set to $True$ or $False$ depending on in which of the two possible ways the corresponding connected series of vertices will be partitioned into triangles. In this way, remaining instances can be interpreted as an Exact 3-Satisfiability instance.

Lemma 4.3. *Let G be an instance of* Partition Into Triangles *of maximum degree four with a vertex v of degree at most three. In constant time, we can either decide that G is a* No*-instance, or obtain an equivalent smaller instance.*

Proof. We can assume that v has degree three: otherwise we apply Lemma 4.1.

Similar to in the proof of Theorem 4.2, the local neighbourhood of v corresponds to one of the four cases in Figure 4.1. If this neighbourhood corresponds to Case 1, then all edges incident to v are not part of any triangle. If this neighbourhood corresponds to Case 2, then the edge between v and the bottom vertex is not part of any triangle. In these two cases, we remove these edges and apply Lemma 4.1 to v, which now has degree at most two. If this neighbourhood corresponds to Case 4, then, since G is of maximum degree four, selecting any triangle in the solution results in the creation of a vertex of degree at most one: we can conclude that we have a No-instance. The same holds for Case 3 unless the vertices a and b (see Figure 4.2) are of degree four.

In this last case, we reduce the graph as in Figure 4.2. Either vertex a or vertex b must be in a triangle with u and v. If we take the triangle $\{a, u, v\}$ in a solution, then b

must be in a triangle with its other two neighbours; the same goes if we switch the roles of a and b. We distinguish three subcases depending on the number of common neighbours of a and b.

Case 1. Let a and b have no other common neighbours than u and v. Observe that an edge between a neighbour of a and a neighbour of b outside the shown part of the graph cannot be in a triangle in any solution: we such edges if any exist. Next, we merge the vertices a and b to a single vertex and remove both u and v. Now, the new vertex is part of only two different triangles, and both possibilities corresponds to taking one of the two possible triangles containing v in the original graph. Also, no extra triangles are introduced as we have removed the edge between the neighbours of the merged vertices. We conclude that the new smaller instance is equivalent.

Case 2. Let a and b have exactly three common neighbours, and let w be this third common neighbour. We must pick a triangle with u, v and either a or b. Consequently, the two edges incident to a and not incident to u or v can be removed if they are not on a common triangle together. If we do so, we obtain a vertex of degree two and can apply Lemma 4.1. The same holds for the two edges incident to b and not incident to u or v. Hence, we can assume that both a and b lie on a triangle with their third common neighbour w. Moreover, depending on which vertex from a and b we pick in a triangle with u and v, the other must be in a triangle with w. Now, we remove u and v and merge a and b to a single vertex and remove double edges. In the new instance, the edge between the merged vertex and w can be in two triangles and the choice corresponds directly to either taking the triangle u, v, a and the triangle with b and w, or taking the triangle u, v, b and the triangle with a and w.

Case 3. Let a and b have four common neighbours, called u, v, w and x. Again, the two pairs of edges incident to a and b not incident to u and v must be pairwise in triangles or we can remove them and apply Lemma 4.1. In the remaining case, each of these pairs of edges forms a triangle with the edge between w and x. Now, we must either pick the triangles u, v, a and b, w, x or we must pick u, v, b, and a, w, x. Both options involve the same vertices, hence we can remove these to obtain an equivalent smaller instance. □

As a result, we can reduce any instance of maximum degree four that is not 4-regular. In a 4-regular graph, a vertex v can have a number of possible local neighbourhoods, all shown in Figure 4.3. We will now show that we can reduce any instance having a vertex whose local neighbourhood does not correspond to one of two specific local neighbourhoods: Cases 2b and 3a from Figure 4.3.

Lemma 4.4. *Let G be a 4-regular instance of* Partition Into Triangles *containing a vertex v whose local neighbourhood is different from Cases 2b, 3a and 3b in Figure 4.3. In constant time, we can either decide that G is a* No*-instance, or we can transform G into an equivalent smaller instance.*

Proof. Consider the possible local neighbourhoods of v shown in Figure 4.3.

If the local neighbourhood of v equals Case 0, 1, 2a, or 3c, then v is incident to an edge that is not part of any triangle in G because there exists an edge incident to v from which both endpoints do not have a common neighbour. For these cases, we remove the edge and apply Lemma 4.3 to v. If this local neighbourhood equals Case 5

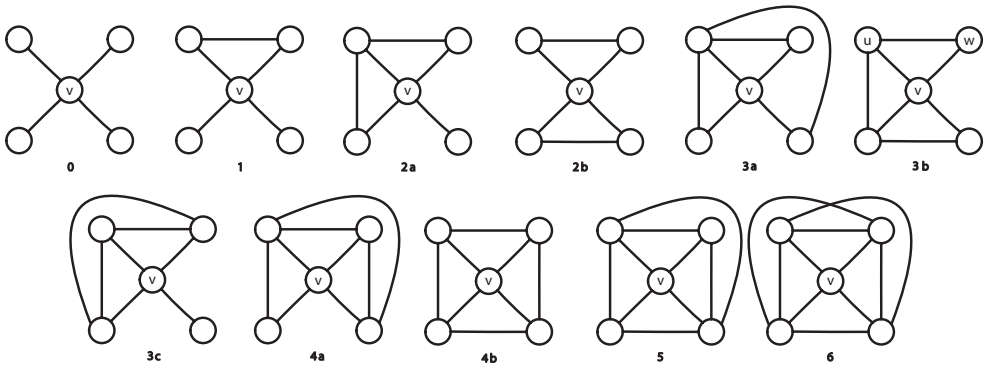


Figure 4.3. Possible edges within the local neighbourhood of a degree four vertex. The numbering corresponds to the number of edges between the neighbours of v.

or 6, then we have a No-instances since picking any triangle containing v results in a vertex of degree at most one.

Next, we consider the remaining two Cases: 4a, and 4b.

Case 4a: Consider the edge from the top left vertex to the bottom right vertex. This edge is part of two triangles, one with the centre vertex v and one with the top right vertex. If we would take any of these two triangles in the solution, a vertex of degree at most one remains. Hence, this edge cannot be part of a triangle in the solution and we can apply Lemma 4.3 after removing this edge.

Case 4b: Consider one of the four edges in $N[v]$ not incident to v, say the edge between the top two vertices. This edge is part of one or two triangles, one with v, and one with a possible third vertex outside of $N[v]$. Assume that we take the triangle with this edge and v in a solution, then the remaining two vertices will get degree two and thus they can be only in a triangle together and with a common neighbour. Hence, for each of the four edges in $N[v]$, we remove it if the endpoints of both the edge and the opposite edge (edge between the other two vertices in $N[v] \setminus \{v\}$) have no common neighbour except for v.

Note that there is no instance in which all four edges remain since each of the four corner vertices has only one neighbour outside of $N[v]$. Hence there can be at most two such common neighbours, and if there are two then they must involve the endpoints of opposite edges. We can now apply Lemma 4.3. □

Having reduced the number of possible local neighbourhoods of a vertex in an instance to three, we now remove one more such possibility.

Lemma 4.5. *Let G be a 4-regular instance of* Partition Into Triangles *in which the local neighbourhood of each vertex equals Case 2b, 3a or 3b in Figure 4.3. Then, vertices whose local neighbourhood equal Case 3b form separate connected components in G. In linear time, we can either decide that G is a* No*-instance, or remove these components to obtain an equivalent smaller instance.*

Proof. Let v be a vertex whose local neighbourhood corresponds to Case 3b of Figure 4.3. Let u be the top left vertex in this picture and consider the local neighbour-

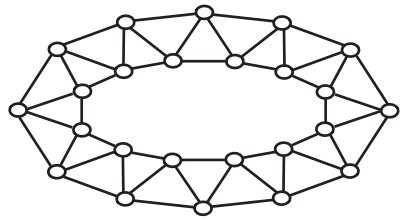


Figure 4.4. A connected component with all local neighbourhoods equal to case 3b of Figure 4.3.

hood of u. This neighbourhood cannot equal Case 2b of Figure 4.3 as it contains one vertex adjacent to two other vertices in the neighbourhood. The neighbourhood can also not equal Case 3a, since v is of degree four and thus cannot have an extra edge to the neighbour of u outside $N[v]$. We conclude that the local neighbourhood of u must equal that of Case 3b in Figure 4.3. Thus, the top two vertices have a common neighbour outside $N[v]$.

We can repeat this argument and apply it to u to conclude that the top right vertex in the picture w also has the same local neighbourhood. This shows that w and the new vertex created in the previous step must have another common neighbour. In this way, we conclude that every vertex in the connected component containing v has this local neighbourhood. An example of such a connected component can be found in Figure 4.4.

It is not hard to see that such a connected component can be partitioned into triangles if and only if its number of vertices is a multiple of three. Therefore, we can decide that we have a No-instance if this is not the case, and otherwise we can remove it in linear time to obtain an equivalent smaller instance. □

Let a *reduced instance* of Partition Into Triangles on maximum degree four graphs be an instance to which Lemmas 4.3, 4.4 and 4.5 do not apply, i.e., an instance in which each local neighbourhood corresponds to Case 2b or 3a in Figure 4.3.

Let v be a vertex in a reduced instance whose neighbourhood equals Case 3a. Note that v has one neighbour with the same neighbourhood and it has three neighbours whose neighbourhoods are equal to Case 2b. We refer to a pair of two vertices which have the neighbourhood of Case 3a as a *fan*. And, we refer to adjacent series of vertices with local neighbourhood equals Case 2b as a *cloud* of triangles. See Figure 4.5.

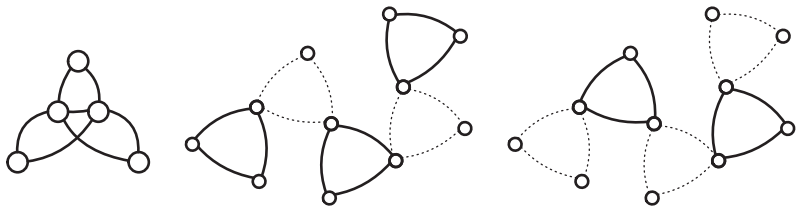


Figure 4.5. A fan and a cloud, with the two ways in which the cloud can be partitioned into triangles.

Observe how these reduced instances can be partitioned into triangles. In a fan, we must select a triangle containing the middle two vertices and exactly one of the three vertices on the boundary. In a cloud, each triangle is either selected or all its neighbouring (cloud or fan) triangles are selected. Hence, adjacent triangles will alternate between being selected and not being selected in a triangle partition of a cloud; see Figure 4.5. If such a series of adjacent triangles forms a cycle consisting of an odd number of these triangles, then the instance is a No-instance since an odd length series cannot alternate between being selected and not being selected. If a cloud does not have such an odd cycle of adjacent triangles, then it has two groups of boundary vertices connecting it to fans: in any solution all fan triangles connected to one group will be selected and all fan triangles connected to the other group will not be selected (see also Figure 4.5). The only exception to this is the single vertex cloud that directly connects two fans; here the single vertex is in both groups of endpoints.

Now, the relation between Partition Into Triangles on graphs of maximum degree four and Exact 3-Satisfiability emerges. Namely, we can interpret a reduced instance of Partition Into Triangles on graphs of maximum degree four as an Exact 3-Satisfiability instance in the following way. We interpret a fan as a clause containing three literals which corresponding variables are represented by the clouds adjacent to this fan. Exactly one fan triangle must be selected and this choice determines exactly which triangles in the adjacent clouds will be selected. In this way, we interpret a cloud as a variable that can be set to $True$ or $False$. Both truth assignments correspond to one of the two possible ways to partition the cloud into triangles. If we fix one of the two possible ways to partition a cloud into triangles and let the corresponding truth value of the corresponding variable by the value $True$, then we can define the positive and negative literals of this variable. Namely, if this partitioning of the cloud into triangles forces that a triangle from a fan is selected, then the literal corresponding to this occurrence of the variable in the clause is a positive literal. Otherwise, this occurrence of the variable in the clause is a negative literal.

Notice that if we had fixed the other possible ways to partition a cloud into triangles, then this would result in the same instance of Exact 3-Satisfiability that we would get from the above procedure only with the sign of all literals of this variable flipped. It is not hard to see that this Exact 3-Satisfiability interpretation of a reduced instance is satisfiable if and only if the partition into triangles instance has a solution.

An Exact 3-Satisfiability instance obtained in this way can have multiple identical clauses. We will now prove that if we count copies of identical clauses separately, then an instance that is obtained in this way satisfies Property 4.6. We remind the reader that $f(x)$ denotes the number of occurrences (frequency) of the variable x, and that $f_+(x)$, $f_-(x)$ denote the number of positive or negative occurrences of x, respectively.

Property 4.6. For any variable x, the number of positive $f_+(x)$ and negative $f_-(x)$ literals differ by a multiple of three.

Proposition 4.7. *An* Exact 3-Satisfiability *instance obtained in the above way from an instance of* Partition Into Triangles *satisfies Property 4.6.*

Proof. Let x be any variable in the Exact 3-Satisfiability instance. Consider the cloud that represents x, and let t_+ and t_- be the number of triangles selected this

cloud when x is set to *True* or *False*, respectively. A cloud has a fixed number of vertices and for each corresponding truth assignment each vertex is either selected in a triangle or part of a corresponding literal, thus: $3t_+ + f_+(x) = 3t_- + f_-(x)$. Hence, $f_+(x) \equiv f_-(x) \pmod 3$. □

The following lemma shows how we can model Exact 3-Satisfiability instances by reduced instances of Partition Into Triangles on graphs of maximum degree four.

Lemma 4.8. *Any variable x satisfying Property 4.6 can be represented by a cloud. Such a cloud consists of $2f(x) - 3$ vertices.*

Proof. Without loss of generality, let $f_+(x) > 0$, and define $F(x) = (f_+(x), f_-(x))$. Notice that the single vertex cloud corresponds to $F(x) = (1, 1)$, a single triangle corresponds to $F(x) = (3, 0)$, two adjacent triangles corresponds to $F(x) = (2, 2)$, and a chain of four triangles corresponds to $F(x) = (3, 3)$.

These small clouds can be extended to larger clouds that correspond to any combination $F(x) = (f_+(x), f_-(x))$ with $f_+(x) \equiv f_-(x) \pmod 3$ by repeatedly increasing $f_+(x)$ or $f_-(x)$ by three in the following way. Take three triangles that are adjacent in the sense that two triangles are connected to the third triangle through having one common vertex. Now, identify the third vertex of the middle triangle with a vertex v that could be connected to a fan in the cloud that we are enlarging. This vertex can now no longer be connected to a fan, but four new such vertices that can be connected to fans are added. Furthermore, these vertices will be in a triangle with the adjacent fan if and only if the vertex v would be in such a triangle before we enlarged the cloud. Therefore, this construction increases the number of positive or negative literals of the variable represented by the cloud by three.

One easily checks that the statement on the number of vertices holds for the initial cases and is maintained every time three triangles are added. □

We conclude by formally expressing the relation between Partition Into Triangles on graphs of maximum degree four and Exact 3-Satisfiability. The proof of the resulting theorem directly follows from the above results.

Theorem 4.9. *There exist linear-time transformations between* Partition Into Triangles *on graphs of maximum degree four and* Exact 3-Satisfiability *restricted to instances that satisfy Property 4.6 such that the following holds:*

1. *Any given instance is equivalent to its transformed instance.*

2. *An* Exact 3-Satisfiability *instance with variable set X and clause set $\mathcal{C}$ obtained from an n-vertex* Partition Into Triangles *instance of maximum degree four satisfies: $2|\mathcal{C}| + \sum_{x \in X} (2f(x) - 3) \leq n$.*

3. *A* Partition Into Triangles *instance on n vertices obtained from an* Exact 3-Satisfiability *instance satisfying Property 4.6 with variable set X and clause set $\mathcal{C}$ satisfies: $2|\mathcal{C}| + \sum_{x \in X} (2f(x) - 3) = n$.*

4.4. Hardness Results for Graphs of Maximum Degree Four

Having formalised the relation between Partition Into Triangles on graphs of maximum degree four and Exact 3-Satisfiability, we are now ready to prove some hardness results. In this section, we will show that Partition Into Triangles on graphs of maximum degree four is $\mathcal{NP}$-complete, and that no subexponential-time algorithm for this problem exists unless the *Exponential-Time Hypothesis* (ETH) fails.

Theorem 4.10. Partition Into Triangles *on graphs of maximum degree four is* $\mathcal{NP}$*-complete.*

Proof. Clearly, the problem is in $\mathcal{NP}$. For hardness, we reduce from the $\mathcal{NP}$-complete problem Exact 3-Satisfiability [162]. Given an Exact 3-Satisfiability instance, we enforce Property 4.6 by making three copies of each clause. Then, the result follows from Theorem 4.9. □

Next, we show that no subexponential-time algorithm for our problem exists. We note that although we prove that, under the ETH, no algorithm subexponential in n exists, this also implies that no algorithm subexponential in m exists as $m = \mathcal{O}(n)$ on bounded-degree graphs.

Theorem 4.11. *Assuming the ETH, there exists no algorithm for* Partition Into Triangles *on graphs of maximum degree four with a running time subexponential in* n*.*

Proof. Consider an arbitrary 3-Satisfiability instance with m clauses. We create an equivalent Exact 3-Satisfiability instance with $4m$ clauses by using the equivalence from [275] shown below. To avoid confusion, we now denote a 3-Satisfiability clause with literals x, y, and z by $\mathrm{SAT}(x, y, z)$ and a similar Exact 3-Satisfiability clause with literals x, y, and z by $\mathrm{XSAT}(x, y, z)$.

$$\begin{aligned}\mathrm{SAT}(x, y, z) \iff\ & \mathrm{XSAT}(x, v_1, v_2) \wedge \mathrm{XSAT}(y, v_2, v_3) \\ & \wedge \mathrm{XSAT}(v_1, v_3, v_4) \wedge \mathrm{XSAT}(\neg z, v_2, v_5)\end{aligned}$$

We then transform this Exact 3-Satisfiability instance into an equivalent instance of Partition Into Triangles of maximum degree four using the construction in the proof of Theorem 4.10. This construction triples the number of clauses to $12m$, and thus the total sum of the number of literal occurrences is at most $36m$. By Lemma 4.8, variables x can be represented by clouds using less than $2f(x)$ vertices each. This gives at most $96m$ vertices: $72m$ for the variables and another $24m$ for the two vertices of a fan for each clause.

Suppose there exists a subexponential-time algorithm for Partition Into Triangles on graphs of maximum degree four, i.e, an $\mathcal{O}(2^{\delta n})$-time algorithm for all $\delta > 0$. Then, this algorithm solves 3-Satisfiability in $\mathcal{O}(2^{\epsilon m})$ for all $\epsilon > 0$ using the above construction and $\delta = \epsilon/96$. However, assuming the ETH, no such algorithm can exist by the sparsification lemma [189] (Lemma 3.12). □

4.5. A Very Fast Exponential-Time Algorithm

In the previous section, we have given two hardness results for Partition Into Triangles on graphs of maximum degree four. Despite these results, this problems seems to admit very fast, though exponential-time, algorithms.

In this section, we give a simple $\mathcal{O}(1.02445^n)$-time algorithm for this problem based on the algorithm for Exact Satisfiability by Byskov et al. [67] and the algorithm for Exact 3-Satisfiability by Wahlström [315]. In Appendix A.1, we also give a faster $\mathcal{O}(1.02220^n)$-time algorithm. This algorithm is based on the same principles as the one in Theorem 4.12 but uses an extensive case analysis.

Theorem 4.12. *There exists an $\mathcal{O}(1.02445^n)$-time algorithm for* Partition Into Triangles *on graphs of maximum degree four.*

Proof. Use Theorem 4.9 to obtain an instance of Exact 3-Satisfiability with variable set X and clause set $\mathcal{C}$ satisfying $n \geq 2|\mathcal{C}| + \sum_{x \in X} (2f(x) - 3)$. Let γ_1 be the number of variables x with $f_+(x) = f_-(x) = 1$ and let γ_3 be the number of variables x with $f(x) \geq 3$; by Property 4.6 the total number of variables γ equals $\gamma_1 + \gamma_3$. Since clauses have size three, we find that $n \geq 2(2\gamma_1 + 3\gamma_3)/3 + \gamma_1 + 3\gamma_3 = 2\frac{1}{3}\gamma_1 + 5\gamma_3$.

If $\gamma_1 \leq 0.10746n$, then apply Wahlström's $\mathcal{O}(1.0984^\gamma)$-time algorithm for Exact 3-Satisfiability [315]. Now, $\gamma = \gamma_1 + \gamma_3 \leq 0.10746n + (n - 2\frac{1}{3} \times 0.10746n)/5 < 0.25732n$ by basic calculus. Therefore, the problem is solved by this algorithm in $\mathcal{O}(1.0984^{0.25732n}) = \mathcal{O}(1.02445^n)$ time.

Otherwise $\gamma_1 > 0.10746n$. In this case, we first reduce the instance in polynomial time removing all variables x with $f_+(x) = f_-(x) = 1$ by using the following equivalence where C and C' are arbitrary sets of literals and Φ denotes any Exact Satisfiability formula:

$$(x, C) \wedge (\neg x, C') \wedge \Phi \iff (C, C') \wedge \Phi$$

Hereafter, we apply the $\mathcal{O}(2^{0.2325\gamma})$ Exact Satisfiability algorithm from Byskov et al. [67]. This algorithm now solves our instance in $\mathcal{O}(1.1749^{\gamma_3}) = \mathcal{O}(1.02445^n)$ time as $\gamma_3 \leq (n - 2\frac{1}{3} \times 0.10746n)/5 < 0.14986n$ by basic calculus. □

Theorem 4.13. *There exists an $\mathcal{O}(1.02220^n)$-time algorithm for* Partition Into Triangles *on graphs of maximum degree four.*

Proof. See Appendix A.1. □

4.6. Concluding Remarks

In this chapter, we have given some results on Partition Into Triangles restricted to bounded-degree graphs. We have given a linear-time algorithm on graphs of maximum degree three. Our main result was an $\mathcal{O}(1.02220^n)$-time algorithm on graphs of maximum degree four. This algorithm uses an interesting relation between Partition Into Triangles on graphs of maximum degree four and Exact 3-Satisfiability; a relation that we also used to give a number of hardness results on the studied problem.

Our algorithm is an example of a 'very fast' exponential-time algorithm. This in the sense that it has a singly-exponential running time with a very small base of the exponent when compared to exact exponential-time algorithms in the literature. The algorithm can be used to solve the problem in practice, even on reasonably large instances. This is because the running time of $\mathcal{O}(1.02220^n)$ does not involve any large constant, or even polynomial, factors hidden in the big-$\mathcal{O}$ notation.

It would be interesting to see whether there exist more natural[1] problems for which similar (possibly even faster) 'very fast' exact exponential-time algorithms exist.

[1] See the discussion in the introduction of this chapter.

II

Branching Algorithms and Measure-Based Analyses

5 Designing Algorithms for Dominating Set

An important paradigm for the design of exact exponential-time algorithms is *branch and reduce*, first used in 1960 for solving Satisfiability problems by Davis and Putnam [101, 102]. A recent breakthrough in the analysis of branch-and-reduce algorithms is *measure and conquer* [144]. The measure-and-conquer approach helps to obtain good upper bounds on the running times of branch-and-reduce algorithms, often improving upon the currently best-known bounds for exact exponential-time algorithms. It has been used successfully on many problems and has become one of the standard techniques in the field.

In this chapter, we will use the measure-and-conquer approach in a step-by-step process to design exact exponential-time algorithms for the Dominating Set problem. We will do so in the following way. Measure and conquer uses a non-standard size measure for instances. This measure is based on weight vectors that are computed by solving a numerical optimisation problem. Analysis of the solution of this numerical optimisation problem not only yields an upper bound on the running time of the algorithm, but also gives information on which instances are worst-case instances with respect to the analysis. This information can then be used to design new reduction rules. We add these new reduction rules to the branch-and-reduce algorithm, possibly improving the algorithm. In the next step of our step-by-step design process, we repeat this procedure by analysing and improving the modified algorithm.

We apply this step-by-step process to Set Cover instances that are equivalent

[†]This chapter is joint work with Hans L. Bodlaender. This research started in my master's thesis [299] supervised by Hans L. Bodlaender, INF/SCR-2006-005. Several results have been added and corrected since. The chapter contains results of which a preliminary version has been presented at the 25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008) [304]. The full version is also available as technical report UU-CS-2009-025 [304].

to instances of the Dominating Set problem. This is done in exactly the same way as the setting in which measure and conquer was first introduced [141]. If we start with the trivial algorithm, then we can obtain the $\mathcal{O}(1.5263^n)$-time algorithm that Fomin et al. used to introduce measure and conquer after a number of steps [141, 144]. After additional steps, we obtain a faster algorithm: an $\mathcal{O}(1.4969^n)$-time algorithm for Dominating Set. For this algorithm, we show that it cannot be improved significantly by performing additional improvement steps in the same way as used to obtain this algorithm.

We first repeat some basic definitions on Dominating Set and survey known results in Section 5.1. In Section 5.2, we give a short introduction to the measure-and-conquer technique. Then, we give a detailed step-by-step overview of how we design an algorithm for Dominating Set using measure and conquer in Section 5.3. In this section, we also prove that there exists an $\mathcal{O}(1.4969^n)$-time algorithm for Dominating Set, although the case analysis involved is moved to Appendix A.2. We also give some intuition that it is hard to improve our algorithm significantly using the same approach in Section 5.4. In Section 5.5, we show that the same algorithm can be used to solve some other problems as well, both in the field of exact exponential-time algorithms and the field of parameterised algorithms. We conclude in Section 5.6 by giving some remarks on how to solve the associated numerical optimisation problems.

5.1. Dominating Set

Let us first recall the definition of Dominating Set and then survey some previous results. A subset $D \subseteq V$ of the vertices of a graph G is called a *dominating set* if every vertex $v \in V$ is either in D or adjacent to some vertex in D, i.e., a dominating set D is a set of vertices in G such that $\bigcup_{v \in D} N[v] = V$. A dominating set in G is called a *minimum dominating set* if it is of minimum cardinality.

Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set $D \subseteq V$ in G of size at most k?

The Dominating Set problem is a classical $\mathcal{NP}$-complete problem [162]. Moreover, there exists no subexponential-time algorithm for this problem unless the Exponential-Time Hypothesis fails. A proof of this can be found in [151], and it is also a direct consequence of Proposition 5.3.

While for several classical combinatorial problems the first non-trivial exact algorithms date from many years ago, the first algorithms with running time faster than $\mathcal{O}^*(2^n)$ for Dominating Set are from 2004. In this year, there were three independent papers giving faster algorithms for the problem: one by Fomin et al. [151], one by Randerath and Schiermeyer [279], and one by Grandoni [172]. Before our work, the fastest algorithms for Dominating Set were by Fomin, Grandoni, and Kratsch [144]: they gave two algorithms, one using $\mathcal{O}(1.5260^n)$ time and polynomial space, and one using $\mathcal{O}(1.5137^n)$ time and exponential space. See Table 5.1 for an overview of recent results on this problem.

Authors		Polynomial space	Exponential space
Fomin, Kratsch, Woeginger	[151]	$\mathcal{O}(1.9379^n)$	
Randerath and Schiermeyer	[279]	$\mathcal{O}(1.8899^n)$	
Grandoni	[172]	$\mathcal{O}(1.9053^n)$	$\mathcal{O}(1.8021^n)$
Fomin, Grandoni, Kratsch	[142]	$\mathcal{O}(1.5263^n)$	$\mathcal{O}(1.5137^n)$
van Rooij	[299]	$\mathcal{O}(1.5134^n)$	$\mathcal{O}(1.5086^n)$
van Rooij, Bodlaender	[302]	$\mathcal{O}(1.5134^n)$	$\mathcal{O}(1.5063^n)$
van Rooij, Nederlof, van Dijk	[307]†		$\mathcal{O}(1.5048^n)$
This chapter		$\mathcal{O}(1.4969^n)$	$\Longleftarrow$

† See also Chapter 8.

Table 5.1. Known exact exponential-time algorithms for Dominating Set.

5.1.1. Set Cover and Dominating Set

We will often use the following Set Cover modelling of Dominating Set. This modelling is useful in the design of algorithms with running times faster than $\mathcal{O}^*(2^n)$ for Dominating Set, a fact that was first observed by Grandoni [172]. This modelling is the basis of most exact exponential-time algorithms for Dominating Set (all except the first two in the overview of Table 5.1) and will be used often throughout this thesis.

Recall that, given a multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$, a *set cover* $\mathcal{C}$ of $\mathcal{S}$ is a subset $\mathcal{C} \subseteq \mathcal{S}$ such that every element in any of the sets in $\mathcal{S}$ occurs in some set in $\mathcal{C}$. I.e., a set cover $\mathcal{C}$ of $\mathcal{S}$ is a collection of sets such that $\bigcup_{S\in\mathcal{C}} S = \bigcup_{S\in\mathcal{S}} S = \mathcal{U}$. A set cover $\mathcal{C}$ is called a *minimum set cover* if it is of minimum cardinality.

Set Cover
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}(\mathcal{S})$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a set cover $\mathcal{C} \subseteq \mathcal{S}$ of size at most k?

Also, recall that the frequency of an element e is denoted by $f(e)$, and the set of all elements in $\mathcal{S}$ that contain e by $\mathcal{S}(e) = \{S \in \mathcal{S} \mid e \in S\}$, thus, $f(e) = |\mathcal{S}(e)|$.

We can reduce Dominating Set to Set Cover by introducing a set for each vertex of G containing the closed neighbourhood of this vertex, i.e., $\mathcal{S} := \{N[v] \mid v \in V\}$, $\mathcal{U} := V$. Hence, we can solve an instance of Dominating Set on an n-vertex graph by a using a set-cover algorithm running on an instance with n sets and a universe of size n.

5.2. Measure and Conquer

In the design of exact exponential-time algorithms, the *branch-and-reduce* paradigm is one of the most prominently used approaches; see Section 2.1. A breakthrough in the analysis of branch-and-reduce algorithms is the *measure-and-conquer* technique by Fomin, Grandoni, and Kratsch [144]. This follows earlier work by Eppstein on analysing branch-and-reduce algorithms by multivariate recurrences [124]. In a measure-and-conquer analysis, a carefully chosen non-standard measure of instance or subproblem size is used. This stands in contrast to classic analyses relying on simple, mostly integer measures for the size of an instance, e.g., the number of vertices in a graph.

The first problem to which the measure-and-conquer technique has been applied is Dominating Set [141, 144]. In these papers, the authors present an algorithm for Set Cover. This algorithm is then applied to instances obtained by transforming an n-vertex Dominating Set instance into an equivalent Set Cover instance on n sets over a universe of size n. The algorithm is analysed using the following measure k, where v and w are weight functions giving an element e of frequency $f(e)$ measure $v(f(e))$ and a set S of size $|S|$ measure $w(|S|)$.

$$k := k(\mathcal{S}, \mathcal{U}) = \sum_{e \in \mathcal{U}} v(f(e)) + \sum_{S \in \mathcal{S}} w(|S|) \qquad \text{with: } v, w : \mathbb{N} \to \mathbb{R}_+$$

The behaviour of the algorithm is analysed using this measure. For each branching rule, a series of recurrence relations is formulated corresponding to all possible situations the branching rule can be applied to.

Let $N(k)$ be the number of subproblems generated by a branch-and-reduce algorithm applied to an instance of measure k. Furthermore, let R be a set of cases considered by the branch-and-reduce algorithm covering all possible situations in which the algorithm can branch. For any such case $r \in R$, we denote by $\#(r)$ the number of subproblems generated when branching in case r, and by $\Delta k_{(r,i)}$ the decrease of the measure k for the i-th subproblem after branching in case r. Similar to the analysis in Section 2.1, we thus obtain a series of recurrence relations of the form:

$$\forall r \in R \quad : \quad N(k) \leq \sum_{i=1}^{\#(r)} N\big(k - \Delta k_{(r,i)}\big)$$

We will often identify R with the set of corresponding recurrences.

A solution to this set of recurrence relations has the form α^k, for some $\alpha > 1$. This gives an upper bound on the running time of the branch-and-reduce algorithm expressed in the measure k. Assume that $v_{\max} = \max_{n \in \mathbb{N}} v(n)$, $w_{\max} = \max_{n \in \mathbb{N}} w(n)$ are finite numbers. Then, $\alpha^{(v_{\max}+w_{\max})n}$ is an upper bound on the running time of the algorithm for Dominating Set since, for every input instance $(\mathcal{S}, \mathcal{U})$, we have that $k(\mathcal{S}, \mathcal{U}) \leq (v_{\max} + w_{\max})n$ and thus $\alpha^{k(\mathcal{S},\mathcal{U})} \leq \alpha^{(v_{\max}+w_{\max})n}$.

What remains is to choose ideal weight functions: weight functions that minimise the proven upper bound on the running time of the algorithm, i.e., weight functions that minimise $\alpha^{v_{\max}+w_{\max}}$. This gives rise to a large numerical optimisation problem. Under some assumptions on the weight functions (see Sections 5.3 and 5.6 for more details), this gives a large but finite quasiconvex optimisation problem. Such a quasiconvex problem can be solved by computer; see [124]. In our case, the numerical optimisation problem is not always a quasiconvex program. Details on how we solved the associated numerical optimisation problems can be found in Section 5.6.

In this way, Fomin, Grandoni and Kratsch [144] prove a running time of $\mathcal{O}(1.5263^n)$ on an algorithm that is almost identical to the $\mathcal{O}(1.9053^n)$-time algorithm in [172]. See Section 5.3 for concrete examples of measure-and-conquer analyses on simple algorithms for the Set Cover problem.

Although relatively new, the measure-and-conquer technique has become one of the standard approaches in the field. Examples of results using this approach include Dominating Set [144], Independent Set [59, 144] (Chapter 7), Dominating Clique [54, 214], Connected Dominating Set [133, 143] (although these results

New reduction rule	Section	Running Time
Trivial algorithm	5.3.1	$\mathcal{O}^*(2^n)$
Unique element rule	5.3.2	$\mathcal{O}(1.7311^n)$
Stop when all sets have cardinality one	5.3.3	$\mathcal{O}(1.6411^n)$
Subset rule	5.3.4	$\mathcal{O}(1.5709^n)$
Stop when all sets have cardinality two	5.3.5	$\mathcal{O}(1.5169^n)$
Subsumption rule	5.3.6	$\mathcal{O}(1.5134^n)$
Counting rule	5.3.7	$\mathcal{O}(1.5055^n)$
Size two set with only frequency-two elements rule	5.3.8	$\mathcal{O}(1.4969^n)$

Table 5.2. The iterative improvement of the algorithm.

were improved without using measure and conquer in [1]), Independent Dominating Set [56, 166], Edge Dominating Set [303] (Chapter 6), Feedback Vertex Set [136], bounding the number of minimal dominating sets [146], and many others. Also, a direct application in the field of parameterised algorithms has been found [23]. Measure and conquer is one of the techniques that we will often use in Chapters 5-10.

5.3. Designing Algorithms Using Measure and Conquer

In this section, we use the measure-and-conquer technique not only to analyse branch-and-reduce algorithms, but also to serve as a guiding tool to design these algorithms. This works in the following way.

Suppose we are given a non-standard measure of instance size using weight functions and some initial branching procedure, i.e. we are given the basic ingredients of a measure-and-conquer analysis and some trivial algorithm. Then, in an iterative process, we will formulate a series of branch-and-reduce algorithms. In this series, each algorithm is analysed using measure and conquer. Hereafter, the associated numerical optimisation problem is inspected, and the recurrence relations that bound the current optimum (the best upper bound on the running time involving this measure) are identified. Each of these bounding recurrence relations corresponds to one or more worst-case instances which can be identified easily. We can use these instances to design new reduction rules, or change the branching procedure, such that some of these worst-case instances are handled more efficiently by the algorithm. The modification then gives a new, faster algorithm. This improvement series ends when we have sufficient evidence showing that improving the current worst cases is hard.

In the next subsection, we set up the framework used to analyse our algorithms by measure and conquer and analyse a trivial algorithm. In each subsequent subsection, we treat the next algorithm from the series: we analyse its running time and apply an improvement step, as just described. See Table 5.2 for an overview of the algorithms in the series and the corresponding upper bounds on the running times.

5.3.1. A Trivial Algorithm

We start with a trivial algorithm for Set Cover: Algorithm 5.1. Given a problem instance $(\mathcal{S}, \mathcal{U})$, this algorithm simply selects a largest set S from $\mathcal{S}$ and considers two

Algorithm 5.1. A trivial set-cover algorithm.

Input: a set cover instance $(\mathcal{S}, \mathcal{U})$
Output: a minimum set cover of $(\mathcal{S}, \mathcal{U})$, or **false** if non exists
MSC$(\mathcal{S}, \mathcal{U})$:
1: **if** $\mathcal{S} = \emptyset$ **then return** $\emptyset$ if $\mathcal{U} = \emptyset$, or **false** otherwise
2: Let $S \in \mathcal{S}$ be a set of maximum cardinality among the sets in $\mathcal{S}$
3: Let $\mathcal{C}_1 = \{S\} \cup \text{MSC}(\{S' \setminus S \mid S' \in \mathcal{S} \setminus \{S\}\}, \mathcal{U} \setminus S)$ and $\mathcal{C}_2 = \text{MSC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$
4: **return** the smallest set cover from $\mathcal{C}_1$ and $\mathcal{C}_2$, or **false** if no set cover is found

subproblems (branches): one in which it takes S in the set cover and one in which it discards S. In the branch where S is taken in the set cover, we remove S from $\mathcal{S}$, and we remove all elements in S from $\mathcal{U}$; consequently, we also remove all elements in S from all sets $S' \in \mathcal{S} \setminus \{S\}$. In the other branch, we just remove S from $\mathcal{S}$. Then, the algorithm recursively solves both generated subproblems and returns the smallest set cover returned by the recursive calls. It stops when there is no set left to branch on; then, it checks whether the generated subproblem corresponds to a set cover.

We analyse this algorithm using measure and conquer. To this end, let $v, w : \mathbb{N} \to \mathbb{R}_+$ be weight functions giving an element e of frequency $f(e)$ measure $v(f(e))$ and a set S of size $|S|$ measure $w(|S|)$, just like in Section 5.2. Furthermore, let $k := \sum_{e \in \mathcal{U}} v(f(e)) + \sum_{S \in \mathcal{S}} w(|S|)$ be our measure.

We start by defining the following very useful quantities:

$$\Delta v(i) = v(i) - v(i-1) \qquad \Delta w(i) = w(i) - w(i-1) \qquad \text{for } i \geq 1$$

We impose some constraints on the weights. We require the weights to be *monotone* and set the weights of sets and elements that no longer play a role in the algorithm to zero:

$$v(0) = 0 \quad w(0) = 0 \qquad \forall i \geq 1 \quad : \quad \Delta v(i) \geq 0 \qquad \Delta w(i) \geq 0$$

Intuitively, this corresponds to the idea that larger sets and higher-frequency elements contribute more to the complexity of the problem than smaller sets and lower-frequency elements, respectively. Furthermore, we impose the following non-restricting *steepness* inequalities, which we will discuss in a moment:

$$\forall i \geq 2 \quad : \quad \Delta w(i-1) \geq \Delta w(i)$$

Let r_i be the number of elements of frequency i in S. In the branch where S is taken in the set cover, the measure is reduced by $w(|S|)$ because we remove S, by $\sum_{i=1}^{\infty} r_i v(i)$ because we remove its elements, and by at least an additional $\min_{j \leq |S|}\{\Delta w(j)\} \cdot \sum_{i=1}^{\infty} r_i(i-1)$ because the removal of these elements reduces other sets in size. In the other branch, the measure is reduced by $w(|S|)$ because we remove S, and by an additional $\sum_{i=1}^{\infty} r_i \Delta v(i)$ because the frequencies of elements in S are reduced by one.

Let Δk_{take} and $\Delta k_{\text{discard}}$ be the decrease in measure k in the branch where we *take* S in the solution and where we *discard* S, respectively. Thus, we have derived

the following lower bounds on the decrease of the measure:

$$\begin{aligned}\Delta k_{\text{take}} &\geq w(|S|) + \sum_{i=1}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=1}^{\infty} r_i (i-1) \\ \Delta k_{\text{discard}} &\geq w(|S|) + \sum_{i=1}^{\infty} r_i \Delta v(i)\end{aligned}$$

Here, we used the steepness inequalities to replace the term $\min_{j \leq |S|}\{\Delta w(j)\}$ by $\Delta w(|S|)$. One can show that these steepness inequalities do not change the solution to the associated numerical optimisation problem; they only simplify its formulation. In other words, the steepness inequalities do not influence the computed upper bound on the running time; they only simplify its computation.

In this way, we find the following set of recurrence relations. Let $N(k)$ be the number of subproblems generated by branching on an instance of measure k.

$$\forall\, |S| \geq 1, \forall\, r_i : \sum_{i=1}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

Finally, we compute the optimal weight functions minimising $\alpha^{v_{\max}+w_{\max}}$, where α^k is a solution to the above set of recurrence relations. To make this set of recurrence relations finite, we first set $v(i) = v_{\max}$ and $w(i) = w_{\max}$ for all $i \geq p$ for some $p \in \mathbb{N}$. This results in the fact that all recurrence relations with $|S| > p + 1$ are dominated by those with $|S| = p + 1$ as $\Delta w(i), \Delta v(i) = 0$ if $i \geq p + 1$. Moreover, we now need to consider only recurrence relations with $|S| = \sum_{i=1}^{p} r_i + r_{>p}$ where $r_{>p} = \sum_{i=p+1}^{\infty} r_i$ and $r_{>p}$ has the role of r_{p+1} in the above formulas. In this chapter, we use $p = 8$, but any $p > 8$ leads to the same results. The choice for p needs to be large enough, such that the recurrence relations using large sets S or large elements e (with generally $\Delta w(|S|) = 0$ and $\Delta v(f(e)) = 0$) are not among the tight cases. In this case, the fact that we make the problem finite does not affect the running time in a negative way.

Notice that if all weights $v(i)$, $w(i)$ are multiplied by a positive real number, then a different value α will result from the set of recurrence relations. However, it is not hard to see that, in this case, the value $\alpha^{v_{\max}+w_{\max}}$ will remain the same. Hence, we can set $w_{\max} = 1$ without loss of generality[1]. We will omit these details concerning finiteness of the numerical optimisation problem in the analyses of the other algorithms throughout the improvement series in the coming subsections.

We solve the corresponding numerical optimisation problem with continuous variables $v(1)$, $v(2), \ldots, v(p)$, $w(1), w(2), \ldots, w(p)$ minimising $\alpha^{v_{\max}+w_{\max}}$ where α^k is a solution to the set of recurrence relations. We do so with a C++ implementation of an algorithm that we designed for this type of numerical optimisation problems. This algorithm is a modification of Eppstein's smooth quasiconvex programming algorithm [124] that we describe in Section 5.6. For an alternative way to solve such numerical problems, see the work of Gaspers and Sorkin [167].

[1]We could equally well have set $v_{\max} = 1$ giving the same upper bound on the running time for all algorithms in the improvement series except for this first algorithm. This is the case since $v_{\max} = 0$ in this analysis, therefore, the optimal weights cannot be multiplied by a positive real number such that $v_{\max} = 1$.

Using our implementation (see Section 5.6), we obtain a solution of $N(k) \leq 2^k$ using the following weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
$w(i)$	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

This gives an upper bound on the running time of Algorithm 5.1 of $\mathcal{O}^*\big(2^{(0+1)n}\big) = \mathcal{O}^*(2^n)$. All recurrences considered contribute to the *bounding cases* (worst cases) in this analysis.

It is no surprise that we obtain a running time of $\mathcal{O}^*(2^n)$ for this trivial algorithm: the algorithm branches on all n sets considering two subproblems. We only formally used measure and conquer here as the optimal weights correspond exactly to the standard measure: the total number of sets in the instance. The above analysis functions to set up our algorithm design process.

5.3.2. The First Improvement Step: Unique Elements

We will now give the first improvement step. To this end, we consider the bounding cases of the numerical optimisation problem associated with the previous analysis. In this case, all recurrences in the numerical optimisation problem form the set of bounding cases. At each improvement step in the design process of our algorithm, we will consider the '*smallest*' worst case. With smallest worst case, we mean the worst case that involves the smallest sets and lowest frequency elements. We note that this is often not uniquely defined, and that we mean the worst case that intuitively looks the smallest. This 'smallest' worst case from the analysis of Algorithm 5.1 is the case where $|S| = r_1 = 1$.

This case can be improved easily. Algorithm 5.1 considers many subsets of the input multiset $\mathcal{S}$ that will never result in a set cover since they cannot cover all elements in the universe $\mathcal{U}$. More specifically, when considering a set with unique elements (elements of frequency one), Algorithm 5.1 still branches on this set. This, however, is not necessary since any set cover includes this set.

In the second algorithm in the series, we add a reduction rule dealing with unique elements improving, among others, the case $|S| = r_1 = 1$. This reduction rule takes any set containing a unique element in the computed set cover.

Reduction Rule 5.1.

if there exists an element $e \in \mathcal{U}$ of frequency one **then**
  return $\{R\} \cup \mathrm{MSC}(\{R' \setminus R \mid R' \in \mathcal{S} \setminus \{R\}\}, \mathcal{U} \setminus R)$, where R is the set with $e \in R$

We can change the formulation of the algorithm after adding this reduction rule to it: we no longer need to check whether every computed cover also covers all of $\mathcal{U}$. In this way, we obtain Algorithm 5.2.

Let us analyse Algorithm 5.2. To this end, we use the same measure as before; only we add some extra constraints. First of all, we note that the new reduction rule does not increase the measure when applied to an instance: this is easy to see as Reduction Rule 5.1 removes sets and elements from the instance without adding any. Secondly, we note that the new reduction rule can be applied exhaustively in polynomial time.

Algorithm 5.2. An improved version of Algorithm 5.1 by adding Reduction Rule 5.1.

Input: a set cover instance $(\mathcal{S}, \mathcal{U})$
Output: a minimum set cover of $(\mathcal{S}, \mathcal{U})$
MSC$(\mathcal{S}, \mathcal{U})$:
1: **if** $\mathcal{S} = \emptyset$ **then return** $\emptyset$
2: Let $S \in \mathcal{S}$ be a set of maximum cardinality among the sets in $\mathcal{S}$
3: **if** there exists an element $e \in \mathcal{U}$ of frequency one **then**
4: **return** $\{R\} \cup \text{MSC}(\{R' \setminus R \mid R' \in \mathcal{S} \setminus \{R\}\}, \mathcal{U} \setminus R)$, where R is such that $e \in R$
5: Let $\mathcal{C}_1 = \{S\} \cup \text{MSC}(\{S' \setminus S \mid S' \in \mathcal{S} \setminus \{S\}\}, \mathcal{U} \setminus S)$ and $\mathcal{C}_2 = \text{MSC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$
6: **return** the smallest set cover from $\mathcal{C}_1$ and $\mathcal{C}_2$

These two properties not only hold for this reduction rule, but also for the reduction rules that we will add later. In these future cases, we will omit these observations from the analysis.

Since unique elements are directly removed from an instance, they do not contribute to the (exponential) complexity of a problem instance; therefore, we can set $v(1) = 0$. Notice that this results in $\Delta v(2) = v(2)$.

Next, we derive new recurrence relations for Algorithm 5.2. Let Algorithm 5.2 branch on a set S containing r_i elements of frequency i. Due to Reduction Rule 5.1, we now have to consider only cases with $|S| = \sum_{i=2}^{\infty} r_i$, i.e., with $r_1 = 0$. In the branch where Algorithm 5.2 takes S in the set cover, nothing changes to the decrease in measure. But, in the branch where it discards S, an additional decrease in measure is obtained when S contains elements of frequency two, that is, when S contains elements that become unique elements after discarding S. In this case where $r_2 > 0$, at least one extra set is taken in the set cover. Because of the steepness inequalities $(\Delta w(i-1) \geq \Delta w(i))$, the smallest decrease in measure occurs when the one set consists exactly of all frequency-two elements in S: this one set will have less measure than multiple smaller sets containing these same elements, and less measure than a larger set containing additional elements. Since the size of this set is r_2, this gives us an additional decrease in measure of $[r_2 > 0]w(r_2)$.

Altogether, this gives the following set of recurrences:

$$\forall\, |S| \geq 1, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\begin{aligned} \Delta k_{\text{take}} &\geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1) \\ \Delta k_{\text{discard}} &\geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + [r_2 > 0] w(r_2) \end{aligned}$$

We solve the associated numerical optimisation problem minimising $\alpha^{v_{\max}+w_{\max}}$ where α^k is a solution to the set of recurrence relations. We obtain a solution of $N(k) \leq 1.58143^k$ using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.011524	0.162465	0.192542	0.197195	0.197195	0.197195	0.197195
$w(i)$	0.750412	0.908050	0.968115	0.992112	1.000000	1.000000	1.000000	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.58143^{(0.197195+1)n}\big) = \mathcal{O}(1.73101^n)$ on the running time of Algorithm 5.2. The bounding cases of the numerical optimisation problem are:

$|S| = r_2 = 1$ $|S| = r_3 = 2$ $|S| = r_4 = 2$ $|S| = r_4 = 3$ $|S| = r_5 = 4$ $|S| = r_5 = 5$
$|S| = r_6 = 5$ $|S| = r_6 = 6$ $|S| = r_{>6} = 6$

This concludes the first improvement step. By applying this improvement step, we have obtained our first simple algorithm with a non-trivial upper bound on the running time.

5.3.3. Improvement Step Two: Sets of Cardinality One

The next step will be to inspect the case $|S| = r_2 = 1$ more closely. This case corresponds to branching on a set S containing only one element and this elements is of frequency two.

We improve this case by the simple observation that we can stop branching when all sets have size one. Namely, in this case, any solution consists of a series of singleton sets: one for each remaining element. This leads to the following reduction rule.

Reduction Rule 5.2.

Let $S \in \mathcal{S}$ be a set of maximum cardinality
if $|S| \leq 1$ **then**
  return $\{\{e\} \mid e \in \mathcal{U}\}$

We add Reduction Rule 5.2 to Algorithm 5.2 to obtain our next algorithm. For this algorithm, we derive the following set of recurrence relations; these are exactly the ones used to analyse Algorithm 5.2 that correspond to branching on a set of size at least two:

$$\forall\, |S| \geq 2, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\Delta k_{\text{take}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1)$$

$$\Delta k_{\text{discard}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + [r_2 > 0] w(r_2)$$

We solve the associated numerical optimisation problem and obtain a solution of $N(k) \leq 1.42604^k$ on the recurrence relations using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.161668	0.325452	0.387900	0.395390	0.395408	0.395408	0.395408
$w(i)$	0.407320	0.814639	0.931101	0.981843	0.998998	1.000000	1.000000	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.42604^{(0.395408+1)n}\big) = \mathcal{O}(1.64107^n)$ on the running time of the algorithm. The bounding cases of the numerical optimisation problem are:

$|S| = r_2 = 2$ $|S| = r_3 = 2$ $|S| = r_3 = 3$ $|S| = r_4 = 3$ $|S| = r_5 = 3$ $|S| = r_5 = 4$
$|S| = r_5 = 5$ $|S| = r_6 = 5$ $|S| = r_6 = 6$

5.3.4. Improvement Step Three: Subsets

Consider the new 'smallest' worst case: $|S| = r_2 = 2$. For this case, the previous section uses the following inequalities in the corresponding recurrences: $\Delta k_{\text{take}} \geq w(2) + 2v(2) + 2\Delta w(2)$ and $\Delta k_{\text{discard}} \geq w(2) + 2\Delta v(2) + w(2)$. An instance that is tight for these inequalities contains a set S of size two containing two frequency-two elements, and, in this instance, these frequency-two elements occur only in the set S and in a set R that is a copy of S. To see this, notice that the last computed weights satisfy $2\Delta w(2) = w(2)$ and that by definition $v(2) = \Delta v(2)$.

Observe that we never have to take two identical sets in any minimum set cover. This can be generalised to sets Q and R that are not identical, but where R is a subset of Q: whenever we take R in the set cover, we could equally well have taken Q, moreover, any set cover containing both R and Q can be replaced by a smaller cover by removing R. This leads to the following reduction rule:

Reduction Rule 5.3.

if there exist sets $Q, R \in \mathcal{S}$ such that $R \subseteq Q$ **then**
  return $\text{MSC}(\mathcal{S} \setminus \{R\}, \mathcal{U})$

Reduction Rules 5.1 and 5.3 together remove all sets of size one from an instance: either the element in a singleton set S has frequency one and is removed by Reduction Rule 5.1, or it has higher frequency and thus S is a subset of another set. Consequently, Reduction Rule 5.2 becomes obsolete after adding Reduction Rule 5.3 to the algorithm.

In the analysis of the new algorithm with Reduction Rule 5.3 added, we set $w(1) = 0$ because sets of size one are now removed. Notice that this violates the steepness inequalities. We solve this by imposing them only for $i \geq 3$:

$$\forall i \geq 3 \quad : \quad \Delta w(i-1) \geq \Delta w(i)$$

Observe what happens when our new algorithm branches on a set S containing r_i elements of frequency i. In the branch where we take S in the set cover, we still decrease the measure by $w(|S|) + \sum_{i=2}^{\infty} r_i v(i)$ for removing S and its elements. Recall that $\Delta w(|S|)$ is a lower bound on the decrease in measure obtained by reducing the size of a set by one; this is because of the steepness inequalities. Even though $w(1) = 0$ in our new analysis, we can still use $\Delta w(|S|) \sum_{i=2}^{\infty} r_i(i-1)$ as a lower bound on the additional decrease in measure due to the fact that no other set R containing elements from S can be a subset of S by Reduction Rule 5.3; therefore, we reduce R at most $|R|-1$ times in size, and the steepness inequalities still make sure that $\sum_{i=2}^{|R|} \Delta w(i) \geq (|R|-1)\Delta w(|S|)$.

In the branch where we discard S, we still decrease the measure by $w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i)$ for removing S and reducing the frequencies of its elements. Observe what happens to frequency-two elements. If $r_2 = 1$, we take at least one more set in the set cover and remove at least one more element since this set cannot be a subset

of S. For every additional frequency-two element in S, the size of the set that is taken in the set cover increases by one in the worst case, unless $r_2 = |S|$. In the last case, all elements cannot be in the same other set because this would make the set larger than S which cannot be the case by the branching rule. Since $w(2) = \Delta w(2)$, this leads to an additional decrease in measure of at least $[r_2 > 0](v(2) + \sum_{i=1}^{\min\{r_2,|S|-1\}} \Delta w(i+1))$ by the above discussion. Finally, if $r_2 = |S|$, then we remove at least one more set, decreasing the measure by at least $w(2)$ more.

This leads to the following set of recurrence relations:

$$\forall\, |S| \geq 2, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\Delta k_{\text{take}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1)$$

$$\Delta k_{\text{discard}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + [r_2 > 0] \left(v(2) + \sum_{i=1}^{\min\{r_2,|S|-1\}} \Delta w(i+1) \right) + [r_2 = |S|] w(2)$$

We solve the associated numerical optimisation problem and obtain a solution of $N(k) \leq 1.37787^k$ on the recurrence relations using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.089132	0.304202	0.377794	0.402590	0.408971	0.408971	0.408971
$w(i)$	0.000000	0.646647	0.853436	0.939970	0.979276	0.995872	1.000000	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.37787^{(0.408971+1)n}\big) = \mathcal{O}(1.57087^n)$ on the running time of the algorithm. The bounding cases of the numerical optimisation problem are:

$$\begin{array}{lllll} |S| = r_2 = 2 & |S| = r_3 = 2 & |S| = r_3 = 3 & |S| = r_4 = 3 & |S| = r_4 = 4 \quad |S| = r_5 = 4 \\ |S| = r_5 = 5 & |S| = r_6 = 5 & |S| = r_6 = 6 & |S| = r_7 = 6 & |S| = r_7 = 7 \end{array}$$

We notice that this analysis is not tight in the following way. The quantities we use for Δk_{take} and $\Delta k_{\text{discard}}$ can both correspond to a real instance on which the algorithm branches. That is, there are real instances to which the reduction in the measure is bounded tightly. However, for some considered cases, there is no real instance in which the reduction in the measure is tight for both the inequality for Δk_{take} and the inequality for $\Delta k_{\text{discard}}$. This even happens on the bounding case $|S| = r_2 = 2$.

We could give a better analysis of the current algorithm. However, this requires a different set-up with some more case analysis. We feel that it is better to ignore this for the moment and continue with new reduction rules for instances tight for any of the individual values of Δk_{take} and $\Delta k_{\text{discard}}$. We will give such a case analysis only for our final algorithm: Algorithm 5.4. This case analysis is given in Appendix A.2.

5.3.5. Improvement Step Four: All Sets Have Size at Most Two

The case $|S| = r_2 = 2$ corresponds to a set S containing two frequency-two elements whose second occurrences are in different sets. In this case, all sets have cardinality at most two since our algorithm branches only on sets of maximum cardinality. It has

Algorithm 5.3. The algorithm of Fomin, Grandoni, and Kratsch [144].

Input: a set cover instance $(\mathcal{S}, \mathcal{U})$
Output: a minimum set cover of $(\mathcal{S}, \mathcal{U})$
MSC$(\mathcal{S}, \mathcal{U})$:
1: **if** $\mathcal{S} = \emptyset$ **then return** $\emptyset$
2: Let $S \in \mathcal{S}$ be a set of maximum cardinality among the sets in $\mathcal{S}$
3: **if** there exists an element $e \in \mathcal{U}$ of frequency one **then**
4: **return** $\{R\} \cup \text{MSC}(\{R' \setminus R \mid R' \in \mathcal{S} \setminus \{R\}\}, \mathcal{U} \setminus R)$, where R is such that $e \in R$
5: **else if** there exist sets $Q, R \in \mathcal{S}$ such that $R \subseteq Q$ **then**
6: **return** MSC$(\mathcal{S} \setminus \{R\}, \mathcal{U})$
7: **else if** $|S| \leq 2$ **then**
8: **return** a minimum set cover computed by using maximum matching
9: Let $\mathcal{C}_1 = \{S\} \cup \text{MSC}(\{S' \setminus S \mid S' \in \mathcal{S} \setminus \{S\}\}, \mathcal{U} \setminus S)$ and $\mathcal{C}_2 = \text{MSC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$
10: **return** the smallest set cover from $\mathcal{C}_1$ and $\mathcal{C}_2$

often been observed that this case can be solved in polynomial time by computing a maximum matching; see for example [144].

In this situation, we can construct a set cover in the following way. We initially repeatedly pick sets that each cover two elements until only sets containing one thus far uncovered element remain. The maximum number of sets that cover two elements per set are used in a minimum set cover. We can find such a maximum set of disjoint size two sets by computing a maximum matching in the following graph $H = (V, E)$: introduce a vertex for every element $e \in \mathcal{U}$, and an edge (e_1, e_2) for every set of size two $\{e_1, e_2\} \in \mathcal{S}$. A maximum matching M in H can be computed in polynomial time [120]. Given M, we can construct a minimum set cover by taking the sets corresponding to the edges in M and add an additional set for each vertex that is not incident to an edge in M.

This leads to the following reduction rule.

Reduction Rule 5.4.

Let $S \in \mathcal{S}$ be a set of maximum cardinality
if $|S| \leq 2$ **then**
return a minimum set cover computed by using maximum matching

If we add this reduction rule to the algorithm of Section 5.3.4, we obtain the algorithm of Fomin, Grandoni, and Kratsch [144]: Algorithm 5.3.

The numerical optimisation problem used to compute the upper bound on the running time of Algorithm 5.3 is the same as the numerical optimisation problem used to compute the upper bound on the running time of the algorithm of Section 5.3.4, except for the fact that we consider branching on only sets S with $|S| \geq 3$.

$$\forall\, |S| \geq 3, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\Delta k_{\text{take}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1)$$

$$\Delta k_{\text{discard}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + [r_2 > 0] \left(v(2) + \sum_{i=1}^{\min\{r_2, |S|-1\}} \Delta w(i+1) \right) + [r_2 = |S|] w(2)$$

We again solve the associated numerical optimisation problem. We obtain a solution of $N(k) \leq 1.28505^k$ on the recurrence relations using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.261245	0.526942	0.620173	0.652549	0.661244	0.661244	0.661244
$w(i)$	0.000000	0.370314	0.740627	0.892283	0.961123	0.991053	1.000000	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.28505^{(0.661244+1)n}\big) = \mathcal{O}(1.51685^n)$ on the running time of Algorithm 5.3. The bounding cases of the numerical optimisation problem are:

$$|S| = r_2 = 3 \quad |S| = r_3 = 3 \quad |S| = r_4 = 3 \quad |S| = r_4 = 4 \quad |S| = r_5 = 4 \quad |S| = r_5 = 5$$
$$|S| = r_6 = 5 \quad |S| = r_6 = 6 \quad |S| = r_7 = 6 \quad |S| = r_7 = 7$$

We note that the analysis in [144] gives an upper bound on the running time of Algorithm 5.3 of $\mathcal{O}(1.5263^n)$. The difference with our better upper bound comes from the fact that we allow $v_{\max}$ to be variable in the associated numerical optimisation problem, while $v_{\max} = 1$ is taken in the analysis in [144].

5.3.6. Improvement Step Five: Subsumption

Again, we consider the 'smallest' bounding case: $|S| = r_2 = 3$. The reduction in the measure in the inequality for $\Delta k_{\text{discard}}$ in the previous section is tight for the following real instance. We have a set S of cardinality three containing three elements of frequency two, and these three frequency-two elements together have their second occurrences in two sets, each containing the same extra frequency-two element such that they are not subsets of S. In other words, we have $S = \{e_1, e_2, e_3\}$ existing next to $\{e_1, e_2, e_4\}, \{e_3, e_4\}$ with $f(e_1) = f(e_2) = f(e_3) = f(e_4) = 2$.

We can reduce this case by introducing the notion of subsumption. We say that an element e_1 is *subsumed* by e_2 if $\mathcal{S}(e_1) \subseteq \mathcal{S}(e_2)$, i.e., if the collection of sets containing e_1 is a subset of the collection of sets containing e_2. Notice that in this case, any collection of sets that covers e_1 will always cover e_2 also. Therefore, an algorithm can safely remove e_2 from the instance to obtain a smaller, equivalent instance. After solving this equivalent instance, it can then put e_2 back in the appropriate sets to get a solution to the original instance.

This leads to the following reduction rule.

Reduction Rule 5.5.

if there exist two elements e_1 and e_2 such that $\mathcal{S}(e_1) \subseteq \mathcal{S}(e_2)$ **then**
  return MSC($\{R \setminus \{e_2\} \mid R \in \mathcal{S}\}, \mathcal{U} \setminus \{e_2\}$) with e_2 put back in the sets in $\mathcal{S}(e_2)$

We will now analyse our algorithm, which gives a first improvement over the algorithm by Fomin, Grandoni and Kratsch: Algorithm 5.3 augmented with Reduction

Rule 5.5. In the branch where S is taken in the set cover, nothing changes. In the branch where we discard S, all sets containing the occurrence outside S of a frequency-two element from S are taken in the set cover. These are at least r_2 sets since no two frequency-two elements can have both occurrences in the same two sets by Reduction Rule 5.5. Hence, we decrease the measure by at least $r_2 w(2)$. The measure is further decreased because these removed sets contain elements that are removed also. Moreover, this removal reduces the cardinality of other sets. We bound the total decrease of the measure from below by considering only the cases where $r_2 \in \{1, 2, 3\}$ in more detail. If $r_2 \in \{1, 2\}$, then at least one extra element is removed which we bound from below by $v(2)$; this is tight if $r_2 = 2$. If $r_2 = 3$, then the three sets can either contain the same extra element which gives a decrease of $v(3)$, or these contain more extra elements giving a decrease of at least $2v(2)$ and an additional $\Delta w(|S|)$ because these exist in other sets also.

This leads to the following set of recurrence relations:

$$\forall\, |S| \geq 3, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\begin{aligned}
\Delta k_{\text{take}} &\geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1) \\
\Delta k_{\text{discard}} &\geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + r_2 w(2) \\
&\quad + [r_2 \in \{1,2\}] v(2) + [r_2 = 3] \min\{v(3),\ 2v(2) + \Delta w(|S|)\}
\end{aligned}$$

We obtain a solution of $N(k) \leq 1.28886^k$ on the recurrence relations using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.218849	0.492455	0.589295	0.623401	0.632777	0.632777	0.632777
$w(i)$	0.000000	0.367292	0.734584	0.886729	0.957092	0.988945	1.000000	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.28886^{(0.632777+1)n}\big) = \mathcal{O}(1.51335^n)$ on the running time of the algorithm. The bounding cases of the numerical optimisation problem are:

$$\begin{array}{lllllll}
|S| = r_2 = 3 & |S| = r_3 = 3 & |S| = r_4 = 3 & |S| = r_4 = 4 & |S| = r_5 = 4 & |S| = r_5 = 5 \\
|S| = r_6 = 5 & |S| = r_6 = 6 & |S| = r_7 = 6 & |S| = r_7 = 7 &
\end{array}$$

5.3.7. Improvement Step Six: Counting Arguments

The new 'smallest' worst case of our algorithm is $|S| = r_2 = 3$. Although this is the same case as in the previous improvement, there is a different instance for which the inequality for $\Delta k_{\text{discard}}$ of the previous section is tight. This corresponds to a set S containing three elements of frequency two that all have their second occurrence in a different set of size two, and, since the optimal weights in the analysis of Section 5.3.6 satisfy $v(3) < v(2) + \Delta w(|S|)$, these sets all contain the same second element which is of frequency three. Thus, we have the following situation: $S = \{e_1, e_2, e_3\}$ existing

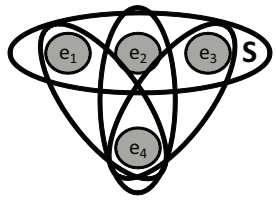


Figure 5.1. An instance corresponding to a bounding case in the analysis in Section 5.3.6.

next to $\{e_1, e_4\}, \{e_2, e_4\}, \{e_3, e_4\}$ with $f(e_1) = f(e_2) = f(e_3) = 2$ and $f(e_4) = 3$; see Figure 5.1.

We notice that we do not have to branch on this set S. Namely, if we take S in the set cover, we cover three elements using one set, while if we discard S and thus take all other sets containing e_1, e_2 and e_3, then we cover four elements using three sets. We can cover the same four elements with only two sets if we take S and any of the three sets containing e_4. Therefore, we can safely take S in the set cover without branching.

This counting argument can be generalised. For any set R, let q be the number of elements in the sets containing a frequency-two element from R that are not contained in R themselves, i.e., $q = \left|\left(\bigcup_{e \in R, f(e)=2, Q \in \mathcal{S}(e)} Q\right) \setminus R\right|$. Also, let r_2 be the number of elements of frequency two in R. We cover $|R|$ elements using one set if we take R in the set cover, and we cover $q + |R|$ elements using r_2 sets if we discard R. Thus, if $q < r_2$, taking R is always as least as good as discarding R since then we use $r_2 - 1$ sets less while also covering $q < r_2$ less elements, i.e., $q \leq r_2 - 1$ less elements; we can always cover these elements by picking one additional set per element.

This leads to the following reduction rule:

Reduction Rule 5.6.
if there exists a set $R \in \mathcal{S}$ s.t. $\left|\left(\bigcup_{e \in R, f(e)=2, Q \in \mathcal{S}(e)} Q\right) \setminus R\right| < |\{e \in R \mid f(e) = 2\}|$
then
return $\{R\} \cup \mathrm{MSC}(\{R' \setminus R \mid R' \in \mathcal{S} \setminus \{R\}\}, \mathcal{U} \setminus R)$

We add Reduction Rule 5.6 to the algorithm of Section 5.3.6 and analyse its running time. If S is discarded, we now know that at least r_2 sets are removed due to Reduction Rule 5.5 and that, due to Reduction Rule 5.6, these sets contain at least r_2 elements that are also removed. This gives a decrease in measure of at least $r_2(v(2) + w(2))$. Furthermore, additional sets are reduced in cardinality because of the removal of these elements. The decrease in measure due to reducing the cardinality of these sets can not be bounded from below by $r_2 \Delta w(|S|)$. This is because repeatedly reducing the cardinality of the same set R can remove this set entirely while the steepness inequalities do not guarantee that this decrease in measure is bounded by $|R|\Delta w(|S|)$ since $w(1) = 0$. Therefore, the smallest possible decrease in measure is obtained by either exploiting this situation repeatedly by putting as many additionally removed elements in sets of size two as possible, or we can again bound this additional decrease in measure from

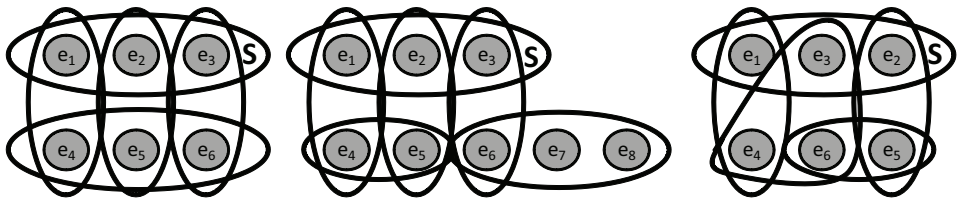


Figure 5.2. Three instances corresponding to a bounding case in the analysis in Section 5.3.7.

below by $r_2\Delta w(|S|)$. The steepness inequalities now do guarantee that the smallest possible decrease in measure due to the removal of these elements is bounded from below by $\min\{r_2\Delta w(|S|), \lfloor \frac{r_2}{2} \rfloor w(2) + (r_2 \bmod 2)\Delta w(|S|)\}$.

This leads to the following set of recurrence relations:

$$\forall\, |S| \geq 3, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\begin{aligned} \Delta k_{\text{take}} &\geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1) \\ \Delta k_{\text{discard}} &\geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + r_2(v(2) + w(2)) + \\ &\quad \min\Big\{ r_2\Delta w(|S|), \Big\lfloor \frac{r_2}{2} \Big\rfloor w(2) + (r_2 \bmod 2)\Delta w(|S|) \Big\} \end{aligned}$$

We obtain a solution of $N(k) \leq 1.29001^k$ on the recurrence relations using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.127612	0.432499	0.544653	0.587649	0.602649	0.606354	0.606354
$w(i)$	0.000000	0.364485	0.728970	0.881959	0.953804	0.987224	0.999820	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.29001^{(0.606354+1)n}\big) = \mathcal{O}(1.50541^n)$ on the running time of the algorithm. The bounding cases of numerical optimisation problem are:

$$\begin{array}{lllllll} |S| = r_2 = 3 & |S| = r_3 = 3 & |S| = r_4 = 3 & |S| = r_4 = 4 & |S| = r_5 = 4 & |S| = r_5 = 5 \\ |S| = r_6 = 5 & |S| = r_6 = 6 & |S| = r_7 = 6 & |S| = r_7 = 7 & |S| = r_8 = 7 & |S| = r_8 = 8 \end{array}$$

5.3.8. The Final Improvement: Folding Some Sets of Size Two

We now present the final improvement step. This improvement step leads to the following result, where the associated Set Cover algorithm is Algorithm 5.4.

Theorem 5.1. *There exists an algorithm that solves* Dominating Set *in* $\mathcal{O}(1.4969^n)$ *time and polynomial space.*

For this final improvement step, we again look at the case $|S| = r_2 = 3$ and closely inspect instances that are tight for the inequality with $\Delta k_{\text{discard}}$ of the previous section. Currently, this corresponds to a set $S = \{e_1, e_2, e_3\}$ containing three elements of frequency two whose second occurrences give rise to one of the following three situations; see also Figure 5.2:

1. $\{e_1, e_4\}, \{e_2, e_5\}, \{e_3, e_6\}$ with e_4, e_5 and e_6 elements of frequency two that also occur in a set $\{e_4, e_5, e_6\}$.
2. $\{e_1, e_4\}, \{e_2, e_5\}, \{e_3, e_6\}$ with e_4, e_5 and e_6 elements of frequency two that also occur in sets $\{e_4, e_5\}$ and $\{e_6, e_7, e_8\}$, where e_7 and e_8 are elements of any frequency.
3. $\{e_1, e_4\}, \{e_2, e_5\}, \{e_3, e_4, e_6\}$ with e_4, e_5 and e_6 elements of frequency two, and where e_5 and e_6 also occur in a set $\{e_5, e_6\}$.

Notice that the optimal weights in the analysis of Section 5.3.7 satisfy $2w(2) = w(3)$, i.e., $w(2) = \Delta w(3)$. Hence, in all three cases, the measure is decreased by the same amount in the branch where S is discarded: $w(3) + 3v(2)$ for removing S plus an additional $3v(2) + 5w(2)$ due to the reduction rules.

We add the following reduction rule that removes any set of cardinality two containing two frequency-two elements to our algorithm and obtain Algorithm 5.4.

Reduction Rule 5.7.

if there exists a set $R \in \mathcal{S}$, $R = \{e_1, e_2\}$, with $f(e_1) = f(e_2) = 2$ **then**
  Let $\mathcal{S}(e_i) = \{R, R_i\}$, for $i = 1, 2$, and let $Q = (R_1 \cup R_2) \setminus R$
  Let $\mathcal{C} = \text{MSC}((\mathcal{S} \setminus \{R, R_1, R_2\}) \cup \{Q\}, \mathcal{U} \setminus R)$
  if $Q \in \mathcal{C}$ **then**
    return $(\mathcal{C} \setminus \{Q\}) \cup \{R_1, R_2\}$
  else
    return $\mathcal{C} \cup \{R\}$

If there exists a set R of cardinality two containing two frequency-two elements e_1, e_2, such that e_i occurs in R and R_i, then the reduction rule transforms this instance into an instance where R, R_1 and R_2 have been replaced by the set $Q = (R_1 \cup R_2) \setminus R$.

Lemma 5.2. *Reduction Rule 5.7 is correct.*

Proof. Let S, R, R_1, and R_2 be as in the formulation of Reduction Rule 5.7. Notice that there exist a minimum set cover of $(\mathcal{S}, \mathcal{U})$ that either contains R, or contains both R_1 and R_2. This is true because if we take only one set from R, R_1 and R_2, then this must be R since we must cover e_1 and e_2; if we take two, then it is of no use to take R since the other two cover more elements; and, if we take all three, then the set cover is not minimal. The rule postpones the choice between the first two possibilities, taking Q in the minimum set cover of the transformed problem if both R_1 and R_2 are in a minimum set cover, or taking no set in the minimum set cover of the transformed problem is R is in a minimum set cover. This works because the transformation preserves the fact that the difference between both possibilities in the number of sets we take in the set cover is one. Hence, Reduction Rule 5.7 is correct. □

Algorithm 5.4. Our final algorithm for the set cover modelling of dominating set.

Input: a set cover instance $(\mathcal{S}, \mathcal{U})$
Output: a minimum set cover of $(\mathcal{S}, \mathcal{U})$
MSC$(\mathcal{S}, \mathcal{U})$:
1: **if** $\mathcal{S} = \emptyset$ **then return** $\emptyset$
2: Let $S \in \mathcal{S}$ be a set of maximum cardinality among the sets in $\mathcal{S}$
3: **if** there exists an element $e \in \mathcal{U}$ of frequency one **then**
4: **return** $\{R\} \cup \text{MSC}(\{R' \setminus R \mid R' \in \mathcal{S} \setminus \{R\}\}, \mathcal{U} \setminus R)$, where R is such that $e \in R$
5: **else if** there exist sets $Q, R \in \mathcal{S}$ such that $R \subseteq Q$ **then**
6: **return** MSC$(\mathcal{S} \setminus \{R\}, \mathcal{U})$
7: **else if** there exist two elements e_1 and e_2 such that $\mathcal{S}(e_1) \subseteq \mathcal{S}(e_2)$ **then**
8: **return** MSC$(\{R \setminus \{e_2\} \mid R \in \mathcal{S}\}, \mathcal{U} \setminus \{e_2\})$ with e_2 put back in the sets in $\mathcal{S}(e_2)$
9: **else if** there exists a set $R \in \mathcal{S}$ such that $\left|\left(\bigcup_{e \in R, f(e)=2, Q \in \mathcal{S}(e)} Q\right) \setminus R\right| < |\{e \in R \mid f(e) = 2\}|$ **then**
10: **return** $\{R\} \cup \text{MSC}(\{R' \setminus R \mid R' \in \mathcal{S} \setminus \{R\}\}, \mathcal{U} \setminus R)$
11: **else if** there exists a set $R \in \mathcal{S}$, $R = \{e_1, e_2\}$, with $f(e_1) = f(e_2) = 2$ **then**
12: Let $\mathcal{S}(e_i) = \{R, R_i\}$, for $i = 1, 2$, and let $Q = (R_1 \cup R_2) \setminus R$
13: Let $\mathcal{C} = \text{MSC}((\mathcal{S} \setminus \{R, R_1, R_2\}) \cup \{Q\}, \mathcal{U} \setminus R)$
14: **if** $Q \in \mathcal{C}$ **then**
15: **return** $(\mathcal{C} \setminus \{Q\}) \cup \{R_1, R_2\}$
16: **else**
17: **return** $\mathcal{C} \cup \{R\}$
18: **else if** $|S| \leq 2$ **then**
19: **return** a minimum set cover computed by using maximum matching
20: Let $\mathcal{C}_1 = \{S\} \cup \text{MSC}(\{S' \setminus S \mid S' \in \mathcal{S} \setminus \{S\}\}, \mathcal{U} \setminus S)$ and $\mathcal{C}_2 = \text{MSC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$
21: **return** the smallest set cover from $\mathcal{C}_1$ and $\mathcal{C}_2$

We note that this reduction rule is similar to the vertex folding rule used in many papers on Independent Set, see for example Chapter 7 or [144].

Notice that we need additional constraints on the weights to analyse Algorithm 5.4. Namely, this reduction rule should not be allowed to increase the measure of an instance. For the moment, we forget that the elements e_1, e_2 and S are removed, and demand that the measure is not increased by removing R_1 and R_2 and adding Q. To this end, we impose the following additional constraint for all $i, j \geq 2$:

$$\forall i, j \geq 2 \quad : \quad w(i) + w(j) \geq w(i + j - 2)$$

For a proper analysis, we need to further subdivide our case analysis by differentiating between different kinds of elements of frequency two depending on the kind of set their second occurrence is in. This leads to a tedious case analysis that we delegated to Appendix A.2. This case analysis again results in a numerical optimisation problem whose optimum gives an upper bound on the running time of Algorithm 5.4. Theorem 5.1 is proved as a result. For the details, see Appendix A.2.

5.4. Intuition Why Further Improvement is Hard

In the step-by-step design process of Section 5.3, we obtained improved algorithms by adding polynomial-time reduction rules that deal with worst-case instances of a previous algorithm. We stopped with this design process after obtaining Algorithm 5.4. We stopped here because it seems hard to further improve the algorithm *significantly* using the same approach. In this section, we will give some considerations as to why this is so.

Adding New Reduction Rules for Cases with Frequency-Two Elements. We acknowledge that it is possible to improve one of the new 'small' worst cases, namely, $|S| = 4, r_2 = r_{e\geq 4} = 4$ (for the meaning of $r_{e\geq 4}$ see the analysis in Appendix A.2). We tried to do so by introducing the following reduction rule that deals with each connected component separately.

Reduction Rule 5.8.

if $\mathcal{S}$ contains multiple connected components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_l$ **then**
return $\bigcup_{i=1}^{l} \mathrm{MSC}(\mathcal{C}_i, \bigcup_{S\in\mathcal{C}_i} S)$

Here, a connected component $\mathcal{C}$ of a set cover instance $(\mathcal{S}, \mathcal{U})$ is an inclusion-minimal, non-empty subset $\mathcal{C} \subseteq \mathcal{S}$ for which all elements in the sets in $\mathcal{C}$ do not occur in $\mathcal{S} \setminus \mathcal{C}$. Note that this definition of a connected component is a natural extension of the same concept for graphs.

However, introducing this new reduction rule helps only marginally in obtaining better bounds with our current type of analysis. To see this, consider what we have been doing in the last few improvement steps. In each of these steps, the 'smallest' worst case involved frequency-two elements, and we looked for new reduction rules dealing with these elements more efficiently. In Proposition 5.3 below, we will see that it is most likely not possible to completely remove frequency-two elements. But what if we could formulate sufficiently powerful reduction rules such that they never occur in any of the worst cases of the algorithm? We may not have such an algorithm, but we can analyse such an algorithm in the same way as we did in Section 5.3.

The best upper bound on the running time we could prove for such an algorithm in exactly the same way as in the previous section corresponds to the solution of the numerical optimisation problem associated with the following set of recurrence relations:

$$\forall\, |S| \geq 3, \forall\, r_i : \sum_{i=2}^{\infty} r_i = |S| \quad : \quad N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\Delta k_{\text{take}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i (i-1) + [r_2 > 0]\infty$$

$$\Delta k_{\text{discard}} \geq w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + r_2(v(2) + w(2)) + [r_2 > 0]\infty$$

In this set of recurrences, we use the value ∞ to make Δk_{take} and $\Delta k_{\text{discard}}$ large enough not to appear as a bounding case whenever $r_2 > 0$. We note that we do so under the convention that $[r_2 > 0]\infty = 0$ if $r_2 = 0$.

We obtain a solution of $N(k) \leq 1.26853^k$ on the recurrence relations using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.000000	0.426641	0.607747	0.671526	0.687122	0.690966	0.690966
$w(i)$	0.000000	0.353283	0.706566	0.866101	0.948043	0.985899	1.000000	1.000000

This leads to an upper bound on the running time of such a hypothetical algorithm proven by these methods of $\mathcal{O}\big(1.26853^{(0.690966+1)n}\big) = \mathcal{O}(1.4952^n)$. The bounding cases of the numerical optimisation problem are

$|S| = r_3 = 3$ $|S| = r_4 = 3$ $|S| = r_5 = 4$ $|S| = r_6 = 4$ $|S| = r_6 = 5$ $|S| = r_6 = 6$
$|S| = r_7 = 6$ $|S| = r_7 = 7$ $|S| = r_{>7} = 7$ $|S| = r_{>7} = 8$

Of course, this is not a lower bound on the complexity of Dominating Set. This is a lower bound on the upper bounds on running times we can get by improving only on the cases involving elements of frequency two on algorithms using the set cover formulation of dominating set that considers to branch on a single set only. It shows that if we put in a lot of effort to try and improve Algorithm 5.4 in exactly the same way as in Section 5.3 by doing an extensive case analysis, then we can improve the running time only marginally.

Removing Frequency-Two Elements Completely. A case analysis to improve the cases containing frequency-two elements may help only marginally, but why can we not remove these cases completely? If all sets have size two, then we can solve the problem in polynomial time using maximum matching (Section 5.3.5); can we not do something similar for elements of frequency two? There are reasons to believe that it is necessary to consider these elements of frequency two: we most likely cannot remove them completely by reduction rules, nor is it likely that we can solve the instance in which all elements have frequency two in subexponential time. This is for the following reason.

Proposition 5.3. *There is no polynomial-time algorithm that solves minimum set cover where all elements have frequency at most two and all sets have cardinality at most three, unless $\mathcal{P} = \mathcal{NP}$. Moreover, under the Exponential-Time Hypothesis, there is not even an algorithm for the problem that runs in subexponential time.*

Proof. Consider an instance $G = (V, E)$ of the Vertex Cover problem with maximum degree three. From this instance, we build an equivalent Set Cover instance: for each edge introduce an element of frequency two, and for each vertex introduce a set containing the elements representing the edges it is incident to. Notice that the sets have cardinality at most three. It is easy to see that a minimum set cover of the constructed instance corresponds to a minimum vertex cover in G.

Therefore, a polynomial-time algorithm as in the statement of the proposition would solve minimum vertex cover on graphs of maximum degree three in polynomial time, which is impossible unless $\mathcal{P} = \mathcal{NP}$. And, such an algorithm running in subexponential time would solve Vertex Cover restricted to graphs of maximum degree three in subexponential time. Johnson and Szegedy showed that this is impossible unless the Exponential-Time Hypothesis fails [197]. □

Changing the Branching Rules. Another possibility to improve the worst-case behaviour of the algorithm is not to iteratively introduce new reduction rules, but to iteratively change the branching rule on specific local structures instead. In this chapter, we did try to modify the branching rule. This is because branching on a set S of maximum cardinality has two advantages. Firstly, a set of maximal cardinality has maximal impact on the instance in the sense that the maximum number of other sets and elements are removed or reduced. Secondly, it allows us to use the cardinality of S as an upper bound on the cardinality of any other set in the instance.

Of course, we could use different branching rules that, for example, branch on larger local configurations. This could improve the algorithm but would also lead to an analysis involving many more subcases. In this case, we could not deal with these many more subcases manually, but it would be interesting to see if such analysis can be done using new techniques involving, for example, automated case analysis. In Chapter 6, we show that changing the branching rules in a similar step-by-step improvement procedure gives good results for a series of edge-domination problems: Edge Dominating Set, Minimum Maximal Matching, Matrix Dominating Set, and weighted version of these problems.

5.5. Additional Results on Related Problems

In this section, we show that the algorithm of Theorem 5.1 can be used to obtain faster algorithms for more problems than only Dominating Set. The applications that we give include some in the field of parameterised algorithms.

The algorithm of Theorem 5.1 is the currently fastest algorithm for Dominating Set. This algorithm uses the modelling of Dominating Set as a Set Cover presented in Section 5.1.1 and then applies Algorithm 5.4. Because of this construction, we can obtain similar results for other graph domination problems that can be modelled in this way as well. We note that this does not lead to a general result on Set Cover, but only on Set Cover instances with $|\mathcal{S}| = |\mathcal{U}| = n$, as the weights in the analysis are chosen to minimise $\alpha^{v_{\max}+w_{\max}}$.

Probably, the most well-known other graph domination problem to which this applies is Total Dominating Set. We recall its definition from Chapter 1. A *total dominating set* $D \subseteq V$ in a graph G is a set of vertices such that every vertex $v \in V$ is adjacent to some vertex in D, i.e., $\bigcup_{v \in D} N(v) = V$. Different to a dominating set, vertices in the set D also need to have a neighbour in D.

Total Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a total dominating set $D \subseteq V$ in G of size at most k?

Corollary 5.4. *There exists an algorithm that solves* Total Dominating Set *in* $\mathcal{O}(1.4969^n)$ *time and polynomial space.*

Proof. An instance of Total Dominating Set can also be modelled as an instance of Set Cover as follows. Let the universe $\mathcal{U}$ of the instance equal V, i.e., $\mathcal{U} = V$. Now, introduce a set S_v with $S_v = N(v)$ for every $v \in V$, i.e., $\mathcal{S} = \{N(v) \mid v \in V\}$.

Clearly, a set cover of the instance $(\mathcal{S}, \mathcal{U})$ corresponds to a total dominating set in G of equal size. We prove the result by applying Algorithm 5.4 to this instance and using the same analysis as in the proof of Theorem 5.1. □

Remark 5.1. There exist many more variants of Dominating Set that can be modelled as a Set Cover instance with at most n sets and at most n elements and that can consequently be solved in $\mathcal{O}(1.4969^n)$ time and polynomial space. Examples include Directed Dominating Set (the equivalent in directed graphs), Distance-r Dominating Set (where vertices must be at distance at most ℓ from a vertex in the dominating set), Strong Dominating Set (where vertices can be dominated only by vertices of equal or larger degree), and Weak Dominating Set (where vertices can be dominated only by vertices of equal or smaller degree).

Besides graph domination problems that can be modelled as Set Cover in such a way that the resulting Set Cover instances satisfy $|\mathcal{S}| = |\mathcal{U}| = n$, there are also graph domination problems whose equivalent Set Cover instances have different sizes. We now give an example of such a problem, namely Red-Blue Dominating Set, and show that we can also use Algorithm 5.4 to obtain a faster algorithm for this problem. We note that we will improve this result at the cost of using exponential space in Section 8.5.

A *red-blue dominating set* in a (bipartite) graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertices $\mathcal{R}$ and blue vertices $\mathcal{B}$ is a subset $D \subseteq \mathcal{R}$ such that $N(D) \supseteq \mathcal{B}$.

Red-Blue Dominating Set
Input: A bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertices $\mathcal{R}$ and blue vertices $\mathcal{B}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a red-blue dominating set $D \subseteq \mathcal{R}$ in G of size at most k?

The previous fastest algorithms for this problem are the $\mathcal{O}(1.2354^n)$-time and polynomial-space algorithm and the $\mathcal{O}(1.2303^n)$-time and exponential-space algorithm that can both easily be derived from the results in [144].

Corollary 5.5. *There exists an algorithm that solves* Red-Blue Dominating Set *in* $\mathcal{O}(1.2279^n)$ *time and polynomial space.*

Proof. It is not hard to see that an instance $G = (\mathcal{R} \cup \mathcal{B}, E)$ of Red-Blue Dominating Set can be transformed into an equivalent instance $(\mathcal{S}, \mathcal{U})$ of Set Cover by setting $\mathcal{U} = \mathcal{B}$ and $\mathcal{S} = \{N(v) \mid v \in \mathcal{R}\}$. We solve the instance $(\mathcal{S}, \mathcal{U})$ resulting from this transformation using Algorithm 5.4.

To prove an upper bound on the running time of this algorithm, we consider the same set of recurrence relations that are associated with the result in Theorem 5.1. Different to the proof of Theorem 5.1, we now solve the associated numerical problem while minimising $\alpha^{\max\{v_{\max}, w_{\max}\}}$ instead of $\alpha^{v_{\max}+w_{\max}}$. Because the instance $(\mathcal{S}, \mathcal{U})$ satisfies $|\mathcal{S}| + |\mathcal{U}| = n$, the instance has a measure k of at most $\max\{v_{\max}, w_{\max}\}n$. Hence, a running time of $\mathcal{O}(\alpha^{\max\{v_{\max}, w_{\max}\}n})$ follows from the computed value of α.

We solve the described numerical problem and obtain a solution of $N(k) \leq 1.22781^k$ using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.100936	0.606876	0.837893	0.933923	0.976152	0.996215	1.000000
$w(i)$	0.000000	0.374675	0.749351	0.903955	0.969394	0.996281	1.000000	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.22781^{\max\{1,1\}n}\big) = \mathcal{O}(1.2279^n)$ on the running time of the algorithm. □

The exact exponential-time algorithm of Theorem 5.1 also has applications in parameterised algorithms. Because the parameterised problem k-Dominating Set is $\mathcal{W}[2]$-complete [116], we cannot expect to find an $\mathcal{FPT}$-algorithm for this problem. However, we can obtain a faster algorithm for the closely related parameterised problem k-Nonblocker. This algorithm is obtained by using a small kernel for this problem and then applying Theorem 5.1.

We first define the problem formally and then give our results. A *non-blocking vertex set* $B \subseteq V$ is a set of vertices of G such that $V \setminus B$ is a dominating set in G.

***k*-Nonblocker**
Input: A graph $G = (V, E)$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a non-blocking set $D \subseteq V$ in G of size at least k?

Alternatively, this problem asks whether there exists a dominating set of size at most $n-k$ in G. I.e., k-Nonblocker equals k-Dominating Set with the parameter k replaced by $n - k$. A pair of problems that can be transformed into each other by changing the parameter from k to $n - k$ (all minus k) are called *parametric duals* [201, 233]. That is, k-Nonblocker is the parametric dual of k-Dominating Set.

To obtain the currently fastest parameterised algorithm for this problem we use the following result of Dehne et al. [104][2] which is a corollary of a graph-theoretic result by McCuaig and Shepherd [234].

Proposition 5.6 ([104]). *k-Nonblocker admits a kernel of size at most $\frac{5}{3}k$.*

Combining Proposition 5.6 and Theorem 5.1 gives the following result.

Corollary 5.7. *There exists an algorithm that solves k-Nonblocker in $\mathcal{O}^*(1.9588^k)$ time and polynomial space.*

Proof. Apply the kernelisation algorithm of Proposition 5.6. This algorithm uses polynomial time and either outputs that the instance is a Yes-instance, or obtains an equivalent instance H with $n \leq \frac{5}{3}k$. In the second case, we solve the new instance H by computing a minimum dominating set in H using Theorem 5.1. The resulting running time equals $\mathcal{O}(1.4969^n) = \mathcal{O}^*(1.4969^{\frac{5}{3}k}) = \mathcal{O}^*(1.9588^k)$. □

Remark 5.2. We note that the same result can be obtained in the following way. First, apply all the reduction rules of Algorithm 5.4 to a Set Cover instance obtained from the k-Nonblocker instance. Then, solve the instance in constant time if $|\mathcal{S}|+|\mathcal{U}| < 16$.

[2] This paper also claims a $\mathcal{O}(1.4123^k)$-time algorithm for k-Nonblocker. However, this result is incorrect as the authors forgot to square a cited result from [144]. We improve their result even after correcting this running time.

Otherwise, output that the instance is a YES-instance if $|\mathcal{S}| > \frac{5}{3}k$ and $|\mathcal{U}| > \frac{5}{3}k$. Finally, solve any remaining instance using Algorithm 5.4. This also correctly gives an $\mathcal{O}^*(1.9588^k)$-time and polynomial-space algorithm for k-NONBLOCKER.

Correctness follows from combining Theorem 5.1 with the proof from [299] that the graph-theoretic result by McCuaig and Shepherd [234] can be translated to corresponding set-cover instances.

5.6. Practical Issues Related to Solving the Associated Numerical Optimisation Problems

We notice that in order to apply the iterative improvement method to any problem, it is vital to have an efficient solver for the numerical optimisation programs associated with a measure-and-conquer analysis: having to wait for a very long time for an analysis to be completed and for the new bounding cases to be known at every iteration is not an option. Solvers for these numerical optimisation problems can be obtained in many ways, for example by means of random search. However, random search does not converge to the optimum quickly enough to handle larger problems involving more than ten variables and hundreds of thousands of subcases. For example, the numerical optimisation problem associated with Algorithm 5.4 is of this form.

In this section, we describe the algorithm that we have used to solve most[3] numerical optimisation problems that arise from the measure-and-conquer analyses in this thesis. This algorithm is a modification of Eppstein's smooth quasiconvex programming algorithm [124]. We implemented this algorithm in C++. Before going into more details, we note that an alternative way to solve these problems was found by Gaspers and Sorkin in [167], who rewrote the measure in such a way that a convex program is obtained. We note that when the research in this chapter was done, the approach of Gaspers and Sorkin was not yet known.

To describe our algorithm, we first introduce some notation. Let $\vec{w}$ be a finite dimensional vector of weights used for the weight functions in a measure-and-conquer analysis. Let D be the feasible polytope $D = \{\vec{w} \in \mathbb{R}^d \mid \mathbf{A}\vec{w} \leq \vec{b}\}$ of all feasible weight vectors $\vec{w}$ defined by the constraints that we imposed on the weights in the analysis. Examples of these constraints are the constraints that require the weights to be monotone, or the steepness inequalities. We denote row i of the constraint matrix $\mathbf{A}$ by $\vec{a}_i$, and the ith component of the right-hand-side vector $\vec{b}$ by b_i.

Let R be the set of recurrence relations involved in the measure-and-conquer analysis. For a given weight vector $\vec{w}$, we can compute a minimal solution $\alpha_r(\vec{w})$ to each of the recurrence relations $r \in R$ by computing the root of a polynomial; this is similar to the case where no weights are involved, see Section 2.1. Given these values $\alpha_r(\vec{w})$, we can compute an upper bound on the running time of the algorithm that we are analysing in the following way. First, we take the largest minimum solution $\alpha(\vec{w})$ over all recurrence relations in R: $\alpha(\vec{w}) = \max_{r \in R} \alpha_r(\vec{w})$. Then, we output $f(\alpha(\vec{w}), \vec{w})$ for some function f that is monotone in $\alpha(\vec{w})$ and that translates the upper bound on the recurrence relations to the base of the exponent in the upper bound on the running time

[3]The only exception is the numerical optimisation problem associated with the algorithms in Proposition 7.12. This analysis has been performed by N. Bourgeois, B. Escoffier and V. Th. Paschos.

of the algorithm we are analysing. To illustrate the purpose of the function f, consider an analysis of an algorithm for Dominating Set in Section 5.3. Here, we have the weight vectors $\vec{v}$ and $\vec{w}$, and the function f is the function $f(\alpha, \vec{v}, \vec{w}) = \alpha^{v_{\max}+w_{\max}}$ since $\mathcal{O}(\alpha^{(v_{\max}+w_{\max})n})$ is the corresponding upper bound on the running time. Notice that the fact that we use two weight vectors here makes no difference as they are finite dimensional and thus can be concatenated to one weight vector.

Notice that, instead of first computing the maximum over all recurrence relations and then evaluating f, one can also first evaluate $f(\alpha_r(\vec{w}), \vec{w})$ for each minimum solution $\alpha_r(\vec{w})$ to a recurrence relation $r \in R$, and then compute the corresponding maximum. This change of order does not influence the outcome as the weights $\vec{w}$ are fixed during the evaluation, and the function $f(\alpha, \vec{w})$ is monotone in the parameter α.

Now, we are ready to describe our algorithm. This algorithm is given an initial weight vector $\vec{w} \in D$. For some initial tolerance level l, and until some desired tolerance level l_d is obtained, our algorithm repeats the following steps.

1. For each $r \in R$, compute $\alpha_r(\vec{w})$ using the Newton-Raphson method.
2. Find $\alpha(\vec{w}) = \max_{r \in R} \alpha_r(\vec{w})$, and compute $f(\alpha_r(\vec{w}), \vec{w})$ for each $r \in R$.
3. Compute the set $R' \subseteq R$ of recurrences $r \in R$ with $f(\alpha_r(\vec{w}), \vec{w}) \geq f(\alpha(\vec{w}), \vec{w}) - l$.
4. For all $r \in R'$, compute the gradient $\nabla f(\alpha_r(\vec{w}), \vec{w})$.
5. Compute the set C of constraints (rows i from $\mathbf{A}$) for which $\vec{a}_i \cdot \vec{w} \geq b_i - c \cdot l$, for some fixed constant $0 < c < 1$ (our implementation uses $c = 0.01$).
6. Compute the vector $\vec{v}$ that maximises γ under the following set of constraints:
   (a) $\vec{v} \cdot \nabla f(\alpha_r(\vec{w}), \vec{w}) \geq \gamma$ for all $r \in R'$.
   (b) $\vec{a}_i \cdot \vec{v} \leq 0$ for all constraints $i \in C$.
   (c) $\vec{v}$ lies in the unit ball in $\mathbb{R}^d$.

   This can be done by solving a small quadratically constrained program. For this many solvers are available (e.g., CPLEX or MOSEK).
7. If no such vector exists, lower the tolerance (we set $l := 0.62l$), and restart from Step 3.
8. Otherwise, minimise $f(\alpha(\vec{w} + \epsilon\vec{v}), \vec{w} + \epsilon\vec{v})$ over all $\epsilon > 0$ such that $(\vec{w} + \epsilon\vec{v}) \in D$ by a line search based on standard one-dimensional optimisation techniques.
   We initially set $\epsilon = l$; then, we perform a doubling search (iteratively double ϵ) to find a point beyond the optimum (or on the boundary of D) on the half-line of the search direction; finally, we bisect the obtained interval of the line until we are sufficiently close to the optimum on this half-line.
9. Repeat from Step 1 using $\vec{w} := \vec{w} + \epsilon\vec{v}$ while lowering the tolerance if the difference between $f(\alpha(\vec{w}), \vec{w})$ and $f(\alpha(\vec{w} + \epsilon\vec{v}), \vec{w} + \epsilon\vec{v})$ is very small (less than $0.01 \cdot l$).

This algorithm tries to minimise $f(\alpha(\vec{w}), \vec{w})$ over all $\vec{w} \in D$. It does so by iteratively looking at the recurrences $r \in R'$ for which $f(\alpha_r(\vec{w}), \vec{w})$ is sufficiently close to the current maximum value $f(\alpha(\vec{w}), \vec{w})$. Then, it looks for a vector $\vec{v}$ such that $(\alpha(\vec{w} + \epsilon\vec{v}) \in D$ and $f(\alpha(\vec{w} + \epsilon\vec{v}), \vec{w} + \epsilon\vec{v})$ is smaller than $f(\alpha(\vec{w}), \vec{w})$ for some $\epsilon > 0$. The search direction $\vec{v}$ is chosen in such a way that it points maximally into the direction of the gradients $\nabla f(\alpha_r(\vec{w}), \vec{w})$ for all $r \in R'$. We note that these gradients can be found by means of implicit differentiation. Also, if $\vec{w}$ is very close to a boundary of D (represented by a constraint in C), the line search will not directly go towards this boundary because the constraints in C force $\vec{v}$ to point away from this boundary, or be parallel

to it. These two objectives are enforced by maximising γ under the conditions (a) and (b) when searching for this vector $\vec{v}$ in Step 6. After searching for a smaller value for $f(\alpha(\vec{w}), \vec{w})$ from $\vec{w}$ in the direction $\vec{v}$ by means of a bisecting line search, the algorithm repeats the process from Step 1. If the improvement in the last iteration was too small, then the algorithm lowers its tolerance level l. The algorithm terminates if l drops below some desired tolerance level l_d, then it outputs $\vec{w}$ and $f(\alpha(\vec{w}), \vec{w})$.

The main ideas behind this algorithm come from Eppstein's smooth quasiconvex programming algorithm [124]. Eppstein's algorithm is designed to solve quasiconvex optimisation problems that arise from analyses of branch-and-reduce algorithms. His algorithm tries to perform line searches in directions that are chosen based on gradients derived from recurrence relations whose solutions are sufficiently close to the current maximum value of these solutions. Compared to Eppstein, we have introduced three mayor modifications that we will outline below.

First of all, we added the function f that translates the upper bound on the recurrence relations in terms of the measure into an upper bound on the running time of the algorithm that is being analysed. This allows us to optimise the value of for example $f\alpha^{v_{\max}+w_{\max}}$. In earlier work, one could require only that $v_{\max} = w_{\max} = 1$, and then optimise α^2. This modification explains why we obtain faster running times for Algorithm 5.3 when compared to [144], and for the algorithm in Section 5.3.7 when compared to [299, 302]. We note that a similar change in the objective function of the optimisation was also used in [146].

A disadvantage of this modification is that we no longer have a proof that the function $f(\alpha(\vec{w}), \vec{w})$ is a quasiconvex function. This is a useful property since, by definition, a function $g(\vec{w})$ is quasiconvex if all its level sets $L_\lambda = \{\vec{w} \in D \mid g(\vec{w}) \leq \lambda\}$ are convex, and as a result, such a function has no local optima except for the global optimum. Eppstein's algorithm uses this property to guarantee that the optimum weights are found. Since we use this additional function f, we can guarantee only that the objective function $f(\alpha(\vec{w}), \vec{w})$ is a quasiconvex function if $f(\alpha, \vec{w})$ does not depend on $\vec{w}$. This holds because we demand $f(\alpha, \vec{w})$ to be monotone in the parameter α and the composition of a quasiconvex function with a monotone function is again a quasiconvex function, and because the maximum over the solutions of the recurrence relations is a quasiconvex function [124].

For our purposes where we are primarily interested in upper bounds on the running times of exact exponential-time algorithms, we do not care too much for the fact that the objective function $f(\alpha(\vec{w}), \vec{w})$ may not necessarily be quasiconvex. Whether the function is quasiconvex or not, we want to find good (possibly optimal for a given type of analysis) upper bounds on running times. Our algorithm appears to perform satisfactory. This may be explained by the fact that the objective functions $f(\alpha(\vec{w}), \vec{w})$ behave nicely near the optimal weights $\vec{w}$ in our analyses.

We now consider the second modification to Eppstein's algorithm. Our algorithm uses a small quadratically constrained program to find the next search direction in which it will perform a line search. That is, the computed search direction is a vector $\vec{v}$ in the unit ball with the property that its minimum inner product with any of the gradients $\nabla f(\alpha_r(\vec{w}), \vec{w})$ with $r \in R'$ is maximised. Because the objective function forces the vector $\vec{v}$ to have unit length, this approach has no bias for any direction. This in contrast to Eppstein's approach that uses linear programming to find the

vector $\vec{v}$. Because of this, the search for $\vec{v}$ can be restricted only to a convex set defined through linear constraints that approximates the unit ball, for example, the d-dimensional hypercube. Since not all vectors have the same length in such a convex set, this introduces a bias for certain directions. As a result, many more iterations can be required before the algorithm terminates.

Finally, we consider the third and last modification to Eppstein's algorithm. Our algorithm uses the description of the feasible set of weights D by inequalities to determine the next direction used for a line search if the current point $\vec{w}$ is sufficiently close to the boundary of D. This is different from Eppstein, who incorporated the feasible set D in the functions that represent the recurrence relations by setting $\alpha_r(\vec{w}) = \infty$, for all $r \in R$, if $w \notin D$. This modification allows our algorithm to smoothly follow these boundaries without getting stuck at such a boundary, or again needing many more iterations.

5.7. Concluding Remarks

In this chapter, we considered exact exponential-time algorithms for the Dominating Set problem. We used the measure-and-conquer technique that is often used to analyse exact exponential-time algorithms as a guiding tool in the design of these algorithms. More specifically, we iteratively identified the bounding cases of the numerical optimisation problems associated with a measure-and-conquer analysis, and we used these bounding cases to formulate new reduction rules to improve the algorithm and the corresponding upper bound on the running time. This resulted in a form of computer-aided algorithm design. We note that we use a similar approach to find improved branching rules for an algorithm instead of finding new reduction rules in Chapter 6 (details in Appendix A.3).

Parts of instances that correspond to 'small' bounding cases in the numerical optimisation problem associated with a measure-and-conquer analysis are generally small: they involve few sets and elements, and these sets and elements in general have small cardinality or frequency, respectively. Therefore, it may be possible to look for new reduction rules or improved branching rules in an automated way. This would result in a form of measure-and-conquer-driven automated algorithm design. It would be interesting to see if such ideas can be used to obtain faster exact-exponential time algorithms for a series of $\mathcal{NP}$-hard problems. For an example of automated design of exact exponential-time algorithms that does not involve measure and conquer, see [170].

6

Designing Algorithms for Edge-Domination Problems

After studying algorithms for vertex-domination problems such as Dominating Set and Total Dominating Set in the previous chapter, we now turn our attention to edge-domination problems. The most notable edge-domination problem is Edge Dominating Set. This problem is identical to Dominating Set in line graphs.

While both Edge Dominating Set and Dominating Set are $\mathcal{NP}$-hard [326], in some ways Edge Dominating Set is easier. For instance, Dominating Set is hard to approximate [127], while Edge Dominating Set is constant-factor approximable [69]. In parameterised algorithms, k-Dominating Set most likely is not fixed-parameter tractable as it is $\mathcal{W}[2]$-complete [116], while k-Edge Dominating Set is fixed-parameter tractable [131]. In the setting of exact exponential-time algorithms, it also seems that Edge Dominating Set is somewhat easier; the currently fastest exact algorithm for Dominating Set has a running time of $\mathcal{O}(1.4969^n)$ time (Chapter 5), while in this chapter we present an $\mathcal{O}(1.3226^n)$-time algorithm for Edge Dominating Set.

Previous results on Edge Dominating Set use the idea of enumerating minimal vertex covers in order to compute a minimum edge dominating set. In this chapter, we further investigate this approach and derive a powerful reduction rule that can be used in this setting to obtain far more effective algorithms. Our first algorithm already improves on the algorithms in the literature by using this reduction rule, but no further complicated techniques at all. The time bound for this algorithm is tightened considerably by analysing it with measure and conquer. Furthermore, we derive faster

[†]This chapter is joint work with Hans L. Bodlaender. The chapter contains results of which a preliminary version has been presented at the 3th International Workshop on Parameterized and Exact Computation (IWPEC 2008) [303]. The full version is also available as technical report UU-CS-2007-051 [301].

algorithms by using the step-by-step improvement procedure from Section 5.3.

We will also show that our ideas for Edge Dominating Set extend to Minimum Maximal Matching and Matrix Dominating Set, and with some more modifications also to weighted versions of these problems. As a consequence of our results, we also solve an open problem of Fernau in [131]. He asked whether vertex cover enumeration techniques can be used to solve the parameterised problem k-Minimum Weight Maximal Matching. We answer his question affirmatively by giving an $\mathcal{O}^*(2.4006^k)$-time algorithm for this problem.

We first introduce some notation, recall the definitions of the problems involved, and survey known results in Section 6.1. Then, in Section 6.2, we show how minimal vertex covers can be used to obtain an exact algorithm for Edge Dominating Set with a running time exponential in the number of vertices (not edges). In Section 6.3, we improve upon this algorithm by introducing a new reduction rule and a change in the branching rule of the algorithm. We then analyse this algorithm using measure and conquer in Section 6.4. In Section 6.5, we further change the branching rules of the algorithm in a step-by-step fashion and obtain an $\mathcal{O}(1.3226^n)$-time and polynomial-space algorithm. We extend our results to weighted edge-domination problems in Section 6.6. Finally, we give our efficient parameterised algorithm for k-Minimum Weight Maximal Matching in Section 6.7.

6.1. Edge-Domination Problems

We first recall the definition of Edge Dominating Set and some related problems, and then survey some previous results. A subset $D \subseteq E$ of the edges in a graph G is called an *edge dominating set* if every edge $e \in E$ is either in D or dominated by an edge f in D, where f dominates e if e and f have an end point in common.

Edge Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of size at most k?

Compare this definition to the definition of Dominating Set: a (minimum) edge dominating set is a (minimum) dominating set in the line graph $L(G)$ of G.

The first exact algorithm for Edge Dominating Set is due to Randerath and Schiermeyer in 2005 [279]. They gave an algorithm with time complexity $\mathcal{O}(1.4423^m)$ that uses polynomial space. This was improved by Raman et al. to $\mathcal{O}(1.4423^n)$ time and polynomial space [259]. Recently by Fomin et al. gave an algorithm requiring $\mathcal{O}(1.4082^n)$ time and space [138].

An interesting related problem is Minimum Maximal Matching. Recall that a matching M is maximal if there is no edge $e \in E$ such that $M \cup \{e\}$ also is a matching.

Minimum Maximal Matching
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of size at most k?

It is not hard to see that this problem is equivalent to Minimum Independent Edge

Algorithm 6.1. A simple algorithm for Edge Dominating Set.

Input: a graph $G = (V, E)$
Output: a minimum edge dominating set in G

```
1: Compute the set 𝒞 of all minimal vertex covers in G
2: for each minimal vertex covers C ∈ 𝒞 do
3:    Let C_i be the set of all isolated vertices in G[C]
4:    Compute a minimum edge cover C' in G[C\C_i]
5:    Let D_C be C' plus an extra edge for each vertex in C_i containing this vertex
6: return the D_C encountered of minimum cardinality
```

Dominating Set, where independence between edges is interpreted in terms of the line graph, i.e., the edges involved may not have any end point in common.

Another interesting related problem is Matrix Dominating Set.

Matrix Dominating Set
Input: An $n \times m$ 0-1 matrix and an integer $k \in \mathbb{N}$.
Question: Does there exist a set S of 1-entries in M of size at most k such that every 1-entry is on the same row or column as an entry in S?

As noted in [326], this problem is equivalent to Edge Dominating Set on *bipartite graphs*: any matrix M that is an instance of Matrix Dominating Set corresponds to the instance Edge Dominating Set on bipartite graphs in which there is a vertex for each row and each column, and an edge between a row vertex and a column vertex if and only if its corresponding entry in M is a 1.

6.2. Using Matchings and Minimal Vertex Covers

We start by first giving a simple exact algorithm for Edge Dominating Set. For this algorithm, we first need the notion of an edge cover. An *edge cover* is a subset $C' \subseteq E$ such that every vertex $v \in V$ is incident to an edge $e \in C'$. Because maximum matchings can be computed in polynomial time [120], a minimum cardinality edge cover in a graph G (*minimum edge covers*) can also be computed in polynomial time in the following way. First, compute a maximum matching M in G. Then, for each unmatched vertex v, add an edge incident to v to M. It is not hard to see that the resulting edge set M is a minimum edge cover.

Our first algorithm is based upon the following observation [259]; see Algorithm 6.1.

Proposition 6.1. *If $D \subseteq E$ is an edge dominating set in $G = (V, E)$, then $C = \{v \in V \mid \exists_{e \in D}\ v \in e\}$ is a vertex cover in G.*

Proof. For each $e \in E$, there is an edge $f \in D$ that dominates e, i.e., e and f have an end point in common. This endpoint belongs to C and therefore C is a vertex cover. □

Theorem 6.2. *Algorithm 6.1 solves* Edge Dominating Set *in $\mathcal{O}(1.4423^n)$ time.*

Proof. We prove that the minimum size of an edge dominating set in G equals the

minimum cardinality of D_C as computed by Algorithm 6.1 over all minimal vertex covers C in G. From this the correctness follows.

If C is a minimal vertex cover, then D_C is an edge dominating set since if any edge is not dominated then both endpoints are not in C which contradicts C being a vertex cover. Therefore, the algorithm returns an edge dominating set. To see that it is minimal, consider a minimum edge dominating set D in G. By Proposition 6.1, its endpoints form a vertex cover C in G. From this vertex cover C, a minimum edge dominating set can be reconstructed by computing a minimum edge cover in $G[C]$. The vertex cover C does not need to be minimal, but, for any minimal vertex cover $C_1 \subseteq C$, the edges incident to a vertex $v \in C \backslash C_1$ are all dominated by any choice of edges incident to the vertices in C_1. Thus, D_{C_1} constructed by Algorithm 6.1 from the minimal vertex cover C_1 dominates the same edges as D. And, it is not larger than D since the edge cover D_{C_1} needs to cover only a subset of the vertices in C. Hence, D_{C_1} is a minimum edge dominating set.

The running time is derived from the Moon-Moser bound [235, 241] on the number of maximal independent sets, and hence minimal vertex covers in G: this number is bounded by $3^{n/3} < 1.4423^n$. Enumerating all minimal vertex covers can be done with only polynomial delay [196, 221], therefore Algorithm 6.1 has a running time of $\mathcal{O}(1.4423^n)$. □

Following the notation of the proof of Property 6.1, the smallest edge dominating set which contains $C \subseteq V$ as endpoints will be denoted by D_C from now on.

By using a standard technique from [177], Theorem 6.2 also gives an algorithm for Minimum Maximal Matching.

Corollary 6.3. Minimum Maximal Matching *can be solved by a modification of Algorithm 6.1 in* $\mathcal{O}(1.4423^n)$ *time.*

Proof. Take the minimum edge dominating set computed by Algorithm 6.1. Let $\{u, v\}$, $\{v, w\}$ be a pair of dominating edges incident to the same vertex v. By minimality, there cannot be another dominating edge incident to w (remove $\{v, w\}$ for a smaller edge dominating set). Also, there must be a vertex x adjacent to w without any incident dominating edge; otherwise, the edge dominating set without $\{v, w\}$ would be a smaller edge dominating set. Hence, we can replace $\{v, w\}$ by $\{w, x\}$ obtaining an edge dominating set with one pair of not independent dominating edges less and repeating this process results in a minimum maximal matching. □

6.3. A Reduction Rule for Edge Dominating Set

The $3^{n/3}$ upper bound on the number of minimal vertex covers in a graph G is tight; consider the family of graphs consisting of l triangles: these graphs have $3l$ vertices and 3^l minimal vertex covers. However, from the perspective of computing a minimum edge dominating set, this class of graphs is trivial: just pick an edge from each triangle.

In this section, we use properties of edge dominating sets in order to enumerate fewer minimal vertex covers, avoiding situations of the type we just described. In this

way, we obtain a faster algorithm than the simple algorithm of Section 6.2. The modifications are very simple, yet powerful enough to already improve upon the algorithm by Fomin et al. [138] which uses far more complicated techniques. First, we will introduce a reduction rule, and secondly, we will introduce a more efficient branching strategy. Like Algorithm 6.1, the new algorithm enumerates a series of minimal vertex covers, and computes for each of these minimal vertex covers C the smallest edge dominating set D_C that contains the vertices C in its set of endpoints. To this end, it continuously keeps track of a partitioning of the vertices of G in three sets: a set C of vertices that must become part of the minimal vertex cover, a set I of vertices that may not become part of the minimal vertex cover (they are in the complementing maximal independent set), and a set U of vertices, which we call the set of *undecided* vertices. We denote such a state by the four-tuple (G, C, I, U).

We introduce the following rule:

Reduction Rule 6.1.

if $G[U]$ has a connected component H that is a clique **then**
  Let $\tilde{G}$ be the result of adding a vertex v connected to all vertices in H to G
  Let $\tilde{C} := C \cup H \cup \{v\}$ and $\tilde{U} := U \backslash H$
  Recursively solve the problem $(\tilde{G}, \tilde{C}, I, \tilde{U})$ and denote its solution by D
  if D contains two distinct edges $\{u, v\}, \{v, w\}$ incident to v **then**
    return $(D \backslash \{\{u, v\}, \{v, w\}\}) \cup \{\{u, w\}\}$
  return $D \backslash \{\{u, v\}\}$, where $\{u, v\}$ is the unique edge in D incident to v

Proof of correctness. After the recursive call the extra vertex v is incident to at least one edge in D, since $v \in \tilde{C}$. Also, v is incident to at most two edges in D. This is because two such edges can be replaced by the edge joining the other endpoints; this gives a smaller edge dominating set with $\tilde{C}$ as a subset of the set of endpoints.

All clique edges in the original graph are dominated if at most one clique vertex is not incident to a dominating edge. Therefore, if D contains only one edge incident to v, then removing this edge results in an edge dominating set in the original graph G with C as a subset of its set of endpoints. Because D is of minimum cardinality (in $\tilde{G}$) and the returned set is one smaller, this returned set must also be of minimum cardinality (in G). Namely, if it is not, then adding the edge between the unique vertex of the clique that is not an endpoint in the edge dominating set and v results in a smaller alternative for D.

If D contains two edges incident to v, replacing these by the edge joining the other endpoints also results in such an edge dominating set in the original graph. This edge dominating set is also of minimal cardinality because adding any edge incident to v gives an alternative for D. □

We note that a simpler rule can be used for connected components that form a clique of size one or two. Namely, isolated vertices in $G[U]$ can be put into I. Furthermore, K_2 components in $G[U]$ can be put into C if they have no neighbours in C in G, and they can be contracted otherwise, after which can we put the resulting vertex into C.

Recall that, for a vertex set U, the *U-degree* of a vertex $v \in U$ is defined to be the degree of v in $G[U]$.

Algorithm 6.2. A faster algorithm for Edge Dominating Set.

Input: a graph $G = (V, E)$, and three vertex sets C, I, and U
Initially, $C := \emptyset$, $I := \emptyset$, $U := V$
Output: a minimum edge dominating set in G

```
1: if G[U] has a connected component H that is a clique then
2:    Apply Reduction Rule 6.1
3: else if a vertex v of maximum degree in G[U] has U-degree at least three then
4:    Create two subproblems and solve each one recursively:
5:       1: (G, C ∪ N_U(v), I ∪ {v}, U\N_U[v])     2: (G, C ∪ {v}, I, U\{v})
6: else
7:    for each minimal vertex covers Ĉ on G[U] do
8:       Compute the candidate edge dominating set D_{C∪Ĉ}
9: return the smallest edge dominating set encountered
```

If Reduction Rule 6.1 does not apply, Algorithm 6.2 picks any undecided vertex $v \in U$ of maximum degree in $G[U]$ (maximum number of undecided neighbours in G). If v has U-degree at least three, we branch on this vertex, generating two subproblems. In one subproblem, v is put in the independent set I; because no neighbour of v can also be in the independent set I, these neighbours (at least three) are all put in the vertex cover C. In the other subproblem, v is put the vertex cover C. We note that this may result in the construction of vertex covers which are not minimal, but all minimal vertex covers are enumerated in this way.

If v has U-degree smaller than three, $G[U]$ is of maximum degree at most two and, due to Reduction Rule 6.1, $G[U]$ does not contain a connected component that is a clique. Therefore, $G[U]$ now consists of a collection of paths on at least three vertices and cycles on at least four vertices. In this case, Algorithm 6.2 enumerates all minimal vertex covers on these paths and cycles.

For each resulting partition of V in an independent set I and a vertex cover C, Algorithm 6.2 computes a candidate for the minimum edge dominating set D_C in the same way as Algorithm 6.1 and returns the candidate of minimum cardinality.

Theorem 6.4. *Algorithm 6.2 solves* Edge Dominating Set *in* $\mathcal{O}(1.3803^n)$ *time and polynomial space.*

Proof. Correctness of the algorithm follows directly from the proof of Theorem 6.2 and the correctness of Reduction Rule 6.1.

Let $P(l)$ be the number of maximal independent sets on a path on l vertices, and let $C(l)$ be the number of maximal independent sets on a cycle on l vertices. For each vertex in a maximal independent set I in a path, the next vertex in I must be at distance two or three; hence:

$$P(1) = 1 \quad P(2) = 2 \quad P(3) = 2 \qquad \forall_{l \geq 4} : P(l) = P(l-2) + P(l-3)$$

We can use this recurrence to inductively prove the following upper bound on $P(l)$ for all $l \geq 3$ after noticing that it holds for $l \in \{3, 4, 5\}$.

$$l \geq 3 : P(l) \leq \beta^l \quad \text{where } \beta < 1.33 \text{ is the root of } 1 = \beta^{-2} + \beta^{-3}$$

We now determine $C(l)$ for cycles on l vertices. For $l \leq 6$ $C(l)$ is determined by a simple enumeration. For $l \geq 7$, consider an arbitrary vertex v on a cycle on l vertices. If v is in a maximal independent set I, then neither of its neighbours are, leaving $P(l-3)$ possibilities. If $v \notin I$, then one or both of its neighbours is in I. Each of the cases where one neighbour is in I leaves $P(l-6)$ possibilities because by maximality of I the neighbour of the neighbour of v that is not in I must belong to I. In the case that both neighbours are in I, five vertices are fixed leaving $P(l-5)$ possibilities. Hence, we obtain:

$$C(4) = 2 \quad C(5) = 5 \quad C(6) = 5 \qquad \forall_{l \geq 7} : C(l) = P(l-3) + P(l-5) + 2P(l-6)$$

Let u be the number of undecided vertices in our problem instance (initially $u = n$), and let $S(u)$ be the number of subproblems generated to solve an instance with $|U| = u$. We have the following recurrence relation:

$$S(u) \leq \begin{cases} S(u-1) + S(u-4) & \text{branch on a vertex of } U\text{-degree at least three} \\ P(l)S(u-l) & \text{minimal vertex covers in a path on } l \text{ vertices} \\ C(l)S(u-l) & \text{minimal vertex covers in a cycle on } l \text{ vertices} \end{cases}$$

Because of the branching on a vertex of degree three, we have that $S(u) \leq \alpha^u$ where α is the solution of $1 = \alpha^{-1} + \alpha^{-4}$. For the enumeration of minimal vertex covers in paths, we have $S(u) \leq \beta^l S(u-l) \leq \beta^l \alpha^{u-l} < \alpha^u$ because $\beta < \alpha$. And, for the enumeration of minimal vertex covers in cycles, we have $S(u) < \alpha^u$. This last inequality holds because the solution to $\gamma^u = C(l)\gamma^{u-l}$ converges to $\gamma = \beta$ when $l \to \infty$ and reaches its maximum on $l \geq 4$ when $l = 5$; here $\gamma < 1.379 < \alpha < 1.3803$. We note the the convergences to $\gamma = \beta$ is due to the fact that, in the recurrence for $C(l)$, the number of maximal independent sets on a path on l vertices increases in influence when l increases. The worst case over these three possibilities gives $S(u) \leq \alpha^u$ which results in the running time of $\mathcal{O}^*(\alpha^n)$ or $\mathcal{O}(1.3803^n)$.

The collection of minimal vertex covers constructed is not being stored, the enumeration search tree is traversed, therefore the algorithm uses only polynomial space. □

We conclude this section, with the remark that we cannot improve the algorithm by putting more paths or cycles in the polynomial part of the algorithm (assuming $\mathcal{P} \neq \mathcal{NP}$). This follows from the following proposition.

Proposition 6.5. *Finding a minimum edge dominating set in a graph G with marked vertices C, that contains all marked vertices as endpoints, and where $G[V \setminus C]$ (this is $G[U]$ in our algorithms) is a collection of paths on at most three vertices is $\mathcal{NP}$-hard.*

Proof. Consider a SATISFIABILITY instance with variables $x_1, x_2, \ldots, x_n$ and clauses $c_1, c_2, \ldots, c_m$. We safely assume that all variables x_i occur at least once as a positive literal and at least once as a negative literal.

Introduce a marked vertex (a vertex in C) for each clause. We will connect this clause vertex to gadgets representing the variables; this will be done in such a way that the edge between the gadget and this clause vertex will be selected in the minimum edge dominating set if and only if the corresponding literal is set to *True* in some given satisfying assignment. Notice that because the clause vertex is marked it must be an endpoint of at least one edge in the required minimum edge dominating set.

Next, introduce a marked vertex (a vertex in C) incident to two edges for each variable. One of both edges must be selected since the variable vertex is a marked vertex, and which one is selected represents whether the variable is set to $True$ or $False$. If the variable occurs only once as a positive or only once as a negative literal, we directly connect the corresponding edge to the vertex representing the corresponding clauses. Otherwise, we let the edge be incident to the middle vertex of an unmarked path on three vertices; these are the only unmarked vertices in our construction and these are of length at most tree as claimed. Each of the two endpoints of this path will be connected to a new marked vertex v_1 and v_2. These vertices v_1 and v_2 are both incident to one more edge which other endpoint we will assign soon. Suppose that the path is connected to the $True$ edge of a variable vertex while this variable is set to $True$. In this case, the edges of the path are dominated by the selected edge of the variable vertex, and it is always optimal to pick the second edge (with so far unassigned other endpoint) of v_1 and v_2. If the current variable x_i occurs twice as a positive literal, we can now connect v_1 and v_2 to the corresponding clause vertices. Otherwise, we can add more of these path gadgets to increase the number of occurrences to any positive number. We repeat the same construction for the negative literals of the variable, and for all variables.

Since all marked vertices that are not clause vertices are non-adjacent, any minimum edge dominating set that contains all marked vertices as endpoints uses at least $|C| - m$ edges. It is not hard to see that such an minimum edge dominating set of size $|C| - m$ exists if and only if the corresponding Satisfiability instance is satisfiable.

In this $\mathcal{NP}$-hardness proof, the paths on three vertices can easily be replaced by cycles on four vertices. □

6.4. Analysis Using Measure and Conquer

When we branch on a vertex v of large degree in $G[U]$, then the removal of v from $G[U]$ in one branch, and the removal of $N_U[v]$ from $G[U]$ in the other branch, will reduce the degree of some of the remaining vertices in $G[U]$. Since we can deal with vertices of U-degree at most two (collections of paths and cycles in $G[U]$) in less time than we need for vertices of U-degree three or four, this reduction of the degrees means additional progress for the algorithm. In this section, we show how we can keep track of this additional progress by using the *measure-and-conquer* technique [142, 144]; see also Section 5.2. In combination with a slightly changed branching strategy on paths and cycles in $G[U]$, this leads to an improved time bound.

We first modify the enumeration of minimal vertex covers on paths and cycles: Algorithm 6.3 no longer enumerates all minimal vertex covers, but instead branches on the third vertex v of a path on at least four vertices and applies Reduction Rule 6.1. In one branch, v is put in the independent set resulting in the removal of four vertices: v, its neighbours, and the remaining isolated vertex. In the other branch, v is put in the vertex cover resulting in the removal of three vertices: v and the first two vertices of the path, since they now form a 2-clique in $G[U]$. Using this branching strategy on paths, we break cycles on at least five vertices by branching in two subproblems: pick any vertex v and put v in the vertex cover or put v in the independent set and its neighbours in the vertex cover. Finally, we still enumerate all minimal vertex

Algorithm 6.3. A third algorithm for Edge Dominating Set.

Input: a graph $G = (V, E)$, and three vertex sets C, I, and U
Initially, $C := \emptyset$, $I := \emptyset$, $U := V$
Output: a minimum edge dominating set in G

if $G[U]$ has a connected component H that is a clique **then**
  Apply Reduction Rule 6.1
else if a vertex v of maximum degree in $G[U]$ has U-degree at least three **or** $G[U]$ has a connected component H that is a cycle on $l \geq 5$ vertices **then**
  Pick any $v \in H$ and recursively solve the following two subproblems:
    1: $(G, C \cup N_U(v), I \cup \{v\}, U \backslash N_U[v])$ 2: $(G, C \cup \{v\}, I, U \backslash \{v\})$
else if $G[U]$ has a connected component H that is a cycle on four vertices **then**
  Let H be the cycle v_1, v_2, v_3, v_4, v_1. Recursively solve the subproblems:
    1: $(G, C \cup \{v_1, v_3\}, I \cup \{v_2, v_4\}, U \backslash \{v_1, v_2, v_3, v_4\})$
    2: $(G, C \cup \{v_2, v_4\}, I \cup \{v_1, v_3\}, U \backslash \{v_1, v_2, v_3, v_4\})$
else if $G[U]$ has a connected component H that is a path on $l \geq 4$ vertices **then**
  Let v_1, v_2, v_3, v_4 be the end of the path. Recursively solve the subproblems:
    1: $(G, C \cup \{v_2, v_4\}, I \cup \{v_1, v_3\}, U \backslash \{v_1, v_2, v_3, v_4\})$ 2: $(G, C \cup \{v_3\}, I, U \backslash \{v_3\})$
else if $G[U]$ had a connected component H that is a path on three vertices **then**
  Let v be the middle vertex of the path. Recursively solve the subproblems:
    1: $(G, C \cup N_U(v), I \cup \{v\}, U \backslash N_U[v])$ 2: $(G, C \cup \{v\}, I \cup N_U(v), U \backslash N_U[v])$
else // Now: $U = \emptyset, C \dot\cup I = V$
  Compute the candidate edge dominating set D_C
return the smallest edge dominating set encountered

covers on remaining paths on three vertices or cycles on four vertices. This results in Algorithm 6.3.

We estimate the number of subproblems generated by branching on paths and cycles:

Lemma 6.6. *For Algorithm 6.3 and $l \geq 4$:*

1. *A cycle component C_l in $G[U]$ gives rise to at most $4^{l/6}$ subproblems.*
2. *A path component P_l in $G[U]$ gives rise to at most $4^{(l-1)/6}$ subproblems.*

Proof. (1.) Let $P'(l)$, $C'(l)$ be the number of subproblems generated by Algorithm 6.3 when dealing with a path or cycle on l vertices, respectively. We derive the values of $P'(l)$ and $C'(l)$ for $l \leq 4$ directly and for $l \geq 5$ we have the following recurrence relation that follows from the branching of the algorithm:

$$P'(1) = P'(2) = C'(3) = 1 \qquad P'(3) = P'(4) = C'(4) = 2$$

$$\forall_{l \geq 5} : P'(l) = P'(l-3) + P'(l-4) \qquad C'(l) = P'(l-1) + P'(l-3)$$

Let γ be the solution of $1 = \gamma^{-3} + \gamma^{-4}$. For $l \geq 4$, $P'(l) < \gamma^l$ follows by induction after noting that it holds for $l \in \{4, 5, 6, 7\}$. For $l \geq 10$ we have:

$$C'(l) < \gamma^{l-1} + \gamma^{l-3} = \gamma^l(\gamma^{-1} + \gamma^{-3}) = \left(\gamma \sqrt[l]{\gamma^{-1} + \gamma^{-3}}\right)^l < (4^{1/6})^l$$

using the fact that $\sqrt[l]{\gamma^{-1}+\gamma^{-3}}$ is decreasing and smaller than $4^{1/6}$ if $l \geq 10$. Direct computation shows that for $l < 10$: $C'(l) \leq 4^{l/6}$.

(2.) For $l \geq 8$, $\gamma^{l/(l-1)}$ is decreasing and smaller than $4^{1/6}$, therefore:

$$P'(l) < \gamma^l = \left(\gamma^{l/(l-1)}\right)^{l-1} < (4^{1/6})^{l-1}$$

For $4 \leq l \leq 7$: $P'(l) \leq 4^{(l-1)/6}$, by direct computation. □

These estimates are tight: when Algorithm 6.3 branches on a C_6 component in $G[U]$ ($l = 6$), we indeed generate $4 = 4^{l/6}$ subproblems.

For the measure-and-conquer analysis, we need a weight function $w : \mathbb{N} \to [0, 1]$ assigning weights $w(d)$ to vertices of degree d in $G[U]$. Instead of counting the number of undecided vertices to measure the progress of our algorithm, we will now use their total weight $k := \sum_{v \in U} w(\deg_{G[U]}(v))$ as a measure. This is justified by the fact that if we can show that our algorithm runs in $\mathcal{O}(\alpha^k)$ time using weight function w, it will also run in $\mathcal{O}(\alpha^n)$ time, since for any problem instance $k \leq n$.

Theorem 6.7. *Algorithm 6.3 solves* Edge Dominating Set *in* $\mathcal{O}(1.3323^n)$ *time and polynomial space.*

Proof. Let $w : \mathbb{N} \to [0, 1]$ be the weight function assigning weight $w(\deg_{G[U]}(v))$ to vertices $v \in G[U]$. The algorithm removes all vertices of U-degree zero, therefore $w(0) = 0$. Let $\Delta w(i) = w(i) - w(i-1)$. Vertices with a larger U-degree should be given a larger weight, hence we demand: $\forall_{n \geq 1}\ \Delta w(n) \geq 0$. Furthermore we impose non-restricting *steepness inequalities*: $\forall_{n \geq 1}\ \Delta w(n) \geq \Delta w(n+1)$.

Consider an instance where the algorithm branches on a vertex v of maximum U-degree $d \geq 3$ with r_i neighbours of degree i in $G[U]$ ($d = \sum_{i=1}^{d} r_i$). If v is put in the vertex cover, it is removed from U and the U-degrees of all its neighbours in $G[U]$ are decreased by one. If v is placed in the independent set then $N_U[v]$ is removed from U, and the total sum of the degrees of the remaining vertices is reduced by at least d_2; here d_2 is a lower bound on the number of edges between $N_U[v]$ and vertices at distance two from v in $G[U]$:

$$d_2 = \left(\sum_{i=1}^{d} (i-1) r_i\right) \bmod 2 \qquad \text{except when } d = r_3 = 3 \text{ then: } \quad d_2 = 2$$

This follows from a parity argument: there must be an edge in $G[U]$ with only one endpoint in $N_U[v]$ if $1 \equiv \sum_{i=1}^{d} (i-1) r_i \pmod 2$. Also $N_U[v]$ cannot be a clique by Reduction Rule 6.1, hence if $d = r_d$ there must be at least two edges in $G[U]$ with only one endpoint in $N_U[v]$.

Altogether, we conclude that the algorithm recurses on two instances, one in which the measures decreases by Δ_{indep}, and one in which the measure decreases by Δ_{vc}, with Δ_{indep} and Δ_{vc} satisfying:

$$\Delta_{\text{indep}} \geq w(d) + \sum_{i=1}^{d} r_i w(i) + d_2 \Delta w(d) \qquad \Delta_{\text{vc}} \geq w(d) + \sum_{i=1}^{d} r_i \Delta w(i)$$

Figure 6.1. Graph corresponding to a worst case for Algorithm 6.3.

Let $S(k)$ be the number of subproblems generated to solve a problem with measure k. For all $d \geq 3$ and $(d = \sum_{i=1}^{d} r_i)$, we have a recurrence relation of the form:

$$S(k) \leq S(k - \Delta_{\text{indep}}) + S(k - \Delta_{\text{vc}})$$

We define $q(w)$ to be the functional mapping a weight function to the solution of this entire set of recurrence relations.

By Lemma 6.6, an l-cycle or l-path generates at most $4^{l/6}$, respectively $4^{(l-1)/6}$, subproblems. An l-cycle has a measure of at least $l \cdot w(2)$ and a path on l vertices has a measure of at least $(l-1) \cdot w(2)$, since $\Delta w(1) \geq \Delta w(2)$ and hence $2w(1) \geq w(2)$. Therefore, in an instance where the vertices in cycle components and path components on at least four vertices in $G[U]$ have measure k', the removal of these vertices from U by Algorithm 6.3 results in at most $4^{k'/6w(2)}$ subproblems.

We now look for the optimal weight function $w : \mathbb{N} \to [0, 1]$, satisfying the restrictions, such that the following maximum over the worst case behaviours of the different branch cases is minimum. We distinguish between the case where the maximum U-degree is three or more, the case where cycles and paths on at least four vertices are removed from $G[U]$, and the case where a path on three vertices is removed from $G[U]$.

$$S(k) \leq \left(\min_{w:\mathbb{N}\to[0,1]} \max \left\{ q(w),\ 4^{1/6w(2)}\ , 2^{1/(w(2)+2w(1))} \right\} \right)^k$$

For some large enough integer $p \geq 3$, we set set $\forall_{i \geq p} w(i) = 1$ to obtain a finite numerical optimisation problem involving the solutions to the recurrence relations for $3 \leq d \leq p+1$ and $d = \sum_{i=1}^{d} r_i$. In this finite problem, all recurrences with $d > p+1$ are dominated by those with $d = p+1$. Solving this finite problem numerically (see Section 5.6), we obtain the optimal weights and a solution $\alpha < 1.3323$:

$$w(1) = 0.750724 \qquad w(2) = 0.914953 \qquad \forall_{i \geq 3}\ w(i) = 1$$

Therefore, an instance of measure k generates less than 1.3323^k subproblems, leading to the upper bound on the running time of $\mathcal{O}(1.3323^n)$. Since we do not store any subproblems, but just traverse an enumeration tree, we use only polynomial space. □

Since measure-and-conquer analyses provide only upper bounds on the running time of an algorithm, it is useful to consider lower bounds also.

Proposition 6.8. *The worst case running time of Algorithm 6.3 is* $\Omega(1.3160^n)$.

Proof. Consider the class of graphs consisting of l disjoint copies of the graph in Figure 6.1. On each individual copy, Algorithm 6.3 can branch on the leftmost vertex resulting in two subproblems: one where this entire copy is removed from U and one where a

path of length three remains in $G[U]$. This leads to a total of three subproblems for each copy of the graph. Therefore, Algorithm 6.3 generates $3^l = 3^{n/4} > 1.3160^n$ subproblems on this class of graphs. This proves the $\Omega(1.3160^n)$ lower bound. □

6.5. Step-by-Step Improvement of the Worst Cases

As we have seen in Section 5.3, it is often a good idea to reconsider the numerical optimisation problem obtained from a measure-and-conquer analysis. The function optimised in the proof of Theorem 6.7 equals the maximum over the solutions to a series of recurrence relations: one for each subcase considered. The solution to this numerical optimisation problem is an optimal point $x \in \mathbb{R}^p$. In x, or any other feasible point in $\mathbb{R}^p$, some of the solutions to the individual recurrence relations are tight to the maximum. If one slightly varies the weights at this optimum x, the solutions to these tight recurrence relations increase (by optimality of x). If we now change our algorithm in such a way that it handles such a tight subcase in a more efficient way, the corresponding recurrence relation changes: its solution becomes smaller. In this case we can move out of x to a new optimum, with a necessarily smaller maximum over the solutions of the recurrence relations. This results in a smaller upper bound on the running time. This approach will be used in this section to improve upon Algorithm 6.3. However, since this approach requires a lot of technical case analysis, we moved large parts to Appendix A.3 and only sketch the most important ideas here.

The numerical optimisation problem associated with Algorithm 6.3 (see the proof of Theorem 6.7) gives the following tight worst cases:

1. $d = 3, r_2 = 2, r_3 = 1$, i.e., we have a vertex of maximum U-degree three, with two neighbours in $G[U]$ of U-degree two and one neighbour in $G[U]$ of U-degree three.
2. $d = 3, r_3 = 3$: we have a vertex of maximum U-degree three, with three neighbours in $G[U]$ of U-degree three.
3. a connected component in $G[U]$ is a path on three vertices.

We can improve upon the first two cases, while improving upon the third seems hard (see the remark above Proposition 6.5).

Consider the first case. In this case, v is a vertex of maximum U-degree where the algorithm branches on. It has degree three with two neighbours $u_1, u_2 \in U$ of U-degree two and one neighbour $u_3 \in U$ of U-degree three. In our analysis of Section 6.4, we had a lower bound d_2 on the number of edges between $N_U[v]$ and the vertices and distance two from v; for this case, we had $d_2 = 0$. We can now consider two subcases.

In the first subcase v, u_1, u_2 and u_3 form a connected component in $G[U]$, isomorphic to the graph in Figure 6.1. Algorithm 6.3 branches on v. We modify this now, by instead branching on one of the U-degree two vertices, e.g., u_1. In both subproblems that are obtained after branching on u_1, the vertices of the subgraph that remain in U form a clique in $G[U]$, and so are dealt with by Reduction Rule 6.1. Therefore, the entire subgraph disappears from $G[U]$ after one branching step and the application of Reduction Rule 6.1, while previously we had a path on three vertices remaining in $G[U]$ in one subproblem that required another branching step.

In the second subcase, u_1, u_2 and/or u_3 are adjacent to vertices in $U\backslash\{v, u_1, u_2, u_3\}$. If we branch on v, then these vertices will have their U-degrees reduced by one in one branch, implying a larger progress than estimated in Section 6.4: by a parity argument we can use $d_2 = 2$ as a new lower bound on the number of edges between $N_U[v]$ and the rest of $G[U]$.

Thus, we modify the algorithm and split this case in two subcases in the measure-and-conquer analysis. If we solve the resulting numerical optimisation problem (see Section 5.6), this proves an upper bound on the running time of $\mathcal{O}(1.3315^n)$ for this modified algorithm.

Arguments, similar to the argument given above for the case $d=3, r_2=2, r_3=1$, can be given in a large number of other cases as well. This leads to a series of improvement steps and a series of algorithms, where each algorithm slightly improves upon the previous one. In each improvement step, we formulate an alternative branching rule for each of the cases tight to the optimum of the numerical optimisation problem associated with the previous algorithm. This process leads to a large case analysis which we moved to Appendix A.3.

This approach leads to the following result.

Theorem 6.9. *There exists a polynomial-space algorithm that solves* Edge Dominating Set *in* $\mathcal{O}(1.3226^n)$ *time.*

Proof. See Appendix A.3. □

We note that we also derived a lower bound of $\Omega(1.2753^n)$ on the running time of the algorithm corresponding to Theorem 6.9. Since this lower bound requires the details of this algorithm, this result can also be found in Appendix A.3.

Remark 6.1. Considering more subcases and deriving more alternative branching rules could further reduce the running time of the algorithm. But, if we continue in the same fashion as in Appendix A.3, we cannot improve beyond $\mathcal{O}(1.3214^n)$. This is because, in each improvement step, we increased the lower bound d_2 on the number of edges between $N_U[v]$ and the rest of $G[U]$; if we branch on a vertex of maximum degree d, then this lower bound d_2 is bounded from above by $d_2 \leq \sum_{i=1}^{d}(i-1)r_i$. If we solve our numerical optimisation problem (see Section 5.6) using these maximum values for d_2, we obtain the running time bound of $\mathcal{O}(1.3214^n)$.

As a consequence of Theorem 6.9, we also obtain the following results (see Section 6.1 and Corollary 6.3):

Corollary 6.10. *There exists an algorithm that solves* Minimum Maximal Matching *in* $\mathcal{O}(1.3226^n)$ *time and polynomial space.*

Corollary 6.11. *There exists an algorithm that solves* Matrix Dominating Set *in* $\mathcal{O}(1.3226^{n+m})$ *time and polynomial space.*

For Matrix Dominating Set a slightly simpler algorithm would suffice since there cannot be any odd cycles in a bipartite graph; therefore, Reduction Rule 6.1 can be replaced by a reduction rule that removes isolated vertices and 2-cliques from $G[U]$.

We note that the previously fastest algorithm for Matrix Dominating Set by Fomin et al. is faster than the algorithm for Edge Dominating Set on which it was based [138]. Fomin et al. obtain this improvement by noticing that a bipartite graph contains less that $3^{n/3}$ minimal vertex covers. We cannot use this improvement here, because our approach does not use a subroutine that enumerates *all* minimal vertex covers.

6.6. Results on Weighted Edge-Domination Problems

Next, we will consider the weighted variants of Edge Dominating Set and Minimum Maximal Matching. Proposition 6.1 still applies to these weighted problems, while other properties exploited by our algorithms need more careful consideration. In this section, we introduce modifications of the algorithm of the previous section that solve these weighted problems with the same upper bound on running time. For both variants, we need a slightly different approach.

6.6.1. Minimum Weight Edge Dominating Set

We start by formally introducing the problem.

Minimum Weight Edge Dominating Set
Input: A graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{R}_+$, and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of total weight at most k?

We note that we use $\mathbb{R}_+$ to denote the set of non-negative real numbers.

Let us first look at the polynomial-time procedure that we execute at at the leaves of the search tree of the algorithm of Theorem 6.9. We note that we could also consider Algorithm 6.3 since the algorithm of Theorem 6.9 is identical to the Algorithm 6.3 except for some changes in the branching rules. In the unweighted case, it is sufficient to compute a minimum edge cover in $G[C]$, but this does not extend to the weighted case. This is because the use of edges between a vertex in the independent set I and a vertex in the vertex cover C can lead to a smaller total weight. To deal with this, we notice that the Minimum Weight Edge Cover problem is solvable in polynomial (cubic) time [252].

First consider the Minimum Weight Generalised Edge Cover problem: in a graph G cover a specified subset of the vertices $C \subseteq V$ by a set of edges of minimum total weight. This problem can also be solved in cubic time [253] in the following way [131]. Create the graph G' with vertex set $C \cup \{v\}$, where v is a new vertex. G' contains all edges from $G[C]$ and some additional edges $\{u, v\}$ between a vertex $u \in C$ and the new vertex v if the vertex u satisfies one of the following cases: $u \in C$ has degree zero in $G[C]$; or, $u \in C$ has an edge in G whose weight is smaller than the weight of any edge incident to u in $G[C]$. The weight of a new edge $\{u, v\}$ will be the minimum weight over all edges incident to u in G.

Proposition 6.12 ([131]). *Let G be a graph, let $C \subseteq V$, and let G' be the graph obtained from G and C using the above construction. Then, the minimum weight*

generalised edge cover in G has weight equal to the minimum of the weights of the minimum weight edge covers in $G[C]$ and G'.

Proof. The minimum weight generalised edge cover in G has weight equal to the minimum weight edge cover in $G[C]$ or G' depending on whether edges with endpoints in $V\backslash C$ are used. This will equal the one with smallest weight, since if no edges incident to a vertex in $V\backslash C$ are used in the minimum weight generalised edge cover in G then the minimum weight edge cover in G' will have greater weight than the one in $G[C]$ (more needs to be covered). Equivalently, if some of these edges are used, then we obtain a solution with smaller weight by using them and hence the minimum weight edge cover in $G[C]$ will have larger weight than the one in G'. □

We now consider Reduction Rule 6.1. Reduction Rule 6.1 no longer applies to the weighted case, and cannot be easily adapted to this case, as it is not possible to assign weights to the new edges it introduces such that we obtain an equivalent instance. However, in the case of cliques of size at most three, the following modified rules can be used. Note that these reduction rules are applied in a setting where the vertices in G are partitioned into tree sets: a set C of vertices that must become part of the minimal vertex cover, a set I of vertices that may not become part of the minimal vertex cover (they are in the complementing maximal independent set), and a set U of undecided vertices.

Reduction Rule 6.2. Put isolated vertices in $G[U]$ in the independent set I.

Proof of correctness. The rule is correct because all edges incident to an isolated vertex in $G[U]$ have their other endpoint in C, and hence will be dominated by an edge incident to this endpoint. □

Reduction Rule 6.3.

if $G[U]$ has a connected component H that is a clique of size two or three **then**
  Let e be an edge of minimum weight in H with weight $\omega(e)$
  Let $\tilde{G}$ be a copy of G with a new vertex v connected to all vertices in H
  Let $\tilde{C} := C \cup H \cup \{v\}$, $\tilde{U} := U\backslash H$, and let the new edges in $\tilde{G}$ have weight $\omega(e)$
  Recursively solve the problem $(\tilde{G}, \tilde{C}, I, \tilde{U})$ with resulting edge dominating set D
  if D contains two distinct edges f, g incident to v **then**
    return $(D\backslash\{f, g\}) \cup \{e\}$
  return $D\backslash\{f\}$, where f is the unique edge in D incident to v

Proof of correctness. Observe that the edges of the clique H are dominated in G if at most one vertex in H is not incident to a dominating edge. Thus, if one edge in D is incident to v, the returned set is an edge dominating set in G. If two edges $\{u, v\}, \{v, w\}$ in D are incident to v, then e is incident to u or w because H consists of no more than three vertices. Therefore, as the returned set contains e, we have that it is an edge dominating set in G also.

The returned set is of total weight $\left(\sum_{d\in D} \omega(d)\right) - \omega(e)$, and therefore it has minimum weight. This is because if there is an edge dominating set D' in G of smaller weight then we can add an edge e' with weight $\omega(e)$ to D' obtaining a minimum weight edge dominating set in $\tilde{G}$ of smaller total weight than D. Here, e' is the edge joining

the one vertex in H not incident to an edge in D' with v, or any edge incident to v if no such vertex exists. □

Notice that for 2-cliques, this is equivalent to contracting the edge and connecting the new vertex by an edge of weight equal to the contracted edge's weight to a new vertex. This new vertex does not need to be covered by the generalised edge cover.

Theorem 6.13. *There exists an algorithm that solves* Minimum Weight Edge Dominating Set *in* $\mathcal{O}(1.3226^n)$ *time and polynomial space.*

Proof. Consider Algorithm 6.3 and modify it in the following way. Replace Reduction Rule 6.1 by Reduction Rules 6.2 and 6.3. Moreover, in the leaves of the search tree, use Proposition 6.12 to compute the minimum weight edge dominating set that contains the vertices in C in its set of endpoints.

That this algorithm is correct follows in exactly the same way as in Theorem 6.4, based on Theorem 6.2, and the correctness of Reduction Rules 6.2 and 6.3.

The running time is dominated by the exponential number of subproblems generated. This is because for each partitioning of V in a minimal vertex cover and a maximal independent set in a leaf of the search tree, the algorithm computes the minimum weight edge dominating set containing the vertex cover in its set of endpoints in polynomial time. The only thing that can change the number of subproblems generated compared to the proof of Theorem 6.7 is the removal of cliques of size at least four by a reduction rule. However, the analysis in the proof of Theorem 6.7 does not require that cliques of size at least four are removed by a reduction rule. Hence, the running time of $\mathcal{O}(1.3323^n)$ follows.

We can improve the running time to $\mathcal{O}(1.3226^n)$ by using the modified branching rules introduced in the proof of Theorem 6.9 in Appendix A.3. □

6.6.2. Minimum Weight Maximal Matching

We have given $\mathcal{O}(1.3226^n)$-time algorithms for Minimum Maximal Matching and Minimum Weight Edge Dominating Set based on modifications of the algorithm of Theorem 6.9. These modifications cannot be combined to construct an algorithm for Minimum Weight Maximal Matching (Minimum Weight Independent Edge Dominating Set) since the transformation of Corollary 6.3 does not preserve edge weights. We will now give another modification of the algorithm of Theorem 6.9 that allows us to solve Minimum Weight Maximal Matching in the same time.

Let us first introduce the problem formally.

Minimum Weight Maximal Matching
Input: A graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{R}_+$ and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of total weight at most k?

Our algorithm for this problem is based on the fact that, similar to the other edge-domination problems considered, we can construct the minimum weight maximal matching containing a minimal vertex cover in its set of endpoints. To this end, we

Algorithm 6.4. Minimum Weight Generalised Independent Edge Cover.

Input: a graph $G = (V, E)$ and a subset of its vertices $C \subseteq V$
Output: a minimum weight generalised independent edge cover of C in G if one exists

1: **if** G has an odd number of vertices **then**
2: Add a new vertex v to G ($v \notin C$)
3: **for each** $v, w \in V \backslash C, v \neq w$ **do**
4: Add a new edge between v and w to G with zero weight
5: **if** there exists a minimum weight perfect matching P in G **then**
6: **return** P with all edges between vertices not in C removed
7: **return false**

consider the Minimum Weight Generalised Independent Edge Cover problem: cover a specified subset of the vertices $C \subseteq V$ in a graph G by a set of edges of minimum total weight such that no two edges are incident to the same vertex.

Proposition 6.14. *Algorithm 6.4 solves the* Minimum Weight Generalised Independent Edge Cover *problem in polynomial time.*

Proof. The returned edge set is a generalised independent edge cover of C in G since it is a matching and it contains all vertices in C in its set of endpoints.

Let G' be the graph obtained from G by adding edges and the possibly adding a vertex, as done in lines 1-4 of Algorithm 6.4. Consider any generalised independent edge cover D of C in G. We notice that we can extend D to a perfect matching P' in G' because all vertices not incident to an edge in D are adjacent as they are not in C, and there is an even number of vertices in G' This perfect matching P' has the same total weight as D since the added edges all have zero weight.

The returned generalised independent edge cover has the same weight as the computed perfect matching P in G'. Because P is of minimum total weight, and all generalised independent edge covers of C in G can be transformed into a matching of equal total weight by using the above construction, the returned set is a minimum weight generalised independent edge cover of C in G. *False* is only returned if no generalised independent edge cover of C exists in G. □

We again give a modified reduction rule suited for the new problem.

Reduction Rule 6.4.

if $G[U]$ has a connected component H that is a clique **then**
Let $\tilde{G}$ be a copy of G with a new vertex v connected to all vertices in H
Let $\tilde{C} := C \cup H$, $\tilde{I} := I \cup \{v\}$, $\tilde{U} := U \backslash H$, and let the new edge have weight zero
Recursively solve the problem $(\tilde{G}, \tilde{C}, \tilde{I}, \tilde{U})$ with resulting edge dominating set D
if D contains an edge e incident to v **then**
return $D \backslash \{e\}$
return D

Proof of correctness. In a clique, a maximum of one vertex is allowed not to be incident to a dominating edge. Since all vertices in H are put in $\tilde{C}$, and in $\tilde{C}$ at most one edge can be incident to v, the returned edge set is an independent edge dominating set. This

returned independent edge dominating set has the same total weight as D. Therefore, it is of minimal total weight: if an independent edge dominating set of smaller weight would exist, then a minimum weight maximal matching in $\tilde{G}$ with smaller weight than D can be constructed. □

Theorem 6.15. *There exists an algorithm that solves* Minimum Weight Maximal Matching *in* $\mathcal{O}(1.3226^n)$ *time and polynomial space.*

Proof. Identical to Theorem 6.13 using Proposition 6.14 and the proof of correctness of Reduction Rule 6.4. □

6.7. An $\mathcal{FPT}$–Algorithm for *k*-Minimum Weight Maximal Matching

The results on Minimum Weight Maximal Matching from Section 6.6.2 also allow us to solve an open problem raised by Fernau in [131]. In this paper, Fernau asks whether vertex covers can be exploited to obtain efficient parametrised algorithms for k-Minimum Weight Maximal Matching as well. Because Algorithm 6.4 allows us to compute the minimum weight maximal matching containing any vertex cover in its set of endpoints, and any minimum weight maximal matching must contain a minimal vertex cover in its set of endpoints since it is an edge dominating set, the answer to Fernau's question must be positive.

In this section, we will give an $\mathcal{O}^*(2.4006^k)$-time algorithm for k-Minimum Weight Maximal Matching. If we would use any of the modifications presented in this chapter, this would also give an $\mathcal{O}^*(2.4006^k)$-time algorithm for the parameterised version of the other edge-domination problems. We restrict ourselves to this problem, as this section functions only to show that it possible to obtain a parameterised algorithm in this way, not to give the currently fastest algorithms.

For the parameterised versions of variants of Edge Dominating Set, the first non-trivial algorithm is due to Prieto in her PhD thesis [256]: she gives an $\mathcal{O}^*(2^{4k(k+2)})$-time and polynomial-space algorithm for k-Minimum Maximal Matching. Fernau improved this to $\mathcal{O}^*(2.6162^k)$ time and polynomial space in [131] and extended the results to k-Edge Dominating Set and k-Minimum Weight Edge Dominating Set. Fomin et al. improved the running time at the cost of exponential space to $\mathcal{O}^*(2.4168^k)$. We will give a modification of this algorithm that uses ideas from [97] that runs in $\mathcal{O}^*(2.4006^k)$ time and exponential space. Very recently, Raible and Fernau [23] improved the running time to $\mathcal{O}^*(2.3819^k)$ and polynomial space using a parameterised version of measure and conquer.

Let us first formally introduce the parameterised problem.

***k*-Minimum Weight Maximal Matching**
Input: A graph $G = (V, E)$ and a weight function $\omega : E \to \mathbb{R}_{\geq 1}$.
Parameter: A non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of total weight at most k?

Algorithm 6.5. An algorithm for k-Minimum Weight Maximal Matching.

Input: a graph $G = (V, E)$, three vertex sets C, I, and U, and a parameter k
Initially, $C := \emptyset$, $I := \emptyset$, $U := V$

Output: a minimum weight maximal matching of weight at most k in G if one exists

if $|C| > 2k$ **or** ($\Delta(G[U]) = 3$ **and** $|\{v \mid v \in U, d_U(v) = 3\}| > 4k - 2|C|$) **then**
  return false
else if ($\Delta(G[U]) = 3$ **and** $|C| \leq 0.130444k$)
  or ($\Delta(G[U]) = 2$ **and** $|C| \leq 0.797110k$) **then**
  Construct a path decomposition of G using Lemma 6.17
  Compute a minimum weight maximal matching M in G using Proposition 6.16
  Stop the algorithm: do not backtrack!
  return M if it is of total weight at most k, or **false** otherwise
else if a vertex v of maximum degree in $G[U]$ has U-degree at least two **then**
  Create two subproblems and solve each one recursively:
    1: $(G, C \cup N_U(v), I \cup \{v\}, U \backslash N_U[v])$ 2: $(G, C \cup \{v\}, I, U \backslash \{v\})$
else // $\Delta(G[U]) \leq 1$
  Exhaustively apply Reduction Rule 6.4 // this results in: $U = \emptyset$
  Let M be a minimum weight generalised independent edge cover of (G, C)
  return M if it is of total weight at most k, or **false** otherwise

Notice that, in order to compare weights to the parameter, it is required that for every input edge e: $\omega(e) \geq 1$. Alternatively one could ask for a minimum weight maximal matching that consists of at most k edges.

Recall the definitions of pathwidth and a path decompositions; see Section 2.2.2 or for example [44, 202]. Also, recall Proposition 2.13 that shows that, given a path decomposition of a graph G of width k, we can solve Edge Dominating Set on G in $\mathcal{O}^*(3^k)$ time. The proof of this proposition can easily be adapted to obtain the following result.

Proposition 6.16. *There is an algorithm that, given a path decomposition of a graph G of width k, solves* Minimum Maximal Matching *on G in $\mathcal{O}^*(3^k)$.*

Proof. We notice that the algorithm of Proposition 2.13 computes a minimum maximal matching in G (which equals a minimum edge dominating set because of the standard replacement technique from [177], used in for example Corollary 6.3). It is not hard to see that the recurrences used for dynamic programming by the algorithm of Proposition 2.13 need only a slight modification to compute the weight of a minimum weight maximal matching instead of the number of edges in a minimum maximal matching. The result then follows. □

Now, consider Algorithm 6.5. This algorithm also uses branching and a reduction rules to enumerate relevant minimal vertex covers to solve k-Minimum Weight Maximal Matching. Different to before, this algorithm sometimes switches to a dynamic programming approach on path decompositions, and it has two rules based on which it decides that in some branches no solution can be found.

Before going into the details, we first need the following lemma which is a combinations of simple results from [97, 138, 147].

Lemma 6.17. *Let $\epsilon > 0$. There exist a constant c_ϵ such that we can bound the pathwidth $pw(G)$ of G by the following quantities in any node of the search tree of Algorithm 6.5:*

1. *If $\Delta(G[U]) \leq 2$, then $pw(G) \leq |C| + 2$.*
2. *If $\Delta(G[U]) \leq 3$, then $pw(G) \leq (\frac{1}{6} + \epsilon)x + |C| + c_\epsilon$, where $x = |\{v \mid v \in U, d_U(v) = 3\}|$.*

Path decompositions of the appropriated width can be found in polynomial time.

Proof. Let C, I, U be the partitioning of the vertices of G in a node of the search tree. For both cases, we will first show that we can obtain path decompositions of $G[U]$ which width equals the above formulas where the term $|C|$ is omitted.

(1.) If $G[U]$ has maximum degree two, it is a collection of paths and cycles. It is easy to construct a path decomposition of width at most two for such a graph in polynomial time. For a path, let the ith bag of the path decomposition contain exactly the two endpoint of the ith edge on the path (starting from any side). For a cycle, remove one vertex that we put in every bag of the path decomposition, and then treat the remaining vertices as a path.

(2.) If $G[U]$ has maximum degree three, then $x \leq 4k$ because the algorithm will not consider the subproblem otherwise. In this case, Theorem 2.14 tells us that there exists a c_ϵ such that we can construct a path decomposition of G of width at most $(\frac{1}{6} + \epsilon)x + c_\epsilon$ in polynomial time.

Because I is an independent set, and non of the neighbours of the vertices in I are in U, the pathwidth of $G[U \cup I]$ equals the pathwidth of $G[U]$. Then, the claimed results follow by adding the vertices in C to every bag of the path decomposition. □

Now, we are ready to give the main result of this section.

Theorem 6.18. *Algorithm 6.5 solves k-*Minimum Weight Maximal Matching *in $\mathcal{O}^*(2.4006^k)$ time and space.*

Proof. Correctness is trivial if a path decomposition is constructed by the algorithm: it then ignores any branching done and outputs a minimum weight maximal matching of G if it is of small enough weight. If no path decomposition is constructed, then the algorithm enumerates minimal vertex covers through branching similar to Algorithm 6.2 whose correctness follows from Theorem 6.4. The only difference is that Algorithm 6.5 discards certain branches: we will now show that these branches will never lead to a vertex cover of size at most $2k$ and hence never lead to a solution of size at most k. Correctness of the algorithm then follows.

In the case that the algorithm finds that $|C| > 2k$, then this is trivially true. In the other case, $G[U]$ is of maximum degree three and $x > 4k - 2|C|$ where x is the number of vertices of degree three in $G[U]$. $G[U]$ contains at least $1\frac{1}{2}x$ edges (three edges incident to at most two degree three vertices), and each vertex in $G[U]$ can cover at most three of them as the maximum degree in $G[U]$ is three. Hence, we can only find a minimal vertex cover of size at most $2k$ if $3(2k - |C|) \geq 1\frac{1}{2}x$. Dividing both sides by $1\frac{1}{2}$ shows that this is not the case when $x > 4k - 2|C|$.

For the running time, let $S(k)$ be the number of subproblems generated to solve a problem with parameter k, and let $\alpha = 0.398555$, $\beta = 0.065222$. We use $\mu = k - \frac{1}{2}|C|$

as a measure of progress for the algorithm which is justified by the fact that the algorithm stops if $|C| > 2k$: initially $\mu = k$. If we branch on a vertex of U-degree at least four, the behaviour of the algorithm corresponds to the recurrence relation $S(\mu) \leq S(\mu - \frac{1}{2}) + S(\mu - 2)$. This holds because $|C|$ increases by one in one branch and by at least four in the other. Solving the recurrence relation leads to a running time of this part of the algorithm of $\mathcal{O}^*(1.9052^{\mu})$.

Now suppose that during the execution of the algorithm a path decomposition of width p is computed while $\Delta(G[U]) = 3$. This happens when the maximum degree in $G[U]$ first drops to three, $|C| \leq 2\beta k$, and $x \leq 4k - 2|C| \leq 4k$. Then, using some small enough $\epsilon > 0$, a minimum weight maximal matching in G is computed in $\mathcal{O}^*(3^{|C|+(\frac{1}{6}+\epsilon)x}) \leq \mathcal{O}^*(3^{(2\beta+\frac{2}{3}+4\epsilon)k}) = \mathcal{O}^*(2.4006^k)$ time. Note that the factors in the running time that depends on ϵ disappear in the rounding of the bases of the exponent when using some small enough $\epsilon > 0$. This leads to a total running time of $\mathcal{O}^*(1.9052^k + 2.4006^k) = \mathcal{O}^*(2.4006^k)$.

If we branch on a vertex of U-degree at least three, then $|C| > 2\beta k$ in every branch in which the maximum degree in $G[U]$ first becomes three as $2\beta = 0.130444$. The behaviour of the algorithm corresponds to the recurrence relation $S(\mu) \leq S(\mu - \frac{1}{2}) + S(\mu - 1\frac{1}{2})$ ($|C|$ increases by one or three) which leads to a running time of this part of the algorithm of $\mathcal{O}(2.1480^{\mu})$. In the first branching steps until $|C| > 2\beta k$, the algorithm generates at most $1.9052^{\beta k}$ subproblems with large enough C such that μ is at most $k - \beta k$. Hereafter, it branches on other vertices solving these subproblems in time at most $\mathcal{O}^*(2.1480^{k-\beta k})$. This leads to a total running time of $\mathcal{O}^*(1.9052^{\beta k}2.1480^{k-\beta k}) = \mathcal{O}^*(2.1312^k)$.

Now, suppose that during the execution of the algorithm a path decomposition of width p is computed while $\Delta(G[U]) = 2$. This happens when the maximum degree in $G[U]$ first becomes at most two and $|C| \leq 2\alpha k$ as $2\alpha = 0.797110$. Then, a minimum weight maximal matching in G is computed in $\mathcal{O}^*(3^{|C|}) \leq \mathcal{O}^*(3^{2\alpha k}) \leq \mathcal{O}^*(2.4006^k)$ time. This leads to a total running time of $\mathcal{O}^*(2.1312^k + 2.4006^k) = \mathcal{O}^*(2.4006^k)$.

Finally, if no path decomposition is computed, we have that $|C| > 2\alpha k$ in every branch in which the maximum degree in $G[U]$ first becomes at most two. Hereafter, the algorithm performs a series of branchings on U-degree two vertices according to the recurrence relation $S(\mu) \leq S(\mu - \frac{1}{2}) + S(\mu - 1)$ ($|C|$ increases by one or two). This recurrence relation solves to $\mathcal{O}^*(2.6180^{\mu})$. When the maximum degree in $G[U]$ becomes one, the minimum weight maximal matching for this branch is computed in polynomial time. The first branching steps until $|C| > 2\alpha k$ as executed such that at most $1.9052^{\beta k}2.1480^{\alpha k-\beta k}$ subproblems are generated. Hereafter, it branches on vertices of U-degree two solving these subproblems in time at most $\mathcal{O}^*(2.6180^{k-\alpha k})$. Together, this leads to a total running time of $\mathcal{O}^*(1.9052^{\beta k}2.1480^{\alpha k-\beta k}2.6180^{k-\alpha k}) = \mathcal{O}^*(2.4006^k)$. □

We note that, in the above proof, α and β are chosen in such a way that:

$$1.9052^{\beta k}2.1480^{\alpha k-\beta k}2.6180^{k-\alpha k} = 3^{2\alpha k} = 3^{(2\beta+\frac{2}{3})k}$$

This balances the three different components in the running time of the algorithm.

6.8. Concluding Remarks

In this chapter, we have presented $\mathcal{O}(1.3226^n)$-time and polynomial-space algorithms for Edge Dominating Set, Minimum Weight Edge Dominating Set, Minimum Maximal Matching (Minimum Independent Edge Dominating Set), and Minimum Weight Maximal Matching. These algorithms are obtained by using a vertex cover structure on the input graph. Furthermore, the iterative improvement of a measure and conquer analysis is used to improve the branching rules of the algorithm. We also applied a variant of our algorithms to the parameterised problem k-Minimum Weight Maximal Matching.

There are many more edge-domination problems. For example, one can define a general class of edge-domination problems in a similar way as the $[\rho, \sigma]$-domination problem (see Section 1.6) are defined for vertex-domination problem. It would be interesting to see which of these problems, or other edge-domination problems, can be solved in time exponential in the number of vertices n as well, maybe using similar techniques.

7

Exact Exponential-Time Algorithms for Independent Set in Sparse Graphs

Independent Set is one of the most intensively studied problems in the field of exact exponential-time algorithms. In 1972 Tarjan gave an algorithm solving the problem in $\mathcal{O}(1.2852^n)$ time and polynomial space [291]. The first improvement of this algorithm is due to Tarjan and Trojanowski [292], who improved the running time in 1977 to $\mathcal{O}(1.2600^n)$ while still using polynomial space. This bound has been improved often since; see Table 7.1.

In this chapter, we will give a faster exact exponential-time algorithm for Independent Set restricted to graphs in which each connected component has average degree at most three. Notice that this includes graphs of maximum degree three. For algorithms for Independent Set, this is a natural class of graphs to consider. Namely, it is well known that Independent Set is linear-time solvable on graphs of maximum degree two, while the problem is $\mathcal{NP}$-complete on graphs of maximum degree [163]. Also, Johnson and Szegedy have shown that subexponential-time algorithms for Independent Set on general graphs exist if and only if they exist for the problem on graphs of maximum degree three [197]. Therefore, Independent Set on graphs of maximum degree three most likely requires exponential time to solve, as under the *Exponential-Time Hypothesis* the general problem requires exponential time [189]. To summarise, on graphs of maximum degree d, the problem seems to

[†]This chapter is joint work with Nicolas Bourgeois, Bruno Escoffier, and Vangelis Th. Paschos. The chapter contains results which have been accepted for publication in Algorithmica [59]. Preliminary versions have been presented at the 7th Annual Conference on Theory and Applications of Models of Computation (TAMC 2010) [60] and the 12th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2010) [58]. A preliminary version of some of the results is also available as technical report: Cahier du LAMSADE 277 [57].

Authors		Polynomial space	Exponential space
Tarjan	[291]	$\mathcal{O}(1.2852^n)$	
Tarjan and Trojanowski	[292]	$\mathcal{O}(1.2600^n)$	
Jian	[195]	$\mathcal{O}(1.2346^n)$	
Robson	[270]	$\mathcal{O}(1.2278^n)$	$\mathcal{O}(1.2109^n)$
Robson	[271]	$\mathcal{O}(1.2025^n)$	$\mathcal{O}(1.1889^n)$
Fomin, Grandoni, Kratsch	[144]	$\mathcal{O}(1.2202^n)$	
Kneis, Langer, Rossmanith	[203]	$\mathcal{O}(1.2132^n)$	
This chapter		$\mathcal{O}(1.2114^n)$	

Table 7.1. Exact exponential-time algorithms for Independent Set.

require exponential-time algorithms if $d \geq 3$: this makes the problem on graphs of maximum degree three an interesting problem on its own.

Another reason to study this problem is that current algorithms for Independent Set on general graphs often rely on techniques that allow the generally faster results on Independent Set restricted to sparse graphs to be used to prove faster running times on general graphs [59, 144, 203]. An example of such a technique is measure and conquer [144], see also Section 5.2.

This chapter is organised as follows. We first recall some basic definitions and survey known results in Section 7.1. Then, we give the faster algorithm for Independent Set on graphs in which each connected component has average degree at most three in Section 7.2. Some of the proofs in this section can be found in Appendix A.4. In Section 7.3, we state some of the results that we have obtained on general graphs and on graphs of larger average or maximum degree. The proofs of these results can be found in [57]. Finally, we give some concluding remarks in Section 7.4

7.1. Independent Set

We first recall the definition of Independent Set and then survey some previous results on the problem both on general graphs and on graphs of maximum degree (or average degree at most) three. A subset $I \subseteq V$ of the vertices in a graph G is an *independent set* if no two vertices from I are adjacent in G.

Independent Set
Input: A graph $G = (V, E)$, and an integer $k \in \mathbb{N}$.
Question: Does there exist an independent set $I \subseteq V$ in G of size at least k?

This problem is a classical $\mathcal{NP}$-complete problem [162, 199], and there exists no subexponential-time algorithm for this problem unless the Exponential-Time Hypothesis fails [189], as we already discussed in the introduction to this chapter.

An overview of exact exponential-time algorithms for Independent Set on general graphs can be found in Table 7.1. We note that the $\mathcal{O}(1.2202^n)$-time algorithm of Fomin et al. [144], the $\mathcal{O}(1.2132^n)$-time algorithm by Kneis et al. [203], and our own

Authors		Running time
Chen, Kanj, Jia	[72]	$\mathcal{O}(1.1740^n)$
Beigel	[15]	$\mathcal{O}(1.1259^n)$
Chen, Liu, Jia	[75]	$\mathcal{O}(1.1504^n)$
Chen, Kanj, Xia	[73]	$\mathcal{O}(1.1255^n)$
Fomin and Høie	[147]	$\mathcal{O}(1.1225^n)$
Kojevnikov and Kulikov	[210]	$\mathcal{O}(1.1225^n)$
Fürer†	[158]	$\mathcal{O}(1.1120^n)$
Razgon	[260]	$\mathcal{O}(1.1034^n)$
Bourgeois, Escoffier, Paschos†	[55]	$\mathcal{O}(1.0977^n)$
Xiao	[322]	$\mathcal{O}(1.0919^n)$
Razgon	[261]	$\mathcal{O}(1.0892^n)$
Xiao	[325]	$\mathcal{O}(1.0885^n)$
This chapter†		$\mathcal{O}(1.0854^n)$

† These results are on the broader class of graphs where each connected component has average degree at most three.

Table 7.2. Previous result on Independent Set on graphs of maximum degree three.

result are all obtained after the faster results that Robson claims[1] [271]. We also note that many details of the algorithm by Kneis et al. [203] can be found in [246, 263].

For an overview of exact exponential-time algorithms for Independent Set on graphs of maximum degree three, see Table 7.2. Apparently many authors have been working on this problem at the same time. At the time when we obtained our $\mathcal{O}(1.08537^n)$-time algorithm for Independent Set on graphs in which each connected component has average degree at most three, the previously fastest algorithm for this problem (and the problem on graphs of maximum degree three) was the algorithm of Bourgeois et al. running in time $\mathcal{O}(1.0977^n)$.

7.2. The Algorithm for Connected Graphs of Average Degree at Most Three

In this section, we will give the main result of this chapter: an $\mathcal{O}(1.08537^n)$-time and polynomial-space algorithm for Independent Set on connected graphs of average degree at most three. As this algorithm can be applied to each separate connected component in a graph, this gives an $\mathcal{O}(1.08537^n)$-time and polynomial-space algorithm on graphs in which each connected component has average degree at most three. In Section 7.3, this algorithm will be used to obtain faster algorithms on general graphs and on graphs of larger average or maximum degree.

Our algorithm is a branch-and-reduce algorithm and has the following form. First, it applies a series of well-known reduction rules that simplify the instance. Secondly,

[1]The results from [271] have not been independently verified since its appearance in 2001, and thus are not generally accepted. Although, we were unable to verify the correctness of these results, we believe that this technical report contains ideas that are powerful enough to obtain significantly faster algorithms than, for example, the algorithms in [144, 203].

it follows the approach of Fürer [158] and looks for vertex separators of size one or two in the graph, which it uses to further simplify the instance. Thirdly, if the graph is of maximum degree four, then it exploits any separators that consist of the closed neighbourhood of a single degree-three vertex that separate a tree from the rest of the graph. Finally, the algorithm looks for a suitable structure in the graph and branches on it: it generates a series of subproblems that are solved recursively and from which the largest solution is returned.

Different from earlier chapters, we will analyse this algorithm using a measure based on *average degrees*. The measure that we use is $m - n$: the number of edges m minus the number of vertices n in the graph; this is similar to [55, 158]. The resulting upper bound on the running time of $\mathcal{O}(\alpha^{m-n})$ implies an $\mathcal{O}(\alpha^{0.5n})$-time algorithm on connected graphs of average degree at most three. We require the input graph to be connected (or the average degree requirement to apply to each connected component) because trees have measure $m - n = -1$: these trees are removed by the reduction rules and thus can increase $m - n$ to over $0.5n$ for the remaining graph, while not simplifying the connected components in this remaining graph.

The rest of this section is divided into two parts. We first introduce all the reduction rules of our algorithm in Section 7.2.1. Then, we give four lemmas that describe the branching rules of our algorithm in Section 7.2.2. We will prove only two of these lemmas here; the proofs of the other two lemmas are based on an extensive case analysis and are deferred to Appendix A.4.

7.2.1. Simple Reduction Rules and Small Separators

We will now give the reduction rules used by our algorithm. First, we give some simple well-known reduction rules. Then, we state a reduction rule based on small separators due to Fürer [158], whose details can be found in Appendix A.4.3 for completeness. Finally, we will a new reduction rule for dealing with situations in graphs of maximum degree four where the neighbourhood of a single degree-three vertex separates a tree from the rest of the graph.

The following well-known reduction rules are used by our algorithm. Although these are thoroughly described in many publications, for example in [55, 144, 158], we will prove their correctness below.

1. *Degree 0, 1:* Take any vertices of degree zero or one in the independent set; in case of a degree-one vertex, remove its neighbour.
2. *Connected components:* If G is disconnected, solve each connected component separately and return the union of the maximum independent sets over all connected components.
3. *Domination:* If the closed neighbourhood of a vertex $u \in V$ is contained in the closed neighbourhood of a vertex $v \in V$, $N[u] \subseteq N[v]$, then we say that u *dominates* v, and we remove v from the graph.
4. *Vertex folding:* If there exists a vertex $v \in V$ of degree two with non-adjacent neighbours u, w, the algorithm removes v, merges u and w to a single vertex, and adds one to the size of the maximum independent set.

Lemma 7.1. *The above four reduction rules for* Independent Set *are correct and remove any vertices of degree at most two.*

Proof. (1.) A vertex v of degree zero is in any maximum independent set, because any independent set that does not contain v can be turned into a larger independent set by adding v. The same holds for a vertex v of degree one, unless its unique neighbour u is in the maximum independent set. In this case, we can replace u by v to obtain an independent set of the same size that contains v.

(2.) If the maximum independent set I in a graph G is not the union of maximum independent sets of the connected components of G, then there is an independent set I' in some connected component $G[C]$ of G that is larger than $I \cap C$. However, the set I is not a maximum independent in this case since we can construct a larger independent set in G by replacing the vertices in $I \cap C$ by I'.

(3.) In any maximum independent set containing v, we can replace v by u without increasing the size of the independent set because the neighbours of u are a subset of the neighbours of v. Hence, there always exists a maximum independent set that does not contain v.

(4.) We can require that either v is in the independent set I, or both neighbours of v are in I. This holds because because taking v in I is an alternative of the same size to taking only one neighbour of v. The choice between these two options is equivalent to either taking the merged vertex in the maximum independent set, which correspond to taking both neighbours, or discarding it, which corresponds to taking v. The difference in the number of vertices taken in I remains one after merging the vertices, and the size of the set remains the same as we artificially add one to the size of the maximum independent set when we perform this operation.

These reduction rules remove any vertices of degree at most two because Rule 1 applies to vertices of degree zero or one, Rule 3 applies to vertices of degree two with adjacent neighbours, and Rule 4 applies to vertices of degree two with non-adjacent neighbours. □

If the given reduction rules do not apply, the graph is of minimum degree three. The algorithm then uses the following reduction rule due to Fürer [158] that exploits vertex separators of size at most two. We give the details of this *small separators* reduction rule in Appendix A.4.3 for completeness.

Lemma 7.2. *If a graph G contains a vertex separator S of size one or two, then we can simplify the instance in the following way. We recursively solve two or four subproblems that correspond to the smallest associated connected component C with each possible combination of vertices from S added. Given the solutions computed by the recursive calls, we can remove S and C from G and adjust the remaining graph to obtain an equivalent instance. If C has size at most some constant c, then this operation can be done in polynomial time.*

We note that using this small separators rule is always beneficial to the running time, also when the smallest associated connected component does not have constant size. However, we do not go into these details as we need the rule only on constant-size associated connected components to prove our claimed running time.

We are now ready to give our new reduction rule that, in a graph of maximum degree four, looks at local configurations in which the closed neighbourhood of a vertex of degree three separates a tree from the rest of the graph. We call this rule the *tree separators rule*.

Lemma 7.3. *If, after application of the reduction rules of Lemmas 7.1 and 7.2, the graph G is of maximum degree four, and G contains a vertex v of degree three whose closed neighbourhood separates a tree T from the rest of G, then we can replace this local configuration with a smaller equivalent one in polynomial time.*

Proof. First, notice that since none of the reduction rules of Lemmas 7.1 and 7.2 are applicable, G is of minimum degree three and has no separators of size at most two.

Let a, b, and c be the neighbours of v. Notice that they all have at least one edge incident to the tree T because otherwise there would exist a small vertex separator. Also notice that, because of the same reason, a, b, and c all have at least one edge not incident to v or to the tree T. Hence, because G is of maximum degree four, there are at least three and at most six edges between $N(v)$ and T, and there exists at most one edge in $G[N(v)]$.

First, consider the case where there exists an edge in $N(v)$ and hence $|T| \leq 2$. In this case, the maximum independent set in $G[N[v] \cup T]$ is of size two as we will see below. Because of this, we can safely pick v and one vertex from T in the maximum independent set I: these are the vertices that, when taken in I, pose no restrictions on which vertices we can still take in I in the remaining graph. That this holds is easy to see if T is a single vertex, namely taking v or T in I forbids taking a, b and c, and because of the edge in $N(v)$ it is not possible to take all tree vertices of $N(v)$. Otherwise, if $|T| = 2$ and without loss of generality a is connected to both vertices in T, then we cannot take three vertices if we take v because this forbids taking a, b and c and there is an edge in T. Also, we cannot take three vertices if we take a because this forbids taking v and T while there is an edge between b and c. The remaining vertices form a four cycle, and thus I can have at most two vertices from $N(v) \cup T$ from which correctness follows.

Secondly, if there is no edge in $N(v)$ and $|T| \leq 2$, then we merge $N(v) \cup T$ to a single vertex and add two to the size of I. This is similar to vertex folding. Namely, the only maximum independent set in $G[N[v] \cup T]$ equals $N(v)$, while if we do not take these three vertices we can safely pick the non-restricting size two option consisting of v and one vertex from T. Merging the local structure to a single vertex postpones this choice. This is clearly correct if $|T| = 1$ since taking v or T in I forbids taking any vertex in $N(v)$. Also, if $|T| = 2$, then taking v in I again forbids taking any vertex from $N(v)$, and taking any vertex from T in I forbids taking anything except for v and one of its neighbours again not allowing three vertices to be picked.

Thirdly, we consider the case where $|T| = 3$; see Figure 7.1. In this case, we can safely pick v and a maximum independent set in $G[T]$. This is correct because of the following reasoning. At least two neighbours of v, say a and b, have two neighbours

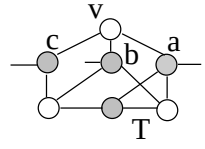


Figure 7.1. T consists of three vertices and at least two neighbours of v have two neighbours in T. In this case, we take the white vertices in I.

in T. If we would have taken a in I, then we would forbid all vertices in $N(v) \cup T$ except for b, c and one vertex in T. By adjacencies, we can take only two of these vertices making our initial choice a safe alternative. The same argument holds when taking b in I. With a and b discarded, it is easy to see that picking the degree-one vertex v and the maximum independent set in T in the maximum independent set is optimal.

What remains is the case $|T| = 4$. In this case, we can also safely pick v and a maximum independent set in $G[T]$ by similar reasoning. If T has three leaves, then there is only one way to pick four vertices from $G[N[v] \cup T]$ in I: v and the three leaves of T. This does not restrict any choices in the rest of the graph and hence is optimal. Otherwise, T is a four vertex path and all local configurations of $G[N[v] \cup T]$ have a maximum independent set of size three. Again, picking v and a maximum independent set in $G[T]$ is a safe and optimal choice. □

Before considering the branching of our algorithm, we look at the effect of each of the reduction rules on the $m - n$ measure. We notice that this measure is invariant under the vertex folding rule: this rule removes as much edges as it removes vertices. The same holds for the degree one rule unless the neighbour of the removed degree-one vertex is of degree one also. It is not so hard to see that, after the application of the degree zero and one rules and the vertex folding rule, each application of the domination rule, the small separators rule, or the tree separator rule decreases the measure $m - n$ by at least two. This is because at least one vertex of degree three is removed in each of these cases.

There are only two cases in which the measure can increase. This is when the degree two rule is applied, and when the degree one rule is applied to a degree-one vertex with a degree-one neighbour. In these cases, the measure increases by one. Since a connected component in the graph that is a tree will be reduced to one of these two cases by the degree one and two rules, these rules causes the measure to increase by one for every such tree.

We note that when we apply the reduction rules to an input graph before the first application of a branching rule, then it is not a problem that the measure can increase. This is because we require that each connected component in the input graph has average degree at most three: this causes the measure of each remaining connected component to be at most $0.5n$. The fact that the measure can increase becomes problematic when a tree is separated from the rest of the graph after applying a branching rule of the algorithm. In this case, we have to be careful not to remove too much measure in the analysis. Therefore, we will be very careful when removing a set of vertices that can separate a tree from the rest of the graph in the analysis of our algorithm. For this purpose, the tree separators rule is a useful tool. Namely, when the graph is of maximum degree four and we remove the closed neighbourhood of a degree-three vertex, then no trees can be separated since otherwise the tree separators rule could have been applied. We note that the branching rules of the next section, in most cases, precisely remove such a closed neighbourhood of a degree-three vertex from the graph; this makes the tree separators rule a useful tool.

7.2.2. The Branching Rules of the Algorithm

Having introduced the reduction rules, we proceed to the branching rules of the algorithm. These branching rules are described in the proofs of the four lemmas below: Lemmas 7.5-7.8. We will give the proofs of two of these lemmas in this section. The proofs of the other two lemmas are based on extensive case analyses and can be found in Appendices A.4.1 and A.4.2.

Let us start by giving the main idea as to why our algorithm is faster than previous algorithms for Independent Set on graphs of maximum degree three or average degree at most three. On maximum degree three graphs, most previous branching algorithms first branch on vertices that are in a cycle of length three or four, as one easily observes that this leads to a relatively small number of generated subproblems. Thereafter, these algorithms give a long subcase analysis to prove that sufficiently efficient branchings exist for graphs without such cycles.

We use these cycles not only to restrict the worst case to small-cycle-free graphs, but to improve the branching on these graphs as well. As we use reduction rules for low-degree vertices, the only maximum degree three graphs that our algorithm considers are 3-regular graphs. If no small cycles exist and we are forced to perform a relatively inefficient branching on a vertex v in such a 3-regular graph, then vertices near v get degree two in the subproblems that are generated. These vertices are folded, resulting in new vertices of degree at least four. If new small cycles emerge by this folding, they must contain the new higher degree vertices: this combination allows for very good branching, counterbalancing the inefficient branching we started with. In the other case in which no new small cycles emerge, the new higher degree vertices can exists in only a relatively small number local configurations. We prove that we can use the new higher degree vertices in these configurations such that branchings can be obtained that counterbalances the initial inefficient branching even more.

One of the basis principles that we use to obtain good branchings on graphs with small cycles is to exploit mirrors in the graph; this was also done in [144, 203].

Definition 7.4. A vertex $m \in V$ is a *mirror* of $v \in V$ if $G[N(v) \setminus N(m)]$ is a clique.

Note that this definition is equivalent to requiring that every pair of two non-adjacent vertices in $N(v)$ contains at least one vertex from $N(m)$. Whenever the algorithm branches on v and discards it, it can discard all its mirrors as well. This holds because of the following argument. Consider any independent set containing only one neighbour u of v. This independent set can be transformed into another independent set of equal size by removing u and adding v. Because such an independent set is already considered in the branch where we take v in I, we can ignore independent sets containing only one neighbour of v in the branch where we discard v. We conclude that a mirror of v cannot be in I when we must take at least two neighbours of v in I by Definition 7.4.

We will now give the branching rules of our branch-and-reduce algorithm. These branching rules are applied only when none of the reduction rules can be applied. In most cases, the branching rules generate two subproblems based on taking some selected vertex v in the maximum independent set I that is being constructed, or not. In one branch, the vertex v is taken in I and hence it is removed together with its neighbourhood. In the other branch, the vertex v is decided not to be in I and hence

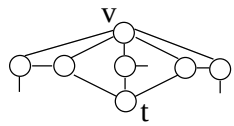


Figure 7.2. A vertex of degree five in which the removal of its neighbourhood leads to the removal of 13 edges and separates a tree.

it is removed (discarded). In the resulting analysis, we let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$.

Our first branching lemma deals with graphs that are not of maximum degree four.

Lemma 7.5. *Let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$. If G has a vertex of degree at least five, then we can branch such that $T(k)$ satisfies $T(k) \leq T(k-4) + T(k-7)$, or a better sequence of branchings exists.*

Proof. Let v be the vertex of degree at least five and assume that we branch on it. If v is discarded, one vertex is removed and at least five edges are removed giving $T(k-4)$ to the recurrence. If v is taken in I, $N[v]$ is removed. All vertices in $N(v)$ have at least one neighbour outside of $N[v]$ because of the domination rule. In the worst case, $N(v)$ consists only of degree-three vertices, hence there are at most two edges in $G[N(v)]$ and at least six edges between $N[v]$ and the rest of G; see also Figure 7.2. If no trees are created, this sums to 13 edges and 6 vertices giving $T(k-7)$ to the recurrence. We note that with six edges between $N[v]$ and the rest of G at most one tree can be separated because each tree requires at least three of these edges, and at least three of these edges must remain to make sure that there does not exist a vertex separator of size at most two.

In the case that there are more than six edges between $N[v]$ and the rest of G because either a vertex in $N(v)$ has degree four or more, or there are fewer edges in $G[N(v)]$, then these extra removed edges compensate for the possible trees that can be separated, still giving $T(k-7)$ to the recurrence, or even better.

Remains is the special case in Figure 7.2 where the minimum number of 13 edges is removed and a separate tree is created. This tree will be a single degree-three vertex t since otherwise there are sufficiently many edges between $N(v)$ and T such that there exists a separator in $N(v)$ of size at most two. In this special case depicted in Figure 7.2, v is a mirror of t. We now branch on t instead of v. Taking t in I leads to the removal of 4 vertices and 9 edges: $T(k-5)$. Discarding t and its mirror v leads to the removal of 8 edges and 2 vertices: $T(k-6)$. This branching with $T(k) \leq T(k-5) + T(k-6)$ leads to a smaller branching number than the required $T(k) \leq T(k-4)+T(k-7)$. □

The second branching lemma deals with graphs that have a vertex of degree four. Here, extra attention is given to small cycles.

Lemma 7.6. *Let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$. If G is of maximum degree four but not 3-regular, then we can branch such that $T(k)$ satisfies the following recurrence relations, or a better sequence of branchings exists.*

1. *if G has a degree-four vertex that is part of a 3- or 4-cycle also containing at least one degree-three vertex, and there are no 3- or 4-cycles containing only degree-three vertices, then $T(k) \leq T(k-5)+T(k-6)$ or $T(k) \leq 2T(k-8)+2T(k-12)$.*
2. *if G has a degree-four vertex that is part of a 3- or 4-cycle also containing at least one degree-three vertex, and the constraint on the degree-three vertices from the previous case does not apply, then $T(k) \leq T(k-4)+T(k-6)$ or $T(k) \leq 2T(k-8)+2T(k-12)$.*
3. *if the above cases do not apply, then $T(k) \leq T(k-3)+T(k-7)$.*

Proof. The proof can be found in Appendix A.4.2. The proof is based on an extensive case analysis of the local structures involved. In essence, we try to exploit any 3- or 4-cycle available, similarly to the proof of Lemma 7.7 below, and we try to exploit the degree-four vertices (that give a larger reduction in the measure) as much as possible, similarly to branching on degree-five vertices in the proof of Lemma 7.5. □

The third branching lemma deals with 3-regular graphs that contain 3- or 4-cycles. In this case, we find efficient branchings without looking at many cases.

Lemma 7.7. *Let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$. If G is 3-regular and contains a 3- or 4-cycle, then we can branch such that $T(k)$ satisfies $T(k) \leq T(k-4)+T(k-5)$, or a better sequence of branchings exists.*

Proof. Let a, b, c form a 3-cycle in G. Assume that one of these vertices, say a, has a third neighbour v that is not part of a 3-cycle. The algorithm branches on v. In one branch, v is taken in I and 9 edges and 4 vertices are removed: $T(k-5)$. In the other branch, v is discarded and by domination a is taken in I resulting in the removal of 8 edges and 4 vertices: $T(k-4)$. No trees can be separated in both branches because of the tree separators rule.

This covers the 3-cycles, unless all third neighbours of a, b and c are in a 3-cycle also. Observe that all three third neighbours are in different 3-cycles because otherwise there would exist a vertex separator of size at most two. In this case, we branch on a. In the branch where a is discarded, domination results in its third neighbour to be taken in I giving $T(k-4)$ as before. In the other branch, a is taken in I, and by domination the third neighbours of b and c are taken in I. This removes their corresponding 3-cycles completely, removing a total number of 10 vertices. Depending on how many edges exist between the corresponding 3-cycles, at least 16 edges are removed also. We notice that a tree can be separated from G, but we still have $T(k-5)$ or better when this happens.

Finally, suppose that G is 3-cycle free and let v be a vertex on a 4-cycle. Any vertex opposite to c on a 4-cycle is a mirror of v. We branch on v. In one branch, we take v in I and 3-cycle freeness results in the removal of 9 edges and 4 vertices: $T(k-5)$. In the other, we discard v and all its mirrors. This results in the removal of 6 edges and 2 vertices if v has only one mirror and possibly more if v has two or three mirrors: $T(k-4)$. Again trees can be separated from G, but this can happen only if v has three mirrors. There is only one local configuration representing this case in which 12 edges and 7 vertices are removed, which is more than enough. □

The last branching lemma deals with 3-regular graphs that are 3- and 4-cycle free. In these graphs, we have to perform a less efficient branching. However, we can use the very efficient branching of Lemma 7.6 in both generated subproblems to counterbalance this effect and obtain the claimed running time.

Lemma 7.8. *Let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$. If G is 3-regular and 3- and 4-cycle free, then we can branch such that $T(k)$ satisfies $T(k) \leq T_1(k-2) + T_3(k-5)$, or a better sequence of branchings exists. Here, we denote by $T_1(k)$ and $T_3(k)$ the recurrences from Cases 1 and 3 from Lemma 7.6 applied to an instance of measure k, respectively.*

Proof. The proof can be found in Appendix A.4.1. The proof goes along the following lines. In both subproblems generated by the less efficient $(2, 5)$ branching, degree-two vertices are created that are folded creating new vertices of degree at least four. This allows for more efficient branchings to counterbalance the less efficient branching by using these higher degree vertices. If these higher degree vertices now lie on newly created 3- or 4-cycles, this allows the application of Lemma 7.6. Otherwise, these higher degree vertices exist in only a relatively small number local configurations. Finding the claimed sequences of branchings for these local configurations requires quite some case analysis. □

Taking all four Lemmas together proves the following result.

Theorem 7.9. *There exists an algorithm solving* Independent Set *on connected graphs of average degree at most three in $\mathcal{O}(1.08537^n)$ time and polynomial space.*

Proof. If we look at all the recurrence relations that correspond to the behaviour of the algorithm as given in Lemmas 7.5-7.8, we find that the recurrence relation corresponding to the worst case of all branchings is $T(k) \leq T_1(k-2) + T_3(k-5) \leq 2T(k-2-8) + 2T(k-2-12) + T(k-5-3) + T(k-5-7)$. Here, T_1 and T_3 have the same meaning as defined in Lemma 7.8: the recurrence relation is formed by combining Lemmas 7.6 and 7.8. We note that the solution to the recurrence $T_1(k) \leq T(k-5)+T(k-6)$ is larger than the solution of $T_1(k) \leq 2T(k-8)+2T(k-12)$, however, when composed with the recurrence corresponding to the branching of Lemma 7.8, the latter leads to a larger solution. The given recurrence relation gives us a running time of $\mathcal{O}(\tau(8, 10, 10, 12, 14, 14)^k) = \mathcal{O}(1.17802^k)$, which equals $\mathcal{O}(1.17802^{0.5n}) = \mathcal{O}(1.08537^n)$ on graphs of average degree at most three. □

As separate connected components can be solved independently, this algorithm solves Independent Set on graphs in which each connected component has average degree at most three within the same running time.

Corollary 7.10. *There exists an algorithm solving* Independent Set *on graphs of maximum degree three in $\mathcal{O}(1.08537^n)$ time and polynomial space.*

Proof. In a graph of maximum degree three, each connected component has average degree at most three. □

Another result that can directly be obtained from Theorem 7.9 comes from the field of parameterised complexity. In this field, the k-Vertex Cover problem is a

benchmark. For the special case where we strict this parameterised problem to graphs of maximum degree three, we obtain the following result.

Corollary 7.11. *There exists an algorithm solving k-*Vertex Cover *on graphs of maximum degree three in $\mathcal{O}^*(1.1781^k)$ time and polynomial space.*

Proof. The k-Vertex Cover problem restricted to graphs of maximum degree three admits a kernel of size $2k$ [73]. This allows us to transform the problem in polynomial time into an equivalent instance G on $n \leq 2k$ vertices. Because the complement of a vertex cover is an independent set, we can use the algorithm of Corollary 7.10 to test whether G has an independent set of size at least $n - k$ which is equivalent to G having a vertex cover of size at most k. This is done in $\mathcal{O}^*(1.08537^{2k}) = \mathcal{O}^*(1.1781^k)$ time and polynomial space. □

This improves the $\mathcal{O}^*(1.1940^k)$ algorithm by Chen et al. [73] (see also [72]) and the $\mathcal{O}^*(1.1864^k)$ algorithm of Razgon [261]. We note that this result has very recently been improved by Xiao to $\mathcal{O}^*(1.1616^k)$ [324].

7.3. Applications to Graphs of Larger Average Degree

The algorithm of the previous section can be used to obtain faster algorithms for Independent Set in graphs of larger average (or maximum) degree also. We have done this in [58, 59] by using our own bottom-up method based on the average degree of a graph [58, 59], and by using measure and conquer [144] (see also Section 5.2). In this section, we will state the obtained results without giving the details.

We start by considering Independent Set on graphs of average degree d, with $3 < d \leq 4$. For d at most $3\frac{3}{7}$, $3\frac{3}{5}$, or 4, we used our *bottom-up method* [58, 59] to obtain faster algorithms for Independent Set on graphs in which each connected component has average degree at most d. We have obtained the following results [59].

Proposition 7.12 ([59]). *There exists polynomial-space algorithms for* Independent Set *on graphs of average degree at most d, for different values of d with the following running times:*

d	$\leq 3\frac{3}{7}$	$\leq 3\frac{3}{5}$	≤ 4
running time	$\mathcal{O}(1.1243^n)$	$\mathcal{O}(1.1396^n)$	$\mathcal{O}(1.1571^n)$

To obtain our result on general graphs and graphs of larger maximum degree, we combine this algorithm with the algorithm for Independent Set by Fomin et al. [144]. This is an $\mathcal{O}(1.2202^n)$-time algorithm whose running time is based on a measure-and-conquer analysis. The combined algorithm is a branch-and-reduce algorithm that has the following form: if the maximum degree in the graph is larger than four, then apply the reduction rules and branching rule of the algorithm by Fomin et al.; otherwise, apply the algorithm for graphs of maximum degree four of Proposition 7.12. If we analyse this combined algorithm using measure and conquer in exactly the same way as in [144], then we obtain the following result; for more details see [59].

Theorem 7.13 ([59]). *There exists an algorithm that solves* Independent Set *in $\mathcal{O}(1.2114^n)$ time and polynomial space.*

In the measure-and-conquer analysis associated with Theorem 7.13, we used a weight function $w : \mathbb{N} \to \mathbb{R}$ giving a vertex of degree d weight $w(d)$ (similar to [144]). Because the total weight of a graph of maximum degree Δ is at most $w(\Delta)n$, we also obtain fast algorithms on graphs of maximum degree five and six.

Proposition 7.14 ([59]). *There exists an $\mathcal{O}(1.1859^n)$-time and polynomial-space algorithms for* Independent Set *on graphs of maximum degree five, and an $\mathcal{O}(1.2050^n)$-time and polynomial-space algorithms for* Independent Set *on graphs of maximum degree six.*

7.4. Concluding Remarks

In this chapter, we have given a faster algorithm for Independent Set restricted to graphs in which each connected component has average degree at most three. As a result, we have obtained faster polynomial-space algorithms for Independent Set on graphs of maximum degree three, four, five, and six. We have also obtained a fast algorithm for the problem on general graphs; the only faster known algorithm is the algorithm claimed by Robson [271].

We have compared the approaches used in the more recent algorithms for Independent Set in Table 7.3. As one can see, our algorithm uses a very different approach when compared to the algorithm by Robson, and has some similarities to the other two algorithms. However, we should note that one of the ideas that is repeatedly used in the case analysis of our algorithm (Appendix A.4) is based on Robson's algorithm. This is the idea to continuously use the fact that when a vertex is discarded, then at least two of its neighbours must be selected; this follow by the replacement argument the can be found under Definition 7.4.

It should be possible to combine the approaches using much faster algorithms on sparse graphs and using a measure-and-conquer, or average-degree-based, analysis, with the approach of Robson that continuously propagates the fact that if a vertex is discarded at least two of its neighbours must be taken in the independent set. Furthermore, it should be possible to combine this approach with memorisation, even though reduction rules such as the vertex folding rule cause subproblems to no longer be induced subgraphs of the original input instance.

property	algorithm			
	Robson [271]	Fomin et al. [144]	Kneis et al. [203]	Our result [59]
running time (poly. space)	$\mathcal{O}(1.2025^n)$	$\mathcal{O}(1.2202^n)$	$\mathcal{O}(1.2132)$	$\mathcal{O}(1.2114^n)$
running time (exp. space)	$\mathcal{O}(1.1889^n)$			
fast small-degree algorithm			X	X
measure-and-conquer analysis		X	X	X
computer-verified case analysis	X		X	
partly computer generated	X			
memorisation (exp. space)	X			

Table 7.3. A comparison of recent algorithms for Independent Set.

However, such a combined approach would most probably result in a much more complex, partly computer-generated algorithm whose case analysis will probably also be mostly computer verified. To create such an algorithm, new ideas need to be developed as to when an algorithm with such an analysis is considered to be correct: how do we verify that all cases are treated correctly, and how do we verify that no cases are missing in the analysis? This may be done by computer verification. We note that initial steps in this direction are already present in the work by Kneis et al. [203]; see also the technical reports [246, 263].

III

Inclusion/Exclusion-Based Branching Algorithms

8

Inclusion/Exclusion Branching for Counting Dominating Sets

One could argue that the two most important recent new developments in the field of exponential-time algorithms are the use of *inclusion/exclusion* by Björklund et al. [27, 33] (see also Section 2.3) and *measure and conquer* by Fomin et al. [144] (see also Section 5.2). We have already seen many recent applications of measure and conquer in Chapters 5-7. On inclusion/exclusion, we note that, although the use of this technique was introduced to the field of exact exponential-time algorithms on the Travelling Salesman Problem by Kohn et al. [207], Karp [200], and Bax [14] much earlier, only recently Björklund et al. popularised it by using it to solve many covering and partitioning problems [27, 33].

In this chapter, we will show that both approaches can be used in a single algorithm. Similar to Bax [14], we observe that inclusion/exclusion can be interpreted as branching. In this way, we can combine traditional branching (as in Chapters 4-7) with inclusion/exclusion-based branching, and we can analyse such an algorithm using measure and conquer. We will use this approach to obtain two algorithms that count set covers. Given a set cover instance $(\mathcal{S}, \mathcal{U})$, these algorithms count the number of set covers of each size κ with $0 \leq \kappa \leq |\mathcal{S}|$. These algorithms are then used to obtain a series of results. The results include faster polynomial-space and exponential-space algorithms for #Dominating Set (*counting minimum dominating sets*), a faster polynomial space algorithm for Domatic Number, and faster exponential space algorithms for

[†]This chapter is joint work with Jesper Nederlof and Thomas C. van Dijk. This research started as joint work in Nederlof's master's thesis [243] supervised by Hans L. Bodlaender, INF/SCR-2007-084. The results have changed much since that time. The chapter contains results for which a preliminary version has been presented at the 17th Annual European Symposium on Algorithms (ESA 2009) [307] and results that have been presented at the 7th International Conference on Algorithms and Complexity (CIAC 2010) [300]. A preliminary version of the results published at ESA 2009 is also available as technical report UU-CS-2008-043 [306].

Dominating Set restricted to some graph classes (see also [165]). We also give faster algorithms for Minimum Weight Dominating Set under the restriction that the number of possible weight sums is bounded by a polynomial in n.

We will introduce the concept of inclusion/exclusion-based branching in Section 8.1. Then, we use inclusion/exclusion-based branching combined with traditional branching and treewidth-based techniques to obtain a polynomial-space algorithm for counting set covers in Section 8.2. In Section 8.3, we use this algorithm to obtain an $\mathcal{O}(1.5673^n)$-time and polynomial-space algorithm for #Dominating Set. The same algorithm is used as a subroutine in Section 8.4 to obtain an $\mathcal{O}(2.7139^n)$-time and polynomial-space algorithm for Domatic Number. Hereafter, we consider using exponential space and give the second algorithm for counting set covers in Section 8.5. Here, we also give an $\mathcal{O}(1.5002^n)$-time and exponential-space algorithm for #Dominating Set. Finally, we use this algorithm to obtain faster exponential-space algorithms for Dominating Set restricted to some graph classes in Section 8.6.

8.1. Inclusion/Exclusion-Based Branching

We will begin by showing that one can look at inclusion/exclusion from a branching perspective; see also [14]. In this way, we can use inclusion/exclusion-based branching to branch on an element in a Set Cover instance in the same way as one would normally branch on a set.

The canonical branching rule for Set Cover is branching on a set, as used in the algorithms in Chapter 5. Sets are *optional* in a solution: either a set is in a solution, or it is not. If we branch on this optional property, we obtain two branches in which the problem is simplified. If we *discard* the set, we decrease the number of sets. If we *take* the set, we decrease the number of sets, and, since this set covers all its elements, its elements can also be removed from the instance, decreasing the number of elements. A minimum set cover of the instance is either returned by the discard branch or returned by the take branch with the set we branched on added to it.

The counting problem can also be handled by branching steps of this type because the total number of solutions is the sum of the number of solutions obtained from each branch. We can do this because sets are *optional* in a solution. Branching on a set can be denoted as adding up the number of solutions where it is *required* to take the set and the number of solutions where it is *forbidden* to take the set:

$$\text{OPTIONAL} = \text{REQUIRED} + \text{FORBIDDEN}$$

If we are counting set covers of size κ, and we branch to take a set (that is, in the required branch), then we should count set covers of size $\kappa - 1$ in that branch. In the forbidden branch, we do not need to decrease κ.

We now consider branching on an element. Such a branching step is unusual, and may appear strange at first sight, as elements are not optional; elements represent a requirement: the requirement to cover the element. Inspired by inclusion/exclusion techniques and because we count the number of solutions, we can, however, rearrange the above formula to give:

$$\text{REQUIRED} = \text{OPTIONAL} - \text{FORBIDDEN}$$

That is, the number of ways to cover a certain element is equal to the number of ways to optionally cover it, i.e., in which we are indifferent about covering it, minus the number of ways in which it is forbidden to cover it. This is interesting because this branching rule also simplifies the instance in both branches. If we choose to make it *optional* to cover a certain element, we can remove the element from the instance: this removes an element and also reduces the size of the sets in which it occurs. If we choose to make the element *forbidden*, then we have to remove the element and every set in which the element occurs. This is an even greater reduction in the size of the instance. We have not selected a set to be in the cover in both branches, so in both branches we are looking for set covers of size κ.

Consider a branching algorithm without reduction rules and without employing branch and bound. If the branching rule is based on an optional property of the problem, as is typically the case, the algorithm is an *exhaustive search algorithm*. A similar concept exists for an algorithm in which branching is based on a required property, which we call *inclusion/exclusion-based branching* or simply *IE-branching*: without reduction rules, this is an *inclusion/exclusion algorithm*; see also Section 2.3.

To see this, consider a Set Cover instance $(\mathcal{S}, \mathcal{U})$. Let c'_κ be the number of set covers of cardinality κ, and let $a(X)$ be the number of sets in $\mathcal{S}$ that do not include any element of X. Consider the branching tree (search tree) after exhaustively applying inclusion/exclusion-based branching. In each subproblem in a leaf of this tree, each element is either optional or forbidden. We look at the contribution of a leaf, where X is the set of forbidden elements in this leaf, to the total number computed in the root of the tree. Notice that the $2^{|\mathcal{U}|}$ leaves represent the subsets $X \subseteq \mathcal{U}$. A minus sign is added for each time we have entered a forbidden branch, so the contribution of this leaf will be $(-1)^{|X|}$ times $\binom{a(X)}{\kappa}$. This last number equals the number of ways to pick κ sets from the $a(X)$ available sets, i.e., the number of set covers of cardinality κ where it is optional to cover each element not in X and forbidden to cover an element in X. All together, this gives us the following expression for c'_κ:

$$c'_\kappa = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} \binom{a(X)}{\kappa}$$

Compare this to the following expression for c_κ given by Björklund et al. [33] (see also Section 2.3):

$$c_\kappa = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a(X)^\kappa$$

These expressions are identical except for the fact that the formula of Björklund et al. counts the number of set covers c_κ where they allow a single set to be picked multiple times while we do not allow this.

The advantage of inclusion/exclusion-based branching over using the inclusion/exclusion formula is that we can use reduction rules to improve upon the standard running time of $\mathcal{O}^*(2^{|\mathcal{U}|})$. A standard method of obtaining similar improvements is *trimming*, where one predicts which summands in the inclusion/exclusion formula are non-zero in order to be able to skip the other summands. This method found applications for Travelling Salesman Problem and Graph Colouring in graphs of bounded degree [29, 32]. In our setting, trimming is equivalent to IE-branching in combination

with a halting rule (see Section 2.1) that returns zero when it can be predicted that all summands enumerated from the current branch are zero. The main advantage of our approach over trimming is that inclusion/exclusion-based branching can easily be used interchangeably with traditional branching rules, and that standard methods of analysing such algorithm, such as measure and conquer, can be applied directly.

We conclude this introductionary section by comparing the effects of both branching rules on the incidence graph of Set Cover instances.

Definition 8.1 (Incidence Graph). The *incidence graph* of a Set Cover instance $(\mathcal{S}, \mathcal{U})$, is the graph $G = (V, E)$ with a vertex for each $S \in \mathcal{S}$ and each $e \in \mathcal{U}$ and an edge between a vertex representing a set S and a vertex representing an element e if and only if $e \in S$, i.e., $V = (\mathcal{S} \cup \mathcal{U})$ and $\{e, S\} \in E$ if and only if $e \in S$.

Taking a set results in exactly the same operation on the graph as making an element forbidden, only the former is on a set vertex while the latter is on a element vertex; namely, the vertex and its neighbours are removed from the incidence graph. The same relation holds between discarding a set and making an element optional; in this case, the selected vertex is removed. The effects of both operations are symmetric to each other on the incidence graph. However, this symmetry is not complete since for other purposes the set and element vertices are not equivalent. That is, element vertices must be dominated by set vertices. This difference leads, for example, to different reduction rules for vertices of different types in Section 8.5.

Note that when considering the incidence graph of a Set Cover instance, then Set Cover becomes equivalent to Red-Blue Dominating Set where the set vertices of the instance are considered red vertices and the element vertices are considered blue vertices.

8.2. A Polynomial Space Algorithm for Counting Set Covers

In this section, we give an exponential-time and polynomial-space algorithm for counting the number of set covers of each size κ, $0 \leq \kappa \leq |\mathcal{S}|$, of a Set Cover instance $(\mathcal{S}, \mathcal{U})$ where $\mathcal{S}$ is a multiset over the universe $\mathcal{U}$. This algorithm combines the traditional branching approach that branches on sets (used in for example Chapter 5) with the inclusion/exclusion-based branching approach introduced in Section 8.1 that branches on elements. On sparse instances, the algorithm will switch to a treewidth-based dynamic programming approach. This algorithm will be used to prove the main results of Sections 8.3 and 8.4.

The combination of branching with treewidth-based dynamic programming to solve sparse instances has been used before. Examples include results by Kneis et al. for Maximum Cut, Maximum 2-Satisfiability, Maximum Exact 2-Satisfiability and Dominating Set on cubic graphs [204], and by Fomin et al. for k-Edge Dominating Set (see also Section 6.7) and to count dominating sets [138]. In this section, we introduce a combination of branching with a novel treewidth-based approach that, in contrast to the work by Fomin et al. [138], requires only polynomial space while being much simpler than the approach of Kneis et al. [204].

Algorithm 8.1. An algorithm for counting the number of set covers of each size κ.

Input: the incidence graph $I = (\mathcal{S} \cup \mathcal{U}, E)$ of $(\mathcal{S}, \mathcal{U})$ and a set of annotated vertices A
Output: a list containing the number of set covers of $(\mathcal{S}, \mathcal{U})$ of each cardinality κ
CSC(I, A):
1: **if** there exists a vertex $v \in (\mathcal{S} \cup \mathcal{U}) \setminus A$ of degree at most one in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ **then**
2: **return** CSC$(I, A \cup \{v\})$
3: **else if** there exist two vertices $v_1, v_2 \in (\mathcal{S} \cup \mathcal{U}) \setminus A$ both of degree two in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ that have the same two neighbours **then**
4: **return** CSC$(I, A \cup \{v_1\})$
5: **else**
6: Let $s \in \mathcal{S} \setminus A$ be a vertex such that $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s)$ is maximal
7: Let $e \in \mathcal{U} \setminus A$ be a vertex such that $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)$ is maximal
8: **if** $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s) \leq 2$ **and** $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e) \leq 2$ **then**
9: **return** CSC-DP(I)
10: **else if** $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s) > d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)$ **then**
11: Let $L_{\text{take}} = \text{CSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus N[s]], A \setminus N(s))$ and increase cardinalities by one
12: Let $L_{\text{discard}} = \text{CSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{s\}], A)$
13: **return** $L_{\text{take}} + L_{\text{discard}}$
14: **else**
15: Let $L_{\text{optional}} = \text{CSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{e\}], A)$
16: Let $L_{\text{forbidden}} = \text{CSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus N[e]], A \setminus N(e))$
17: **return** $L_{\text{optional}} - L_{\text{forbidden}}$

Our approach is based on an annotation procedure that allows us to deal with sparse instances in polynomial space with reasonable efficiency. This procedure acts as a set of reduction rules while it does not remove anything from the instance: it just annotates parts of it. When the unannotated part of the instance is simple enough, we remove the annotations. Because of the specific way used to annotate vertices, we can prove that the resulting instance has treewidth at most two, i.e., it is a generalised series-parallel graph. On such graphs, the problem can be solved in polynomial time.

We will begin by describing our polynomial-space algorithm for counting set covers: Algorithm 8.1. This algorithm considers the incidence graph I of the Set Cover instance $(\mathcal{S}, \mathcal{U})$. It uses a set of annotated vertices A which is initially empty. Intuitively, annotating a vertex corresponds to ignoring the vertex when selecting a vertex to branch on, not only by not considering it for branching, but also by ignoring it as a neighbour for vertices that can be branched on. Annotated vertices, however, are not ignored when the algorithm branches: here they are treated as ordinary vertices. During the execution of the algorithm, Algorithm 8.1 annotates any vertex of degree at most one in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$. Furthermore, it annotates a degree two vertex in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ if there exists another degree two vertex in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ with the same two neighbours.

If no vertex can be annotated, Algorithm 8.1 selects an element vertex and a set vertex that are both of maximum degree among the vertices of their type in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$. If the degree in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ of any vertex is at most two, then the algorithm switches to a different approach and solves the problem by dynamic programming by calling the procedure CSC-DP(I). This procedure generates a list containing the number of

set covers of $(\mathcal{S}, \mathcal{U})$ of each cardinality κ, $0 \leq \kappa \leq n$, in polynomial time and will be described later. For now, it suffices to say that the annotation procedure guarantees that any incidence graph I on which CSC-DP(I) is called is simple enough to be dealt with in polynomial time. Otherwise, the maximum degree in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ is at least three. In this case, Algorithm 8.1 branches on a vertex of maximum degree in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ while it prefers an element vertex if vertices of maximum degree of both types exist.

If Algorithm 8.1 decides to branch on a set vertex $s \in \mathcal{S}$, then it considers two subproblems: one in which it takes the set corresponding to s in the set cover, and one in which it discards it. We have seen in Section 8.1 that the number of set covers of cardinality κ corresponds to the sum of the numbers computed in both branches: it is the sum of the number of set covers containing the set corresponding to s and the number of set covers that do not contain this set. In the first subproblem, s can be removed together with all vertices corresponding to elements contained in the set corresponding to s since they are now covered; this corresponds to removing $N[s]$ from I. In the second subproblem, only s can be removed. The algorithm computes the required values by a component-wise summation of the lists returned from both branches. Notice, that this works only if we increase the cardinalities in the first branch by one because we have taken a set in the set cover: this is done by the algorithm accordingly.

If Algorithm 8.1 decides to branch on an element vertex $e \in \mathcal{U}$, then we perform inclusion/exclusion-based branching; see Section 8.1. The number of sets covers of $(\mathcal{S}, \mathcal{U})$ equals the number of set covers of the subproblem in which we remove e, i.e. the total number of covers that either cover the element or not (*optional*), minus the number of sets covers of the subproblem in which we remove e and all sets containing it, i.e., the the number of covers that do not cover the element corresponding to e (*forbidden*). In one branch, e is removed. In the other branch, e and all vertices representing sets containing the element represented by e are removed, i.e., $N[e]$ is removed. For each cardinality κ, the algorithm subtracts the result from the second branch from the result of the first branch, correctly computing the number of set covers of each size κ. Again, this is done by component-wise operations on the lists returned from both branches.

This concludes the description of the branching of the algorithm. Since the annotation procedure does not affect the output of the Algorithm 8.1, we can conclude that the algorithm is correct based on the correctness of the branching procedure.

We will now further explain the function of the annotation procedure and give the details on the procedure CSC-DP(I). To this end, we need the following well-known fact on the treewidth of a graph, e.g., see [39].

Proposition 8.2. *Let G be a graph of treewidth at least two. The following operations do not increase the treewidth of G:*

- *duplicating an edge,*
- *adding a vertex of degree one,*
- *subdividing an edge,*
- *contracting an edge incident to a vertex of degree two.*

Using this proposition, we can prove the following lemma on the structure of the incidence graph I when the algorithm uses the procedure CSC-DP(I).

Lemma 8.3. *When Algorithm 8.1 makes a call to CSC-DP(I), then the treewidth of I is bounded by two.*

Proof. Let A be the set of annotated vertices maintained by Algorithm 8.1 when calling CSC-DP(I). At this point, all vertices in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ are of degree two vertices because it is of maximum degree two and any vertex of degree at most one would have been annotated. Hence, $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ has treewidth at most two since it is a (possibly empty) collection of cycles (also see the proof of Lemma 6.17).

We remove the annotations from the remaining vertices of I in reverse order of their moment of annotation. By doing so, each step consists of either adding a vertex of degree at most one to $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$, or adding a degree two vertex v to $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ for which there exist another degree two vertex $u \notin A$ with $N(u) = N(v)$ in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$. We notice that the operation of adding the degree two vertex is identical to first contracting an edge incident to v, then doubling the contracted edge, and then subdividing both copies of the edge again. Hence, both operations do not increase the treewidth of $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ above two by Proposition 8.2. We conclude that I has treewidth at most two. □

Alternatively, one could say that when Algorithm 8.1 calls CSC-DP(I), then the incidence graph I is a generalised series-parallel graph.

Corollary 8.4. *The procedure CSC-DP(I) in Algorithm 8.1 can be implemented in polynomial time.*

Proof. Standard dynamic programming on tree decompositions or generalised series-parallel graphs. For an idea of how such an algorithm works see Chapter 11 or [44]. □

We will use Algorithm 8.1 to both count the number of dominating sets in a graph and as a subroutine to compute the domatic number. In both cases, the input to Algorithm 8.1 is different compared to the number of vertices in the graph of a problem instance. Therefore, we postpone the running-time analysis to Sections 8.3 and 8.4.

8.3. Counting Dominating Sets in Polynomial Space

In this section, we give the first application of Algorithm 8.1: we use it to compute the number of dominating sets of each size κ with $0 \leq \kappa \leq n$. We also give the currently fastest polynomial-space algorithms for #Dominating Set and Minimum Weight Dominating Set for the case where the set of possible weight sums is polynomially bounded. We note that the annotation procedure used in the algorithm plays an important role in the running-time analyses of these algorithms.

We will start by introducing the #Dominating Set problem formally.

#Dominating Set
Input: A graph $G = (V, E)$.
Question: How many dominating sets $D \subseteq V$ in G exist of minimum size?

We note that although this problem concerns the counting of *minimum dominating sets*, we consider the more general problem of counting the number of dominating sets of each size κ with $0 \leq \kappa \leq n$.

Theorem 8.5. *There exists an algorithm that counts the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a graph G in $\mathcal{O}(1.5673^n)$ time and polynomial space.*

Proof. We use Algorithm 8.1 to count the number of set covers of $(\mathcal{S}, \mathcal{U})$ of each cardinality κ where $\mathcal{S} = \{N[v] \mid v \in V\}$ and $\mathcal{U} = V$. Recall that these numbers correspond to the number of dominating sets of each cardinality κ in G; see Section 5.1.1.

For the running-time analysis, we use *measure and conquer* [144] (see Section 5.2). To this end, we use a variant of the measure used in Section 5.2 (and in [144]) on the size of a subproblem. We introduce weight functions $v, w : \mathbb{N} \to \mathbb{R}_+$ and use the following measure k on a subproblem (I, A) with $I = (\mathcal{S} \cup \mathcal{U}, E)$:

$$k := \sum_{e \in \mathcal{U}, e \notin A} v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) + \sum_{s \in \mathcal{S}, s \notin A} w(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s))$$

Here, $d_X(v)$ denotes, for a vertex set $X \subseteq V$, the degree of a vertex $v \in X$ in the induced subgraph $G[X]$.

We also define $\Delta v(i) = v(i) - v(i-1)$, $\Delta w(i) = w(i) - w(i-1)$, and impose the following constraints on the weights functions v, w:

1. $v(0) = v(1) = 0$
2. $\Delta v(i) \geq 0$ for all $i \geq 2$
3. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 2$
4. $2\Delta v(3) \leq v(2)$
5. $w(0) = w(1) = 0$
6. $\Delta w(i) \geq 0$ for all $i \geq 2$
7. $\Delta w(i) \geq \Delta w(i+1)$ for all $i \geq 2$
8. $2\Delta w(4) \leq w(2)$

Notice that annotating a vertex reduces the measure, and that annotated vertices have zero measure. Constraints 1 and 5 represent the fact that if a vertex gets degree at most one in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$, then it is annotated; hence, we can give it zero measure. Constraints 2 and 6 represent the fact that we want vertices with a higher degree to contribute more to the measure of an instance. Furthermore, Constraints 3 and 7 are non-restricting steepness inequalities that make the formulation of the problem easier, and the function of Constraints 4 and 8 is explained later.

We will now formulate a series of recurrence relations representing the branching of the algorithm. Consider branching on a vertex $s \in \mathcal{S}$ representing a set in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ and with r_i neighbours of degree i in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$.

In the branch where we take the set corresponding to s in the set cover, we remove s decreasing the measure by $w(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s))$, we remove all its neighbours decreasing the measure by $\sum_{i=2}^{\infty} r_i v(i)$, and we reduce the degrees of all vertices at distance two from s. If $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s) \geq 4$, we can bound the decrease in measure due to this last reduction by $\Delta w(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s)) \sum_{i=2}^{\infty} r_i(i-1)$ because of Constraint 7 and the fact that s is of maximum degree. Notice that if we reduce another set vertex to degree at most one in this way, then we do not remove too much measure because of Constraints 7 and 8. If $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s) = 3$, then the situation is different. Because Algorithm 8.1 prefers to branch on elements vertices, all neighbours of v are of degree two. When branching on the set vertex, no two of its degree two neighbours in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ can have the same neighbours by the annotation procedure. This also leads to a decrease in measure of $\Delta w(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s)) \sum_{i=2}^{\infty} r_i(i-1)$.

In the other branch, we remove s decreasing the measure by $w(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s))$, and we reduce the degrees of the neighbours of s decreasing the measure by $\sum_{i=2}^{\infty} r_i \Delta v(i)$.

Let Δk_{take} and $\Delta k_{\text{discard}}$ be the decrease of the measure in the branch where we take the set corresponding to s and where we discard it, respectively. We have shown that:

$$\begin{aligned}\Delta k_{\text{take}} &\geq w(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) \sum_{i=2}^{\infty} r_i(i-1)\\ \Delta k_{\text{discard}} &\geq w(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} r_i \Delta v(i)\end{aligned}$$

Now, consider branching on an element vertex e in $I[(\mathcal{S}\cup\mathcal{U})\setminus A]$ with r_i neighbours of degree i in $I[(\mathcal{S}\cup\mathcal{U})\setminus A]$. Let $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ be the decrease of the measure in the branch where it is optional of cover the element corresponding to e and forbidden to cover it, respectively. In almost the same way, we deduce:

$$\begin{aligned}\Delta k_{\text{optional}} &\geq v(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} r_i \Delta w(i)\\ \Delta k_{\text{forbidden}} &\geq v(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} r_i w(i) + \Delta v(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) \sum_{i=2}^{\infty} r_i(i-1)\end{aligned}$$

We now consider all possible cases in which Algorithm 8.1 can branch. As a result, we obtain the following set of recurrence relations. Let $N(k)$ be the number of subproblems generated on a problem of measure k. For all $d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s), d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(e) \geq 3$ and r_i such that, $\sum_{i=2}^{d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)-1} r_i = d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)$ if we consider branching on vertex s representing a set, and $\sum_{i=2}^{d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(e)} r_i = d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(e)$ if we consider branching on a vertex e representing an element, we have:

$$\begin{aligned}N(k) &\leq N(k-\Delta k_{\text{take}}) + N(k-\Delta k_{\text{discard}})\\ N(k) &\leq N(k-\Delta k_{\text{optional}}) + N(k-\Delta k_{\text{forbidden}})\end{aligned}$$

An upper bound on the solution to this set of recurrence relations is of the form α^k.

Let $v_{\max} = \max_{i\in\mathbb{N}} v(i)$ and $w_{\max} = \max_{i\in\mathbb{N}} w(i)$. The graph G is transformed into a Set Cover instance $(\mathcal{S},\mathcal{U})$ of measure at most $(v_{\max}+w_{\max})n$. Therefore, we have proven that Algorithm 8.1 can be used to count the number of dominating sets of each cardinality κ in G in $\mathcal{O}(\alpha^{(v_{\max}+w_{\max})n})$ time and polynomial space.

We compute the weight functions v and w minimising $\alpha^{v_{\max}+w_{\max}}$ by computer (see Section 5.6). In this way, we have obtained $\alpha = 1.205693$ using the following weights:

i	2	3	4	5	6	>6
$v(i)$	0.640171	0.888601	0.969491	0.998628	1.000000	1.000000
$w(i)$	0.819150	1.218997	1.362801	1.402265	1.402265	1.402265

This proves a running time of $\mathcal{O}(1.205693^{(1+1.402265)n}) = \mathcal{O}(1.5673^n)$. □

Notice that the analysis of the running time is very similar to the analysis of the algorithms in Section 5.3.

We conclude this section with a remark on the weighted versions of Dominating Set and #Dominating Set.

MINIMUM WEIGHT DOMINATING SET
Input: A graph $G = (V, E)$ and a weight function $\omega : V \to \mathbb{R}_+$ and a number $k \in \mathbb{R}_+$.
Question: Does there exist a dominating set $D \subseteq V$ in G of total weight at most k?

The counting variant of the above problem, #MINIMUM WEIGHT DOMINATING SET, is defined analogously.

For MINIMUM WEIGHT DOMINATING SET the fastest polynomial-space algorithm is due to Fomin et al. and uses $\mathcal{O}(1.5780^n)$ time [145]. Later, Fomin et al. improved this at the cost of using exponential space to $\mathcal{O}(1.5535^n)$ time by giving an exponential-space algorithm for #MINIMUM WEIGHT DOMINATING SET [138].

As a corollary of Theorem 8.5, we can improve the polynomial-space results for MINIMUM WEIGHT DOMINATING SET and #MINIMUM WEIGHT DOMINATING SET if the set of possible weight sums, i.e., the set of possible weights that a vertex set can have, is polynomially bounded. This is the case, for example, when using a finite set of weights. We note that, under the same condition, we will also improve the above exponential-space results for MINIMUM WEIGHT DOMINATING SET and #MINIMUM WEIGHT DOMINATING SET in Section 8.5.

Corollary 8.6. *For* MINIMUM WEIGHT DOMINATING SET *and* #MINIMUM WEIGHT DOMINATING SET *with weight functions such that the set of possible weight sums is polynomially bounded, there are algorithms solving these problems in* $\mathcal{O}(1.5673^n)$ *time and polynomial space.*

Proof. We can modify Algorithm 8.1 such that it returns a polynomial size list containing the number of set covers of each weight κ instead of a linear size list containing the number of set covers of each cardinality. The running time follows from Theorem 8.5. □

8.4. Computing the Domatic Number in Polynomial Space

The second problem for which we use Algorithm 8.1 is DOMATIC NUMBER. For this problem, we cannot apply the algorithm as direct as in Theorem 8.5 to obtain our result. Instead, we will use the algorithm as a subroutine in the inclusion/exclusion framework of Björklund et al. [33]. In this way, we obtain the currently fastest polynomial-space algorithm for DOMATIC NUMBER.

Let us first introduce this problem formally. A *domatic k-partition* of a graph G is a partition $V_1, V_2, \ldots, V_k$ of the vertices V such that each V_i is a dominating set in G. The *domatic number* of G is the largest $k \in \mathbb{N}$ for which there exists a domatic k-partition, that is, it is the maximum number of disjoint dominating sets in G.

DOMATIC NUMBER
Input: A graph $G = (V, E)$ and an integer k.
Question: Can G be partitioned into at least k dominating sets?

Every graph has domatic number at least one, since the set of all vertices V is a dominating set in G. Moreover, every graph that has no isolated vertices has domatic number of at least two. This follows from the following simple observation: take any maximal independent set I in G (such a set must clearly exist); now, I is a dominating set because I is maximal (Proposition 1.1), and $V \setminus I$ is a dominating set because I is an independent set and there are no isolated vertices. Deciding whether the domatic number of G is at least k, for $k \geq 3$, is $\mathcal{NP}$-hard [162].

The first exponential-time algorithm related to this problem is due to Reige and Rothe: they gave an $\mathcal{O}(2.9416^n)$-time and polynomial-space algorithm to decide whether the domatic number of a graph is at least three [265] (this problem is also known as 3-Domatic Number). Later, Fomin et al. gave an $\mathcal{O}(2.8718^n)$-time and exponential-space algorithm for Domatic Number in [146]. Using the set partitioning via inclusion/exclusion framework, Björklund et al. have improved this to $\mathcal{O}^*(2^n)$ time and space [33], and later to $\mathcal{O}^*(2^n)$ time and $\mathcal{O}(1.7159^n)$ space [31]. Using only polynomial space, they also gave an $\mathcal{O}(2.8718^n)$-time algorithm that uses the minimal dominating set enumeration procedure of Fomin et al. [146]. Finally, Reige et al. show that the 3-Domatic Number can be computed in $\mathcal{O}(2.695^n)$ time and polynomial space [266].

In this section, we will give an $\mathcal{O}(2.7139^n)$-time and polynomial-space algorithm for Domatic Number. This improves previous results using polynomial space, and its running time nears that of the best known polynomial space result on the special case of 3-Domatic Number [266].

We will first introduce the reader to the set partitioning via inclusion/exclusion framework of Björklund et al. [33]. They use the following result to compute the domatic number.

Proposition 8.7 ([33]). *Let $\mathcal{D}$ be a set of sets over the universe $\mathcal{V}$, and let $k \in \mathbb{N}$. The number of ways $p_\kappa(\mathcal{D})$ to partition $\mathcal{V}$ into k sets from $\mathcal{D}$ equals:*

$$p_k(\mathcal{D}) = \sum_{X \subseteq \mathcal{V}} (-1)^{|X|} a_k(X)$$

where $a_k(X)$ equals the number of k-tuples $(S_1, S_2, \ldots, S_k)$ with $S_i \in \{D \in \mathcal{D} \mid D \cap X = \emptyset\}$ and for which $\sum_{i=1}^{k} |S_i| = |\mathcal{V}|$.

Proof. Direct application the inclusion/exclusion formula, see Section 2.3 or [33]. □

If we let $\mathcal{V} = V$ and let $\mathcal{D}$ be the set of dominating sets in G, then Proposition 8.7 can be used to check whether there exists a domatic k-partition, i.e., whether the domatic number of a graph is at least k, by checking whether $p_k(\mathcal{D}) > 0$ or not. To do so, we would need to compute the numbers $a_k(X)$. In other words, for every $X \subseteq V$, we would need to compute the number of k-tuples $(S_1, S_2, \ldots, S_k)$ with $S_i \in \{D \in \mathcal{D} \mid D \cap X = \emptyset\}$ and for which $\sum_{i=1}^{k} |S_i| = n$.

Let $d_\kappa(V')$ be the number of dominating sets of size κ in G using only vertices from $V' \subseteq V$. Björklund et al. [33] show how to compute the values $a_k(X)$ required to use Proposition 8.7 from the numbers $d_\kappa(V')$. To this end, they give the following recurrence that can be used for dynamic programming.

Let $a_X(i, j)$ be the number of i-tuples $(S_1, S_2, \ldots, S_i)$ where each S_i is a dominating set in G using only vertices from $V \setminus X$ and such that $\sum_{i=1}^{k} |S_i| = j$. The values $a_X(i, j)$

can be computed by dynamic programming over the following recurrence summing over the possible sizes for the i-th element of the i-tuple $(S_1, S_2, \ldots, S_i)$:

$$a_X(i,j) = \sum_{l=0}^{j} a_X(i-1, j-l) \cdot d_l(V \setminus X)$$

Björklund et al. obtain the required values by observing that $a_k(X) = a_X(k,n)$.

We are now ready to prove the main result of this section.

Theorem 8.8. *The domatic number of a graph can be computed in $\mathcal{O}(2.7139^n)$ time and polynomial space.*

Proof. We can use Proposition 8.7 for increasing $k = 3, 4, \ldots$ to find the domatic number of G. To do so, we need to compute the required values $a_k(X)$.

For each $V' \subseteq V$, we use Algorithm 8.1 on $(\mathcal{S}, \mathcal{U})$ where $\mathcal{S} = \{N[v] \mid v \in V'\}$ and $\mathcal{U} = V$ to produce a list containing, for each κ, $0 \leq \kappa \leq n$, the number of dominating sets in G of size κ using only vertices in V'. We observe that each such list can be used to compute a single value $a_k(X)$ from the formula in Proposition 8.7 by the dynamic programming procedure shown above.

We obtain a polynomial-space algorithm for the domatic number by producing these lists one at a time and summing their corresponding contributions to the formula. Since the 2^n calls to Algorithm 8.1 are on instances of different sizes, the total number of subproblems generated by Algorithm 8.1 over all 2^n calls can be bounded from above by:

$$\sum_{i=0}^{n} \binom{n}{i} \alpha^{v_{\max} n + w_{\max} i} = (\alpha^{v_{\max}}(1 + \alpha^{w_{\max}}))^n$$

Here, we use the notation from the proof of Theorem 8.5. Because this number of subproblems equals the exponential factor in the running time, we conclude that the algorithm runs in $\mathcal{O}^*((\alpha^{v_{\max}}(1+\alpha^{w_{\max}}))^n)$ time.

We recompute the weight functions used in Theorem 8.5 minimising $\alpha^{v_{\max}}(1 + \alpha^{w_{\max}})$ and obtain $\alpha = 1.099437$ using the following set of weights:

i	2	3	4	5	6	7	> 7
$v(i)$	0.822647	0.971653	1.000000	1.000000	1.000000	1.000000	1.000000
$w(i)$	2.039702	3.220610	3.711808	3.919500	4.016990	4.052718	4.052729

The running time is $\mathcal{O}\big((1.099437(1+1.099437^{4.052729}))^n\big) = \mathcal{O}(2.7139^n)$. □

We notice that one can construct the associated domatic partitions (or dominating sets) by repeating the counting algorithms a polynomial number of times in the following way. First compute the maximum size k of a domatic partition. Then, assign a vertex to a partition and test if this alters the domatic number: this can be done by cleverly merging the output of calls to Algorithm 8.1 on a different input corresponding to each of the k thus far constructed partitions. If the domatic number changes, we try assigning the vertex to another partition, otherwise, we repeat this process.

8.5. Counting Dominating Sets in Exponential Space

We now drop the constraint of using only polynomial space and give an exponential-space algorithm for #Dominating Set. This algorithm will be similar to the one in Section 8.3 but it will switch to a dynamic programming approach on tree decompositions at an earlier stage. This causes the dynamic programming to use exponential time and space, while it allows the branching phase of the algorithm to avoid the less efficient branchings on lower degree vertices in the incidence graph.

If we would directly apply this modification to Algorithm 8.1, then this would result in an $\mathcal{O}(1.5014^n)$-time algorithm for #Dominating Set: the construction would be identical to the exponential-space algorithm for Partial Dominating Set in Section 9.2 in the next chapter. Here, we will use a slightly different approach that mixes the dynamic programming on tree decompositions with a standard branch-and-reduce algorithm. This will give a slightly better result: an $\mathcal{O}(1.5002^n)$-time algorithm.

This result is again based on an algorithm that counts the number of set covers of each size κ ($0 \leq \kappa \leq n$) and that uses the combination of the two branching rules from Section 8.1 with dynamic programming on tree decompositions: Algorithm 8.2. Different from previous algorithms is that the algorithm takes as input the incidence graph of a Set Cover instance $(\mathcal{S}, \mathcal{U})$ and a *multiplicity function* $m : \mathcal{S} \to \mathbb{N}$. Initially, this function takes the value 1 for each set, i.e, for all $s \in \mathcal{S}$: $m(s) = 1$. The purpose of this multiplicity function will become clear from the discussion of the reduction rules.

We will start by considering the reduction rules of Algorithm 8.2. First, observe that not all reduction rules from the branch-and-reduce algorithm in Section 5.3.8 can be used in the setting where we count the number of solutions. For example, consider the subsets rule from Section 5.3.4: if there exist two sets $S, R \in \mathcal{S}$ with $R \subseteq S$, then remove R. This works to find a minimum set cover, but not for counting them as a set cover containing R and not S could very well be a minimum set cover.

As an alternative to the subsets rule, we introduce the identical sets rule that uses the multiplicity function. Here, we replace multiple vertices in I that represent different sets that contain the same elements by a single vertex and keep track of their total number by storing this number in the multiplicity function. In other words, the multiplicity function stores, for each vertex $s \in \mathcal{S}$ that represents a set, the number $m(s)$ of identical copies of this set that are represented by the vertex. This leads to the following reduction rule (lines 1-3 in Algorithm 8.2):

Reduction Rule 8.1.

if there exist two vertices $s_1, s_2 \in \mathcal{S}$ with $N(s_1) = N(s_2)$ **then**
  Modify m such that $m(s_1) := m(s_1) + m(s_2)$
  return ExpCSC($I[(\mathcal{S} \cup \mathcal{U}) \setminus \{s_2\}], m$)

If we select a set to be in the set covers that we are counting at a later stage in the algorithm, for example at lines 6 or 14, then we have to correct the computed numbers for the fact that multiple sets are represented by this vertex. This can be done using the following proposition.

Proposition 8.9. *Let x_κ be the number of set covers of size κ computed in a recursive call to the algorithm where a set represented by a vertex $s \in \mathcal{S}$ is decided to be in*

Algorithm 8.2. An exponential-space algorithm for counting set covers of each size.

Input: the incidence graph $I = (\mathcal{S} \cup \mathcal{U}, E)$ of $(\mathcal{S}, \mathcal{U})$, a multiplicity function $m : \mathcal{S} \to \mathbb{N}$
Output: a list containing the number of set covers of $(\mathcal{S}, \mathcal{U})$ of each cardinality κ
ExpCSC(I, m):

if there exist two vertices $s_1, s_2 \in \mathcal{S}$ with $N(s_1) = N(s_2)$ **then**
  Modify m such that $m(s_1) := m(s_1) + m(s_2)$
  return ExpCSC$(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{s_2\}], m)$
else if there exists a vertex $e \in \mathcal{U}$ of degree one **then**
  Let $L_{\text{take}} = \text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus N[s]], m)$ with s the unique neighbour of e
  return L_{take} after updating it using Proposition 8.9
else if there exist two vertices $e_1, e_2 \in \mathcal{U}$ such that $N[e_1] \subseteq N[e_2]$ **then**
  return ExpCSC$(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{e_2\}, m)$
else
  Let $s \in \mathcal{S}$ and $e \in \mathcal{U}$ be two vertices with maximum degrees from the set of vertices of I that are not an exceptional case as defined in Overview 8.1
  if $d(s) \leq 3$ **and** $d(e) \leq 4$ **then**
    return ExpCSC-DP(I, m)
  else if $d(s) > d(e)$ **or** $d(e) \leq 4$ **then**
    Let $L_{\text{take}} = \text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus N[s]], m)$
    Let $L_{\text{discard}} = \text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{s\}], m)$
    Update L_{take} using Proposition 8.9
    return $L_{\text{take}} + L_{\text{discard}}$
  else
    Let $L_{\text{optional}} = \text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{e\}], m)$
    Let $L_{\text{forbidden}} = \text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus N[e]], m)$
    return $L_{\text{optional}} - L_{\text{forbidden}}$

every set cover. That is, in a recursive call where we have decided to take at least one of the $m(s)$ copies of the set represented by s in the set cover, but where we have not yet taken this into account in the values x_κ: we have removed only $N[s]$ from the incidence graph. Also, let x'_κ be the number of set covers of size κ when adjusted for the fact that we could have taken any number (but at least one) of the $m(s)$ copies of the set represented by s. Then:

$$x'_\kappa = \sum_{i=1}^{m(s)} \binom{m}{i} x_{\kappa-i}$$

Proof. We sum over the different number of copies i of the set represented by s that we can take. The binomial gives the number of ways to pick these i copies from the total of $m(s)$ copies. □

We note that, in the above formula, we assume that $x_i = 0$ for i outside the range $0 \leq i \leq n$.

As a consequence of Proposition 8.9, we can update the values in the list containing the number of set covers of each size κ maintained by the algorithm in polynomial time whenever we decide to take at least one copy of some set $s \in \mathcal{S}$ in all set covers that we are counting. Algorithm 8.2 uses this proposition whenever it decides to count only set covers containing (a copy of) a certain set: see lines 6 and 14.

The second reduction rule of Algorithm 8.2 is the following (lines 4-6):

Reduction Rule 8.2.

if there exists a vertex $e \in \mathcal{U}$ of degree one **then**
  Let $L_{\text{take}} = \text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus N[s]], m)$ with s the unique neighbour of e
  return L_{take} after updating it using Proposition 8.9

This reduction rule is almost identical the unique elements rule in Section 5.3.2. The only difference is that, due to the multiplicities of the sets, the unique set containing the element may occur as multiple copies. In this case, we still must take at least one of these copies. We recursively solve the problem where we take such a set and use Proposition 8.9 to correct the computed values for the fact that we could also have taken a number of copies of the set.

What remains is the third reduction rule (lines 7-8), which is identical to the subsumption rule in Section 5.3.6. Here, the multiplicities of the sets do not play any role: if an element e_1 occurs in all sets (with or without multiplicities) in which another element e_2 occurs, then any subset $\mathcal{C} \subseteq \mathcal{S}$ that covers e_1 also covers e_2; hence, we can remove e_2.

Reduction Rule 8.3.

if there exist two vertices $e_1, e_2 \in \mathcal{U}$ such that $N[e_1] \subseteq N[e_2]$ **then**
  return $\text{ExpCSC}(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{e_2\}, m)$

Having treated the reduction rules, we now continue with the branching rules of the algorithm. Similar to Algorithm 8.1 the algorithm branches on a vertex of maximum degree and in general prefers to branch on a vertex that represents an element if degrees

There are exceptional cases on which Algorithm 8.2 does not branch. These cases represent local configurations of vertices which would increase the running time of the algorithm when branched on, and that can be handled by the dynamic programming on path decompositions quite effectively. The exceptional cases are:

1. Vertices of degree five that represent elements that occur in many sets of small cardinality. More specifically, if we let a 5-tuple $(s_1, s_2, s_3, s_4, s_5)$ represent a vertex e of degree five representing an element with s_i neighbours of degree i that represent a set, then our special cases can be denoted as:

   (1, 4, 0, 0, 0) (1, 3, 1, 0, 0) (1, 2, 2, 0, 0) (1, 1, 3, 0, 0) (1, 0, 4, 0, 0)
   (1, 3, 0, 1, 0) (1, 2, 1, 1, 0) (0, 5, 0, 0, 0) (0, 4, 1, 0, 0) (0, 3, 2, 0, 0)
   (0, 2, 3, 0, 0) (0, 1, 4, 0, 0) (0, 4, 0, 1, 0) (0, 3, 1, 1, 0)

   Note that e can have at most one neighbour of degree one due to Reduction Rule 8.1.

2. Vertices of degree four or five representing sets that contain an element represented by a vertex that corresponds to one of the exceptional cases defined above.

Overview 8.1. Exceptional Cases for Algorithm 8.2

are equal. A small difference is that because of the use of the multiplicity function m, the algorithm uses Proposition 8.9 to update the list containing the number of set covers of each size κ whenever it decides to take (at least one copy of) a set in the set covers it is counting.

There are also some big differences between Algorithm 8.2 and Algorithm 8.1. First of all, the algorithm never branches on vertices of degree at most three, and it never branches on vertices of degree four that represent an element: these are left for the dynamic programming phase of the algorithm. This means that if the maximum degree in I is four and there exist vertices of degree four representing sets, then these are used for branching whether vertices of degree four representing elements exist or not. Also, the algorithm does not branch on vertices that are considered an exceptional case as defined in Overview 8.1. These are vertices of degree five representing elements that have specific local configurations, and vertices representing sets that contain elements represented by vertices with such a specific local configuration.

These exceptional cases exist because, in the analysis in Lemmas 8.11 and 8.12, we often know the local configuration of a vertex representing a set or an element in the incidence graph. Such neighbourhoods are important for the worst-case behaviour of the algorithm; for some neighbourhoods it is more efficient to handle them in the dynamic programming phase of our algorithm than by branching. These neighbourhoods are our exceptional cases. How this influences the running time of our algorithms will become more clear from the proofs of Lemmas 8.11 and 8.12.

We conclude the description of the algorithm by considering the dynamic programming subroutine ExpCSC-DP(I, m). This subroutine first uses the pathwidth bounds due to Fomin et al. [138] in Proposition 2.16 to construct a path decomposition of small pathwidth. Hereafter, it applies the following result. For the definition of a path decomposition and some related concepts, see Section 2.2.2.

Proposition 8.10. *Let I be the incidence graph of a set cover instance $(\mathcal{S}, \mathcal{U})$ given with a path decomposition X of I of width at most p, and let $m : \mathcal{S} \to \mathbb{N}$ be a multiplicity*

function. The number of set covers of $(\mathcal{S}, \mathcal{U})$ *of each size* κ, $0 \leq \kappa \leq n$, *can be computed in* $\mathcal{O}^*(2^p)$ *time.*

Proof. If m is the all-1 function, then the problem is equivalent to counting red-blue dominating sets in I. This can be done in $\mathcal{O}^*(2^p)$ on path decompositions of width p by applying two modifications to the algorithm in Proposition 2.12. First, we let the algorithm consider red-blue dominating sets instead of dominating sets. As a result it runs in $\mathcal{O}^*(2^p)$ time instead of $\mathcal{O}^*(3^p)$ time; for details see Proposition 11.11. Secondly, the algorithm is modified such that it counts the number of solutions instead of computing the size of a solutions; for an example, see Theorem 11.7. This modification can be done in such a way that the multiplicity of sets stored in m is respected by using the formula given in Proposition 8.9 in each introduce bag. □

Remark 8.1. Consider the pathwidth bound of Proposition 2.16. If our incidence graph I has a vertex of degree d with a neighbour of degree one, then this vertex can be considered to be of degree $d-1$ in the formula of Proposition 2.16. Namely, if we remove all degree one vertices from I and then compute a path decomposition, then we can reintroduce these vertices in the following way. Let $(X_1, X_2, \ldots, X_l)$ the path decomposition constructed by Proposition 2.16, and let X_i be a bag containing the neighbour of the degree one vertex v. Now, we can add v by inserting the bag $X_i \cup \{v\}$ between the bags X_i and X_{i+1}. Repeating this for all degree one vertices increases the pathwidth by at most one.

We now give a bound on the running time of Algorithm 8.2. To this end, we give two lemmas: the first one will deal with the branching phase of the algorithm, and the second one will deal with the dynamic programming phase. Since both lemmas use the same measure on the size of the subproblems, we will introduce it first. The measure was chosen to minimise the resulting running time; see Section 5.6.

Let k be the following measure:

$$k := \sum_{e \in \mathcal{U}} v(d(e)) + \sum_{s \in \mathcal{S}} w(d(s))$$

using fixed weight functions $v, w : \mathbb{N} \to \mathbb{R}_+$ with the weights given below.

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.275603	0.551206	0.658484	0.719009	0.745509	0.749538	0.749538
$w(i)$	0.354816	0.625964	0.785112	0.923325	0.986711	1.000000	1.000000	1.000000

We again use the quantities $\Delta v(i) = v(i) - v(i-1)$ and $\Delta w(i) = w(i) - w(i-1)$ and observe that the measure satisfies the following constraints:

1. $v(0) = v(1) = w(0) = 0$
2. $\Delta v(i) \geq 0$ for all $i \leq 1$
3. $\Delta w(i) \geq 0$ for all $i \leq 1$
4. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 1$
5. $\Delta w(i) \geq \Delta w(i+1)$ for all $i \geq 1$
6. $2\Delta v(5) \leq v(2)$

The roles of these constraints are identical to their roles in the proof of Theorem 8.5.

We start the analysis of the running time of Algorithm 8.2 by bounding the number of subproblems generated by branching.

Lemma 8.11. *Let $N_h(k)$ be the number of subproblems of measure h generated by Algorithm 8.2 on an input of measure k. Then:*

$$N_h(k) < 1.26089^{k-h}$$

Proof. Similar to the proof of Theorem 8.5, we derive a set of recurrence relations of the following form:

$$N_h(k) \leq N_h(k - \Delta k_{\text{optional}}) + N_h(k - \Delta k_{\text{forbidden}})$$

$$N_h(k) \leq N_h(k - \Delta k_{\text{discard}}) + N_h(k - \Delta k_{\text{take}})$$

with the appropriate values of $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ for every possible branching on a vertex representing an element, and the appropriate values $\Delta k_{\text{discard}}$ and Δk_{take} for every possible branching on a vertex representing a set.

The difference here is that $N_h(k)$ is defined to be number of subproblems of measure h, instead of the total number of subproblems, generated on an input of measure k. An upper bound on the solution of this set of recurrence relations is of the form α^{k-h}.

If we branch on a vertex representing an element, this leads to almost the same recurrence relations as in Theorem 8.5. Let e be a vertex representing an element with s_i neighbours of degree i, and let $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ be the decrease in the measure in the branch where we make it optional to cover the element represented by e, or forbidden to cover the element represented by e, respectively.

$$\begin{aligned} \Delta k_{\text{optional}} &\geq v(d(e)) + \sum_{i=1}^{\infty} s_i \Delta w(i) \\ \Delta k_{\text{forbidden}} &\geq v(d(e)) + \sum_{i=1}^{\infty} s_i w(i) + \Delta v(d(e)) \sum_{i=1}^{\infty} (i-1) s_i \end{aligned}$$

The only difference with the same recurrence relations in Theorem 8.5 is that we now also consider neighbours of degree one.

If we branch on a vertex s representing a set, then the changes are somewhat larger. Let s be a vertex representing a set with e_i neighbours of degree i, and let $\Delta k_{\text{discard}}$ and Δk_{take} be the decrease in the measure in the branch where we take some of the $m(s)$ sets represented by s in a set cover, or where we discard them, respectively. Different to the formula for $\Delta k_{\text{discard}}$ given in Theorem 8.5, reduction rules now fire when neighbours of degree two are involved. In the branch where the sets represented by s are discarded, these vertices get degree one and are removed by the reduction rules. As these degree two neighbours all represent elements, they cannot have any common neighbour besides s as Reduction Rule 8.3 would otherwise have fired before branching. Therefore, Reduction Rule 8.2 removes at least one additional set of weight at least $w(1)$ per neighbour of degree two.

In this way, we obtain the following bounds on $\Delta k_{\text{discard}}$ and Δk_{take}:

$$\begin{aligned} \Delta k_{\text{discard}} &\geq w(d(s)) + \sum_{i=2}^{\infty} e_i \Delta v(i) + e_2 w(1) \\ \Delta k_{\text{take}} &\geq w(d(s)) + \sum_{i=2}^{\infty} e_i v(i) + \Delta w(d(S)) \sum_{i=2}^{\infty} (i-1) e_i \end{aligned}$$

Notice that the exact branching rules and, in particular, the exceptional cases in Overview 8.1 play an important role in the analysis as they restrict the generated set of recurrence relations. Using the measure defined above this lemma, we solve the recurrence relations in the same way as in previous measure-and-conquer analyses and obtain an upper bound on their solution of $N_h(k) \leq 1.26089^{k-h}$. □

Next, we prove a bound on the running time of a call to ExpCSC-DP(I, m). We note that, although being slightly more complicated, the analysis is almost identical to a similar analysis in [138].

Lemma 8.12. *ExpCSC-DP(I, m) runs in time $\mathcal{O}(1.25334^k)$ when called on a set cover instance of measure k.*

Proof. We will prove an upper bound on the pathwidth of an incidence graphs of measure k that is an input of ExpCSC-DP(I, m). To this end, we formulate a linear program in which all variables have the domain $[0, \infty)$.

We start by stating the first part of the linear program:

$$\begin{aligned} \max \quad z &= \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 \qquad \text{such that:} \\ 1 &= \sum_{i=1}^{5} w(i)x_i + \sum_{i=2}^{5} v(i)y_i \qquad (8.1) \\ \sum_{i=1}^{5} ix_i &= \sum_{i=2}^{5} iy_i \qquad (8.2) \end{aligned}$$

Here, x_i and y_i represent the number of vertices of degree i that represent a set or an element per unit of measure in a worst-case instance, respectively. Using these variables, Constraint 8.1 guarantees that these x_k and y_k use exactly one unit of measure, and Constraint 8.2 guarantees that both partitions of the bipartite incidence graph have an equal number of edges.

The objective function, however, is formulated in terms of the variables n_i. It is based on Proposition 2.16 and represents the maximum pathwidth z of a graph per unit of measure up to the term ϵn. The variables n_i represent the number of vertices of degree i in the input graph per unit of measure. Following Remark 8.1, these degrees are taken after removing any vertices of degree one that represent sets. Any such vertex of degree one that represents a set will use measure and not increase the pathwidth, so we can assume that they will exist only if involved in an exceptional case.

Let C be the set of exceptional cases for vertices of degree five representing elements defined in Overview 8.1, and let c_i be the number of neighbours of degree i of exceptional case $c \in C$. We introduce the variables p_c for the number of occurrences of each exceptional case $c \in C$ per unit of measure. These definitions directly give us the following additional constraints that can be added to the linear program:

$$n_3 = x_3 + y_3 \qquad (8.3)$$

$$n_4 = x_4 + y_4 + \sum_{c \in C, c_1 > 0} p_c \qquad (8.4)$$

$$n_5 = x_5 + \sum_{c \in C, c_1 = 0} p_c \tag{8.5}$$

$$y_5 = \sum_{c \in C} p_c \tag{8.6}$$

Notice that we use here that, in a subproblem on which ExpCSC-DP(I, m) is called, a vertex can have degree at most five. We also use that, in such a subproblem, vertices of degree five exist only if they are exceptional cases (Overview 8.1).

The next thing to do is to add additional constraints justified by the exceptional cases. Observe that whenever $p_c > 0$ for some $c \in C$, then there exist further restrictions on the instance because we know the cardinalities of the sets in which these exceptional elements occur. We impose a lower bound on the number of sets of cardinalities one, two, and three in the instance by introducing Constraint 8.7. Also, we impose an upper bound on the number of sets of cardinality four, five, and six by using that there can be at most one such set per exceptional frequency five element contained in it. This is done in Constraint 8.8.

$$x_i \geq \sum_{c \in \mathcal{C}} \frac{c_i}{i} p_c \qquad \text{for } i \in \{1, 2, 3\} \tag{8.7}$$

$$x_i \leq \sum_{c \in \mathcal{C}} c_i p_c \qquad \text{for } i \in \{4, 5\} \tag{8.8}$$

The solution to this linear program is $z = 0.325782$ with all variables equal to zero, except: $x_3 = n_3 = 0.781876$ and $n_4 = y_4 = 0.586407$. As a result the dynamic programming on the path decomposition can be done in time $\mathcal{O}^*(2^{(z+\epsilon)k}) = \mathcal{O}(1.25334^k)$.

We complete the proof by noting that although Proposition 2.16 applies only to graphs of size at least n_ϵ, the result holds because we can fix ϵ to be small enough to disappear in the rounding of the running time and consider all smaller graphs than n_ϵ to be handled in constant time. □

Combining Lemma 8.11 and Lemma 8.12 gives the main result of this section.

Theorem 8.13. *There exists an algorithm that counts the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a graph G in $\mathcal{O}(1.5002^n)$ time and space.*

Proof. Let $T(k)$ be the time used on a problem of measure k, and let H_k be the set of all possible measures that subproblems of a problem of measures k can have. Then, by Lemma 8.11 and Lemma 8.12:

$$T(k) \leq \sum_{h \in H_k} N_h(k) \cdot 1.25334^h \leq \sum_{h \in H_k} 1.26089^{k-h} \cdot 1.25334^h \leq \sum_{h \in H_k} 1.26089^k$$

Because we use only a finite number of weights, $|H_k|$ is polynomially bounded. Therefore, Algorithm 8.2 runs in $\mathcal{O}(1.26089^k)$ time.

Using the transformation between dominating sets and set covers from Section 5.1.1 (also used in Theorem 8.5), this proves a running time of $\mathcal{O}(1.26089^{(v_{\max}+w_{\max})n}) = \mathcal{O}(1.26089^{(0.749538+1)n}) = \mathcal{O}(1.5002^n)$. □

Similar to Corollary 8.6 in Section 8.3, we can also use a modification of Algorithm 8.2 to solve the weighted problems Minimum Weight Dominating Set and #Minimum Weight Dominating Set when restricted to instances in which the set of possible weight sums is polynomially bounded.

Corollary 8.14. *There exist algorithms that solve* Minimum Weight Dominating Set *and* #Minimum Weight Dominating Set *in* $\mathcal{O}(1.5002^n)$ *time and space when the problems are restricted to using weight functions with the property that the set of possible weight sums is polynomially bounded.*

Proof. Identical to Corollary 8.6 using Theorem 8.13 instead of Theorem 8.5. □

Another problem that we can solve using Algorithm 8.2 is Red-Blue Dominating Set. For this problem, we will now give a faster algorithm than in Corollary 5.5. We improve the running time in this corollary at the cost of using exponential space.

Corollary 8.15. *There exists an algorithm that solves* Red-Blue Dominating Set *in* $\mathcal{O}(1.2252^n)$ *time and space.*

Proof. We repeat the proof of Theorem 8.13 using a different set of weights. We chose these weights in order to minimise $\alpha^{\max\{v_{\max},w_{\max}\}n}$ instead of $\alpha^{(v_{\max}+w_{\max})n}$); this is similar to Corollary 5.5. Because an instance of Red-Blue Dominating Set can be transformed into an instance of Set Cover that satisfies $|\mathcal{S}| + |\mathcal{U}| = n$, the instance has a measure k of at most $\max\{v_{\max}, w_{\max}\}n$. Hence, a running time of $\mathcal{O}(\alpha^{\max\{v_{\max},w_{\max}\}n})$ follows from the computed value of α.

Since the proof is entirely analogous to the proof of Theorem 8.13, including the analyses in Lemmas 8.11 and 8.12, we will give only the main differences here.

We use the following new set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.357141	0.714281	0.861804	0.947324	0.991880	1.000000	1.000000
$w(i)$	0.000000	0.421168	0.684691	0.840134	0.953577	0.998175	0.999999	1.000000

Using these weights, we again solve the recurrence relations given in the proof of Lemma 8.11. This results in an upper bound on the number of subproblems $N_h(k)$ of measure h generated to solve an instance of measure k of $N_h(k) \leq 1.22519^k$.

Next, we again solve the linear program from the proof of Lemma 8.12 to prove an upper bound on the pathwidth of the instances that are solved by dynamic programming on path decompositions. Notice that some of the coefficients in the linear constraints of the linear program are changed because we now use different weights. The solution to this new linear program is $z = 0.280303$ with all variables equal to zero, except: $x_3 = n_3 = 0.672727$ and $n_4 = y_4 = 0.504545$. As a result, the dynamic programming on the path decomposition can be done using Proposition 8.10 in time $\mathcal{O}^*(2^{(z+\epsilon)k}) = \mathcal{O}(1.21445^k)$.

The $\mathcal{O}(1.2252^n)$ upper bound on the running time now follows in the same way as in the proof of Theorem 8.13. □

graph class	result of Gaspers et al. [165]	our result
c-dense graphs	$\mathcal{O}(1.5063^{(\frac{1}{2}+\frac{1}{2}\sqrt{1-2c})n})$	$\mathcal{O}(1.5002^{(\frac{1}{4}+\frac{1}{4}\sqrt{9-16c})n})$
chordal graphs	$\mathcal{O}(1.4124^n)$	$\mathcal{O}(1.3687^n)$
weakly chordal graphs	$\mathcal{O}(1.4776^n)$	$\mathcal{O}(1.4590^n)$
4-chordal graphs	$\mathcal{O}(1.4845^n)$	$\mathcal{O}(1.4700^n)$
circle graphs	$\mathcal{O}(1.4887^n)$	$\mathcal{O}(1.4764^n)$

Table 8.1. Results for Dominating Set on five different graph classes.

8.6. Dominating Set Restricted to Some Graph Classes

We conclude this chapter with another application of inclusion/exclusion-based branching. Namely, we study Dominating Set restricted to some graph classes. We consider the algorithms for these problems by Gaspers et al. [165]. We improve their algorithms by applying both the traditional branching rule that branches on sets and our inclusion/exclusion-based branching rule that branches on elements in this setting. The series of subproblems that is generated by this approach will be solved using Algorithm 8.2.

Gaspers et al. consider exact exponential-time algorithms for Dominating Set on some graph classes on which this problem remains $\mathcal{NP}$-complete [165]. They consider c-dense graphs, circle graphs, chordal graphs, 4-chordal graphs, and weakly chordal graphs. They show that if we restrict ourselves to such a graph class, then either there are many vertices of high degree allowing more efficient branching, or the graph has low treewidth allowing us to efficiently solve the problem by dynamic programming on a tree decomposition.

In this section, we show that we can do with fewer vertices of high degree to obtain the same effect by using our two branching rules. In this way, we improving the results on four of these graph classes. See Table 8.1 for an overview of the results. We note that, on some graph classes, part of the improvement comes from using faster algorithms to solve the problem on tree decompositions; details on these algorithms will be presented in Chapter 11.

We begin by showing that having many vertices of high degree can be beneficial to the running time of an algorithm. First, consider the following result of Gaspers et al. [165]:

Proposition 8.16 ([165]). *Let $t \geq 1$ be a fixed integer, and let α be such that there exists an $\mathcal{O}(\alpha^n)$-time algorithm for* Red-Blue Dominating Set. *If $\mathcal{G}_t$ is a class of graphs with, for all $G \in \mathcal{G}_t$, $|\{v \in V \ : \ d(v) \geq t-2\}| \geq t$, then there is an $\mathcal{O}(\alpha^{2n-t})$-time algorithm to solve* Dominating Set *on graphs in $\mathcal{G}_t$.*

We now give and prove a stronger variant of this proposition; the resulting lemma uses Algorithm 8.2 and its running-time analysis that we have given in Section 8.5.

Lemma 8.17. *Let, for each integer $t \geq 1$, $\mathcal{G}_t$ be the class of graphs with $|\{v \in V \ : \ d(v) \geq t-2\}| \geq \frac{1}{2}t$. There is an algorithm that counts the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a graph $G \in \mathcal{G}_t$ in $\mathcal{O}(1.5002^{n-\frac{1}{2}t})$ time.*

Proof. Given a graph G, we determine the largest integer $t \geq 1$ such that $G \in \mathcal{G}_t$. It is easy to see that this can be done in polynomial time. Let H be the set of vertices of degree at least $t-2$ in G from the definition of $\mathcal{G}_t$.

Consider the set cover formulation $(\mathcal{S}, \mathcal{U})$ with incidence graph I of the dominating set problem on G. Also, let α be the base of the exponent of the running time of Algorithm 8.2 using the measure given in Section 8.5 with maximum weights $v_{\max}$ and $w_{\max}$ for a vertex representing an element and a set, respectively. We will prove the lemma by giving a procedure that generates $t+1$ instances of measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ from which the result can be computed. We solve these instances using Algorithm 8.2. Since the number of instances generated is at most linear in n, this gives an algorithm running in $\mathcal{O}^*(\alpha^{(v_{\max}+w_{\max})(n-\frac{1}{2}t)}) = \mathcal{O}(1.5002^{n-\frac{1}{2}t})$ time.

Let e be a vertex corresponding to an element in $\mathcal{U}$ constructed for a vertex in H. Our procedure first uses inclusion/exclusion-based branching on e: in the optional branch, we remove e; and in the forbidden branch, we remove $N[e]$. Because $N[e]$ contains at least $t-1$ neighbours representing sets constructed for v and its neighbours, the instance generated in the forbidden branch has measure at most $(n-1)v_{\max} + (n-t+1)w_{\max} < (v_{\max}+w_{\max})(n-\frac{1}{2}t)$ as $w_{\max} \geq v_{\max}$. In the optional branch, we branch on the next vertex corresponding to an element in $\mathcal{U}$ constructed for a vertex in H. We repeat this process until all vertices in H are used. This gives a series of $\frac{1}{2}t$ instances of measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ and one instance of measure at most $v_{\max}(n-\frac{1}{2}t) + w_{\max}n$ in which all vertices corresponding to an element in $\mathcal{U}$ constructed for a vertex in H are exhausted.

In this last instance, we branch on vertices corresponding to sets. These vertices still have at least $\frac{1}{2}t-1$ neighbours in I, since only $\frac{1}{2}t$ vertices have been removed in optional branches. Let s be a vertex corresponding to a set in $\mathcal{S}$ corresponding to a vertex in H. When branching on s and taking the corresponding set in a solution, again a subproblem of measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ is generated as $N[s]$ is removed. In the branch where we discard the set corresponding to s, we continue by branching on the next set corresponding to a vertex in H. In this way, we again generate a series of $\frac{1}{2}t$ instances of measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ and one instance in which all vertices to branch on are exhausted.

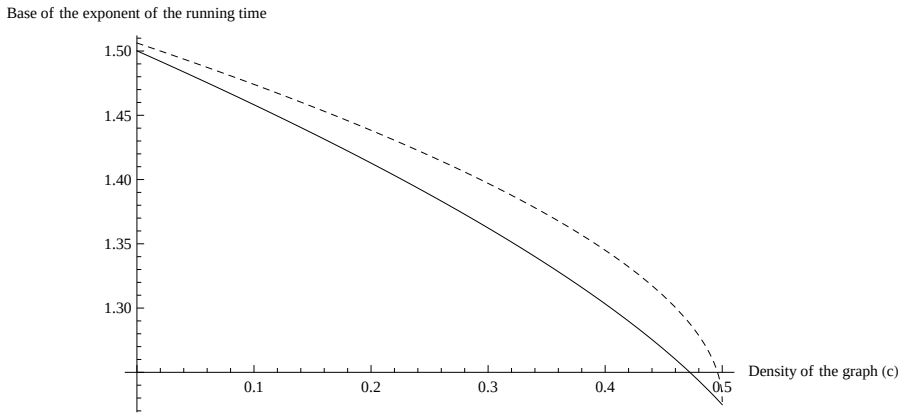
In the remaining instance, $\frac{1}{2}t$ vertices corresponding to elements are removed and $\frac{1}{2}t$ vertices corresponding to sets are removed. This results in an instance that also has measure at most $(v_{\max}+w_{\max})(n-\frac{1}{2}t)$ proving the lemma. □

We note that the same result could have been obtained if $w_{\max} < v_{\max}$ by switching the order of the branching rules: then, we would first branch on vertices representing sets, and thereafter on vertices representing elements.

We now continue by defining the different graph classes and giving the results on these graph classes by using Lemma 8.17.

Definition 8.18 (*c*-Dense Graph). A graph $G=(V,E)$ is said to be c-dense if $|E| \geq cn^2$ where c is a constant with $0 < c < \frac{1}{2}$.

Gaspers et al. have given an $\mathcal{O}(1.5063^{(\frac{1}{2}+\frac{1}{2}\sqrt{1-2c})n})$-time algorithm for Dominating Set on c-dense graphs [165]. We will improve this below. A graphical comparison of both results can be found in Figure 8.1. Note that if c is very small, then Theorem 5.1 can give a slightly faster algorithm.



The solid line represents the base of the exponent of the upper bound on the running time of our algorithm. The dashed line represents the base of the exponent of the upper bounds from [165].

Figure 8.1. Comparison of bounds on the running time on c-dense graphs.

Corollary 8.19. *The number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a c-dense graph can be counted in $\mathcal{O}(1.5002^{(\frac{1}{4}+\frac{1}{4}\sqrt{9-16c})n})$ time.*

Proof. By a counting argument in [165], any graph with sufficiently many edges has a set of high degree vertices that allow application of Lemma 8.17 with parameter t. For c-dense graphs this occurs when:

$$|E| \geq cn^2 \geq \frac{1}{2}\left(\frac{1}{2}t-1\right)(n-1) + \frac{1}{2}\left(n-\frac{1}{2}t+1\right)(t-3)$$

If $t \leq \frac{1}{2}(4+3n) - \frac{1}{2}\sqrt{-8n+9n^2-16cn^2}$, then this is the case. By taking t maximal in this inequality and removing all factors that disappear in the big-$\mathcal{O}$ notation, we obtain a running time of $\mathcal{O}(1.5002^{(\frac{1}{4}+\frac{1}{4}\sqrt{9-16c})n})$. □

Corollary 8.19 gives the currently fastest algorithm for Dominating Set on c-dense graphs.

We now proceed by giving faster algorithms on circle graphs, 4-chordal graphs, and weakly chordal graphs. We first define these graphs classes. We also define one additional graph class, namely chordal graphs, for which we will give a faster algorithm at the end of the section.

Definition 8.20 (Circle Graph). A *circle graph* is an intersection graph of chords in a circle: every vertex represents a chord, and vertices are adjacent if their corresponding chords intersect.

A *chordless cycle* in a graph G is a sequence of vertices $(v_1, v_2, \ldots, v_l, v_1)$ such that $G[\bigcup_{i=1}^{l}\{v_i\}]$ is an induced cycle in G, that is, the only edges of $G[\bigcup_{i=1}^{l}\{v_i\}]$ are the edges that form the cycle.

Definition 8.21 (Chordal Graph). A graph is *chordal* if it has no chordless cycle of length more than three.

Definition 8.22 (4-Chordal Graph). A graph is *4-chordal* if it has no chordless cycle of length more than four.

Definition 8.23 (Weakly Chordal Graph). A graph G is weakly chordal if both G and its complement are 4-chordal.

On these graph classes Gaspers et al. balance dynamic programming on tree decompositions to the many vertices of high degree approach [165]. For the dynamic programming, we need Theorem 11.7. This theorem states that we can count the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in G in $\mathcal{O}^*(3^k)$ time when G is given with a tree decomposition of width k.

The following lemma is based on [165] and gives our running times on circle graphs, 4-chordal graphs, and weakly chordal graphs. Note that a graph class $\mathcal{G}$ is a hereditary class if all induced subgraphs of any graph $G \in \mathcal{G}$ are in $\mathcal{G}$ also.

Lemma 8.24. *Let $\mathcal{G}$ be a hereditary class of graphs such that all $G \in \mathcal{G}$ have the properties that $\text{tw}(G) \leq c\Delta(G)$ and that a tree decomposition of G of width at most $c\Delta(G)$ can be computed in polynomial time. For any $t' \geq 1$, there exists an algorithm running in $\mathcal{O}(\max\{1.5002^{(1-\frac{1}{2}t')n}, 3^{(c+\frac{1}{2})t'n}\})$ time that counts the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a graph $G \in \mathcal{G}$.*

Proof. Let X be the set of vertices of degree at least $t'n$. If $|X| \geq \frac{1}{2}t'n$, then we can apply Lemma 8.17 with $t = t'n$ giving a running time of $\mathcal{O}(1.5002^{(1-\frac{1}{2}t')n})$.

Otherwise, $|X| < \frac{1}{2}t'n$. Since $G[V \setminus X]$ belongs to $\mathcal{G}$, we know that:

$$\text{tw}(G) \leq \text{tw}(G[V \setminus X]) + |X| \leq c\Delta(G[V \setminus X]) + |X| < ct'n + \frac{1}{2}t'n = \left(c + \frac{1}{2}\right)t'n$$

Note that the first inequality is based on the fact that we can add all vertices in X to all bags of a tree decomposition of $G[V \setminus X]$): this increases its width by at most $|X|$.

Now, we can apply the $\mathcal{O}^*(3^k)$-time algorithm of Theorem 11.7. This gives us a running time of $\mathcal{O}(3^{(c+\frac{1}{2})t'n})$. □

The running times follow from using the following values for c on the different graph classes.

Proposition 8.25 ([165]). *The following graph classes are hereditary graph classes with $\text{tw}(G) \leq c\Delta(G)$ for all $G \in \mathcal{G}$ using the indicated values of c:*

- *Weakly Chordal graphs ($c = 2$).*
- *4-Chordal graphs ($c = 3$).*
- *Circle graphs ($c = 4$).*

The corresponding tree decompositions can be computed in polynomial time.

Corollary 8.26. *There exist algorithms that count the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a weakly chordal graph in $\mathcal{O}(1.4590^n)$ time, in a 4-chordal graph in $\mathcal{O}(1.4700^n)$ time, and in a circle graph in time $\mathcal{O}(1.4764^n)$ time.*

Proof. Combine Lemma 8.24 and Proposition 8.25, and compute t' such that both running times in Lemma 8.24 are balanced. □

We conclude this section by considering chordal graphs. For this graph class, we cannot use the improvement of Lemma 8.17 over Proposition 8.16. That is, we do not benefit from the fact that we need only $\frac{1}{2}t$ vertices of degree at least $t-2$ instead of at least t vertices of degree at least $t-2$. However, we can improve the result of Gaspers et al. on this graph class by using both a faster branch-and-reduce algorithm that is used if the graph is dense (the algorithm of Theorem 8.13), and a faster tree-decomposition-based dynamic programming algorithm that is used if the graph is sparse. The faster tree-decomposition-based dynamic programming algorithm is given by Proposition 8.27 below.

We use that a chordal graph can be represented as a clique tree [168]. A *clique tree* T of a chordal graph G is a tree such that there exists a bijection between the nodes of the tree T and the maximal cliques in G and such that, for each vertex $v \in V$, all nodes in T corresponding to a clique that contains v form a connected subtree. It is well known that a clique tree of a chordal graph is an optimal tree decomposition of G whose width equals the size of the largest clique in G minus one.

Proposition 8.27. *There is an algorithm that, given a clique tree of a chordal graph G of treewidth k, computes the number of dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}^*(2^k)$ time.*

Proof. To prove this proposition, we combine the techniques that we will introduce in Chapter 11 to obtain faster algorithms on tree decompositions with an observation of Gasper et al. on solving Dominating Set on tree decompositions of chordal graphs [165]. For the proof below, we require the reader to be familiar with the details of the proof of Theorem 11.7.

In the algorithm in the proof of Theorem 11.7, a table A_x is computed for each node x in the nice tree decomposition T. Let X_x be the vertices in the bag associated with the node x in T. The table A_x has entries $A_x(c, \kappa)$ containing the number of partial solutions of Dominating Set of size exactly κ in G_x satisfying the requirements defined by the states in the at most 3^k colourings c of X_x using states 1, 0_0 and $0_?$. Recall from Table 11.1 that each table entry $A_x(c, \kappa)$ counts only partial solutions whose vertex set contains exactly those vertices from X_x with state 1 in c; also, it counts only partial solutions that do not yet dominate the vertices in X_x with state 0_0 in c; and, it is indifferent about whether vertices with state $0_?$ are dominated or not.

Translating the observation of Gaspers et al. [165] to this situation, we observe that if the tree decomposition that is given as input is a clique tree of a chordal graph, then, for each node x of the nice tree decomposition T, the vertices the bag X_x form a clique in G. Consequently, an entry in A_x corresponding to a colouring c that contains both the state 1 and the state 0_0 will always have the value zero. This is because no partial solution can exist that contains a vertex from X_x in the vertex set of the dominating set and still has that a vertex from X_x is undominated.

Consequently, the algorithm has to compute only the values $A_x(c, \kappa)$ for the at most $2 \cdot 2^k$ remaining colourings of X_x. It is not hard to verify that the dynamic programming recurrences in the proof of Theorem 11.7 can be used to compute all

these $\mathcal{O}^*(2^k)$ values in A_x for each type of node of T in $\mathcal{O}^*(2^k)$ time per node. In these computations, all values that have not been computed, i.e., entries $A_x(c, \kappa)$ with both a 1 and a 0_0 state in c, are considered to be zero. As a result, the modified algorithms runs in $\mathcal{O}^*(2^k)$ time. □

Our result on chordal graphs now follows.

Corollary 8.28. *There exists an algorithm that counts the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, in a chordal graph in $\mathcal{O}(1.3687^n)$ time.*

Proof. Compute a clique tree T of the chordal graph G; this can be done in polynomial time [293]. Find a largest clique C in G by considering the cliques corresponding to each node of T.

If $|C| \geq 0.452704n$, then there exists at least $0.452704n$ vertices of degree at least $0.452704n - 1$. Hence, we can apply Lemma 8.17 using $t = 0.452704n$ and solve the instance in $\mathcal{O}(1.5002^{n-\frac{1}{2}0.452704n}) = \mathcal{O}(1.3687^n)$ time. Otherwise, $|C| < 0.452704n$. In this case, T is a tree decomposition of width at most $0.452704n$. Now, we solve the instance in $\mathcal{O}(2^{0.452704n}) = \mathcal{O}(1.3687^n)$ time using Proposition 8.27. □

8.7. Concluding Remarks

In this chapter, we have shown that the principle of inclusion/exclusion can be used as a branching rule in a branch-and-reduce algorithm. We combined the use of such an inclusion/exclusion-based branching rule with different series of reduction rules and standard (non-inclusion/exclusion-based) branching rules. This resulted in non-trivial branch-and-reduce algorithms that we analysed using measure and conquer. In this way, we obtained, amongst others, the currently fastest polynomial and exponential space algorithms for #Dominating Set, the currently fastest polynomial space algorithm for Domatic Number, the currently fastest algorithm for Red-Blue Dominating Set, and the currently fastest algorithms for Dominating Set on some graph classes.

We note that our approach has further applications. Namely, any inclusion/exclusion algorithm (for example those in Section 2.3) can be turned into a branch-and-reduce algorithm without reduction rules. For example, the algorithm for #Perfect Matching in Corollary 2.20 is essentially an algorithm that counts set covers on instances with m sets of size two and n elements. One can easily add reduction rules similar to those used in this chapter to this algorithm. The resulting algorithm will possibly not have a faster worst-case running time, but will certainly generate less subproblems on a given instance possibly improving its running time in practice.

There are more techniques to prove good upper bounds on branch-and-reduce algorithms, for example techniques based on average degrees in a graphs; see for example [59]. Our branch-and-reduce algorithms can also be analysed by such means. The general idea of interpreting an inclusion/exclusion algorithm as branch-and-reduce algorithm is a nice approach to get better upper bounds on the running time of an inclusion/exclusion algorithm than the usual running times of $\mathcal{O}^*(2^n)$; an upper bound on the running time present in, for example, all examples in Section 2.3.

We conclude with an observation on the currently fastest algorithm for Red-Blue Dominating Set given in this chapter (Corollary 8.15). Notice that the currently fastest algorithm for Dominating Set (Theorem 5.1) does not use inclusion/exclusion-based branching and is based on an algorithm for Set Cover. Also notice that the same set-cover-based algorithm is used to give the currently fastest polynomial-space algorithm for Red-Blue Dominating Set (Corollary 5.5). Although this set-cover-based algorithm is faster for Dominating Set, it is slower for Red-Blue Dominating Set when compared to the results in this chapter. That is, the exponential-space algorithm for Red-Blue Dominating Set of Corollary 8.15 is faster than the one of Corollary 5.5, while the underlying algorithm that counts set covers is slower when used for Dominating Set.

This difference can be explained from the different functions that are minimised in the two measure-and-conquer analyses. When minimising $\alpha^{(v_{\max}+w_{\max})n}$, as for Dominating Set, different values of $v_{\max}$ and $w_{\max}$ can be used while still giving good running times. The fact that these values can be different can be beneficial if the sets of reduction rules for sets is very different from the set of reduction rules for elements. When minimising $\alpha^{\max\{v_{\max},w_{\max}\}n}$, as for Red-Blue Dominating Set, a lower value for either $v_{\max}$ or $w_{\max}$ does not directly give a better bound on the running time. In this case, we can benefit only little from the asymmetry. The extra power given by the inclusion/exclusion-based branching rule now leads to a faster algorithm.

9

Inclusion/Exclusion Branching for Partial Requirements: Partial Dominating Set and *k*-Set Splitting

In the previous chapter, we have introduced *inclusion/exclusion-based branching*. We used branching rules based on the principle of inclusion/exclusion to construct algorithms that count the number of set covers of each size κ with $0 \leq \kappa \leq n$. This led to faster algorithms for #Dominating Set and several related problems. In a more abstract sense where we speak of requirements instead of covering elements or dominating vertices, we can say that these algorithms count the number of solutions of each size that satisfy all requirements that are imposed by their respective problems.

In this chapter, we continue studying this approach in a slightly different setting: we will count the number of solutions that satisfy exactly (or at least) t requirements. E.g., we will count the number of partial set covers of each size κ that cover exactly t elements. In this setting, we introduce a new inclusion/exclusion-based branching rule that we call *extended inclusion/exclusion-based branching* or simply *extended IE-branching*. We give two applications of this branching rule. First, we consider Partial Dominating Set. This is a natural extension of Dominating Set where we are asked to compute a vertex set of minimum size that dominates at least t vertices. Second, we consider the parameterised problem k-Set Splitting. For both problems, we will present new exact algorithms that improve upon previous results.

This chapter is organised in the following way. We introduce extended inclusion/exclusion-based branching in Section 9.1. This branching rule will then be used in an

[†]This chapter is joint work with Jesper Nederlof. The chapter contains results of which a preliminary version has been presented at the 5th International Symposium on Parameterized and Exact Computation (IPEC 2010) [245]. The paper received the 'Excellent Student Paper Award' at this conference.

algorithm for Partial Dominating Set in Section 9.2. In this section, we first show that the usual set cover modelling of Partial Dominating Set equals the Partial Red-Blue Dominating Set problem. This will be followed by a result showing that the counting variant of this problem has some interesting symmetry properties. We conclude the section by giving faster polynomial-space and exponential-space algorithms for Partial Dominating Set. In Section 9.3, we give another application of the extended inclusion/exclusion-based branching by giving a faster algorithm for the parameterised problems k-Set Splitting and k-Not-All-Equal Satisfiability. Finally, we give some concluding remarks in Section 9.4

9.1. Extended Inclusion/Exclusion Branching

We start by introducing our extended inclusion/exclusion-based branching rule. We will do so in a slightly more formal setting compared to the introduction of inclusion/exclusion-based branching in Section 8.1.

Let $\mathcal{A}$ be a set and let $\mathcal{P}_1, \mathcal{P}_2 \ldots, \mathcal{P}_n$ be subsets of $\mathcal{A}$. In the context of this chapter, the sets $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ can be thought of as properties. These properties can represent parts of a problem instance. For an example, consider the Set Cover problem; this example will be analogous to the setting in which we introduced inclusion/exclusion-based branching in Section 8.1. Given a Set Cover instance $(\mathcal{S}, \mathcal{U})$ where $\mathcal{S}$ is a collection of sets over the universe $\mathcal{U}$, we let $\mathcal{A}$ contain all possible collections of sets from $\mathcal{S}$, ie.e., $\mathcal{A} = 2^{\mathcal{S}}$. Here, we have a property $\mathcal{P}_S$ for every set $S \in \mathcal{S}$ and a property $\mathcal{P}_e$ for every element $e \in \mathcal{U}$. For a property associated with a set $S \in \mathcal{S}$, we let $\mathcal{P}_S$ contain the collections of sets in $\mathcal{A}$ that contain the set S, i.e., $\mathcal{P}_S = \{\mathcal{C} \in \mathcal{A} \mid S \in \mathcal{C}\}$. For a property associated with an element $e \in \mathcal{U}$, we let $\mathcal{P}_e$ contain the collections of sets in $\mathcal{A}$ that cover e, i.e., $\mathcal{P}_S = \{\mathcal{C} \in \mathcal{A} \mid e \in \bigcup_{S \in \mathcal{C}} S\}$.

Given $\mathcal{A}$ and a series of properties $\mathcal{P}_1, \mathcal{P}_2 \ldots, \mathcal{P}_n$, we let a partitioning (R, O, F) of $\{1, 2, \ldots, n\}$ define whether a property $\mathcal{P}_i$ is a *required property* ($i \in R$), an *optional property* ($i \in O$), or a *forbidden property* ($i \in F$).

Following Bax [14], we define for a partitioning (R, O, F) of $\{1, 2, \ldots, n\}$:

$$a(R, O, F) = \left| \left(\bigcap_{i \in R} \mathcal{P}_i \right) \setminus \left(\bigcup_{i \in F} \mathcal{P}_i \right) \right|$$

That is, $a(R, O, F)$ counts the number of $\mathcal{C} \in \mathcal{A}$ that have all the required properties ($\mathcal{C} \in \mathcal{P}_i$ is counted when $i \in R$), and none of the forbidden properties ($\mathcal{C} \in \mathcal{P}_i$ is not counted when $i \in F$). Since (R, O, F) partitions $\{1, 2, \ldots, n\}$, all optional properties are ignored, i.e., a $\mathcal{C} \in \mathcal{A}$ is counted both if $\mathcal{C} \in \mathcal{P}_i$ and if $\mathcal{C} \notin \mathcal{P}_i$, for any $i \in O$.

Recall that in a Set Cover instance $(\mathcal{S}, \mathcal{U})$, initially, every set $S \in \mathcal{S}$ corresponds to an optional property since it can either be taken in a solution or not, and that every element $e \in \mathcal{U}$ corresponds to a required property since it must be covered. In this initial case, $a(R, O, F)$ counts the number of set covers of $(\mathcal{S}, \mathcal{U})$.

Through branching, the role of a property can change between being required, optional, and forbidden. It is easy to see that the branching rules that are based on branching on an optional property (a set $S \in \mathcal{S}$) or a required property (an element

$e \in \mathcal{U}$) correspond to the following formulas:

$$\begin{array}{lrcl} \text{OPTIONAL:} & a(R, O \cup \{i\}, F) & = & a(R \cup \{i\}, O, F) + a(R, O, F \cup \{i\}) \\ \text{REQUIRED:} & a(R \cup \{i\}, O, F) & = & a(R, O \cup \{i\}, F) - a(R, O, F \cup \{i\}) \end{array}$$

In this setting, $a(R, O, F)$ counts the number of covers $\mathcal{C} \subseteq \mathcal{S}$ that satisfy the following properties:

- for an element $e \in \mathcal{U}$, $\mathcal{C}$ covers e if $\mathcal{P}_e$ is associated with R, $\mathcal{C}$ does not cover e if $\mathcal{P}_e$ is associated with F, and $\mathcal{C}$ is indifferent about covering e if $\mathcal{P}_e$ is associated with O.
- for a set $S \in \mathcal{S}$, $\mathcal{C}$ contains S if $\mathcal{P}_S$ is associated with R, $\mathcal{C}$ does not contain S if $\mathcal{P}_S$ is associated with F, and $\mathcal{C}$ may contain S if $\mathcal{P}_S$ is associated with O.

Let us now extend these ideas to the setting of this chapter. Here, we are interested in counting all $\mathcal{C} \in \mathcal{A}$ that have *exactly* t of the properties associated with R. In a similar way as above, we define:

$$a_t(R, O, F) = |\{\mathcal{C} \in \mathcal{A} \ : \ |R[\mathcal{C}]| = t \wedge |F[\mathcal{C}]| = 0\}|$$

where we denote $R[\mathcal{C}] = \{i \in R \mid \mathcal{C} \in \mathcal{P}_i\}$ and $F[\mathcal{C}] = \{i \in F \mid \mathcal{C} \in \mathcal{P}_i\}$. We can say that $\mathcal{C} \in \mathcal{A}$ is counted in $a_t(R, O, F)$ if it is counted in $a(R, O, F)$ and satisfies exactly t of the required properties. Notice that if we now consider a required property $\mathcal{P}_i$, then $\mathcal{C} \in \mathcal{A}$ can be counted in $a_t(R, O, F)$ both when the required property is satisfied and when the required property is not satisfied. This is so because $a_t(R, O, F)$ counts all $\mathcal{C} \in \mathcal{A}$ that satisfy exactly t of the properties in R instead of all these properties.

We can branch on the choice whether a required property (now using the parameter t) will be satisfied or not. This leads to the following recurrence:

$$a_t(R \cup \{i\}, O, F) = a_t(R, O, F \cup \{i\}) \ + |\{\mathcal{C} \in \mathcal{P}_i \ : \ |R[\mathcal{C}]| = t \wedge |F[\mathcal{C}]| = 0\}|$$

Note that the second term on the right hand side represents the number of $\mathcal{C} \in \mathcal{A}$ counted in $a_t(R \cup \{i\}, O, F)$ that also satisfy the property $\mathcal{P}_i$.

Since any $\mathcal{C} \in \mathcal{A}$ counted in the second term on the right hand side must be in $\mathcal{P}_i$, we can now apply the familiar inclusion/exclusion-based branching rule from Section 8.1 to the required property $\mathcal{P}_i$. That is, the number of $\mathcal{C} \in \mathcal{A}$ counted in $a_t(R, O, F)$ that are also in $\mathcal{P}_i$, equals the total number of $\mathcal{C} \in \mathcal{A}$ counted in $a_t(R, O, F)$, minus those that are not in $\mathcal{P}_i$. The composition of these two branching rules results in the extended inclusion/exclusion-based branching rule that we will use in this chapter:

$$\begin{aligned} a_t(R \cup \{i\}, O, F) = a_t(R, O, F \cup \{i\}) \ + \\ (a_{t-1}(R, O \cup \{i\}, F) - a_{t-1}(R, O, F \cup \{i\})) \end{aligned}$$

where we set $a_t(R, O, F) = 0$ if $t > |R|$ or $t < 0$. The parameter t is decreased in the last two terms on the right hand side because the subtraction guarantees that the requirement $\mathcal{P}_i$ will be satisfied; therefore, we need to satisfy $t - 1$ remaining requirements from R. In the case that we no longer have the possibility to choose to satisfy a requirement or not, because $t = 0$ or $t = |R|$, then the rule is correct because we set $a_t(R, O, F) = 0$ for any $t < 0$ or $t > |R|$.

At first, the above branching rule does not look particularly efficient since the branch-and-reduce algorithm has to solve three instances recursively for each application of the branching rule. However, we will now show that two recursive calls suffice. Observe that the first and last recursive call differ only in the subscript parameter t. We can exploit this by computing a_s for all $0 \leq s \leq n$ simultaneously whenever we need to compute a_t. Although this will slow down the algorithm by only a factor n, this allows us to consider only the cases $a_t(R, O, F \cup \{i\})$ and $a_{t-1}(R, O \cup \{i\}, F)$ since $a_{t-1}(R, O, F \cup \{i\})$ will be computed when the algorithm computes $a_t(R, O, F \cup \{i\})$.

If one expands the recurrence that defines the extended inclusion/exclusion-based branching rule, then the following natural variation on the inclusion/exclusion formula can be obtained for computing $a_t(R, O, F)$:

$$a_t(R, O, F) = \sum_{X \subseteq R} (-1)^{|X|-|R|+t} \binom{|X|}{|R|-t} a_0(\emptyset, O \cup (R \setminus X), F \cup X)$$

To see this, consider the branching tree (search tree) after exhaustively applying the extended inclusion/exclusion-based branching rule. Let X be a set of forbidden properties, i.e., the set of properties that go into the first or third branch when branching. We consider the set of leaves of this tree where X is the set of forbidden properties. For each such leaf, we have lowered the parameter t exactly t times, thus we have taken $|R| - t$ times the first branch and $|X| - |R| + t$ times the second branch. As a result, we have $\binom{|X|}{|R|-t}$ such leaves, and the contribution of each leaf to the sum in the root is multiplied exactly $|X| - |R| + t$ times by -1.

9.2. Exact Algorithms for Partial Dominating Set

As a first application of our new branching rule, we will give faster algorithms for Partial Dominating Set. This problem is a natural extension of Dominating Set where we need to dominate only a given number of vertices. In this section, we give an $\mathcal{O}(1.5673^n)$-time algorithm for this problem that uses polynomial space and an $\mathcal{O}(1.5014^n)$-time algorithm that uses exponential space.

Let us first define the problem formally. Any vertex set $D \subseteq V$ is a *partial dominating set*, and such a partial dominating set D dominates all vertices in $N[D]$.

Partial Dominating Set
Input: A graph $G = (V, E)$, an integer $t \in \mathbb{N}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a partial dominating set $D \subseteq V$ in G of size at most k that dominates at least t vertices?

Let us first consider some previous results on Partial Dominating Set. The previously fastest exact exponential-time algorithm for this problem is due to Liedloff [223]; this algorithm runs in $\mathcal{O}(1.6183^n)$ time and polynomial space. When considering the parameterised version of the problem, parameterised by the parameter t, Kneis et al. have shown that the problem is Fixed-Parameter Tractable [205]. The fastest known parameterised algorithm is due to Koutis and Williams and runs in $\mathcal{O}^*(2^t)$ time [213]. When parameterised by the size of the partial dominating set and restricted to planar

graphs, Amin et al. have shown that the problem is Fixed-Parameter Tractable [4], and Fomin et al. have given a subexponential-time algorithm [152].

The rest of this section is divided into three parts. First, we give a transformation from Partial Dominating Set to Partial Red-Blue Dominating Set in Section 9.2.1 and show that this problem has an interesting symmetry relation between the red and the blue vertices. Secondly, we give our polynomial-space algorithm for Partial Dominating Set in Section 9.2.2. We conclude by giving our exponential-space algorithm for this problem in Section 9.2.3.

9.2.1. Symmetry in the Partial Red-Blue Dominating Set Problem

The algorithms for Partial Dominating Set in this chapter are based on algorithms for a variant of Set Cover on the instance $(\mathcal{S}, \mathcal{U})$ where $\mathcal{S} = \{N[v] \mid v \in V\}$ and $\mathcal{U} = V$. This is similar to the approach in Chapters 5 and 8 where we considered different variants of the Dominating Set problem.

In this chapter, we will mainly consider the incidence graph I of $(\mathcal{S}, \mathcal{U})$ as defined in Section 8.1. On this incidence graph, the problem is equivalent to Partial Red-Blue Dominating Set where the red vertices are the vertices representing the sets and the blue vertices are the vertices representing the elements. We note that any vertex set $D \subseteq \mathcal{R}$ in a bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with a red partition $\mathcal{R}$ and blue partition $\mathcal{B}$ is a *partial red-blue dominating set*, and such a partial red-blue dominating set D dominates all vertices in $N(D)$.

Partial Red-Blue Dominating Set
Input: A bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertices $\mathcal{R}$ and blue vertices $\mathcal{B}$, an integer $t \in \mathbb{N}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a partial red-blue dominating set $D \subseteq \mathcal{R}$ in G of size at most k that dominates at least t vertices in $\mathcal{B}$?

Note that we obtain the Red-Blue Dominating Set problem when we take $t = |\mathcal{B}|$.

Given a graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertex set $\mathcal{R}$ and blue vertex set $\mathcal{B}$, we define for any $0 \leq i \leq |\mathcal{R}|$ and $0 \leq j \leq |\mathcal{B}|$:

$$b_{i,j}(\mathcal{R}, \mathcal{B}) = |\{X \subseteq \mathcal{R} : |X| = i \wedge |N[X] \cap \mathcal{B}| = j\}|$$

In words, $b_{i,j}(\mathcal{R}, \mathcal{B})$ is the number of partial red-blue dominating sets of size i that dominate exactly j blue vertices. Clearly, an instance of Partial Red-Blue Dominating Set is a Yes-instance if and only if there exist $i \leq k$ and $j \geq t$ such that $b_{i,j}(\mathcal{R}, \mathcal{B}) > 0$.

The algorithms in this chapter will compute, for a given graph $G = (\mathcal{R} \cup \mathcal{B}, E)$, a matrix M_G with $b_{i,j}(\mathcal{R}, \mathcal{B})$ for all $0 \leq i \leq |\mathcal{R}|$ and $0 \leq j \leq |\mathcal{B}|$. There is a lot of symmetry present in the relation between the two colour classes in this approach. The remainder of this section will be used to illustrate this symmetry relation and give a nice property of it. This property is given by Proposition 9.1.

Given a graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertex set $\mathcal{R}$ and blue vertex set $\mathcal{B}$, we define its *flipped graph* G' to be G with the colours of every vertex flipped between red and blue, i,e., G' is the graph G with red vertex set $\mathcal{B}$ and blue vertex set $\mathcal{R}$. We now show that the matrix M_G that contains the values $b_{i,j}(\mathcal{R}, \mathcal{B})$, for all i and j, associated

with the graph G can be transformed into the matrix $M_{G'}$ associated with the flipped graph G' in polynomial time.

Proposition 9.1. *Given a graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertex set $\mathcal{R}$ and blue vertex set $\mathcal{B}$, let the values $b_{i,j}(\mathcal{R}, \mathcal{B})$ associated with G for $0 \leq i \leq |\mathcal{R}|$ and $0 \leq j \leq |\mathcal{B}|$ be as defined above, and let $b_{j,i}(\mathcal{B}, \mathcal{R})$ be the same values associated to the flipped graph G' of G. We have that:*

$$b_{i,j}(\mathcal{R}, \mathcal{B}) = \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} (-1)^{l+j-|\mathcal{B}|} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} b_{l,k}(\mathcal{B}, \mathcal{R})$$

Proof. In section 9.1, we gave the following formula obtained by expanding the recurrence of the extended inclusion/exclusion-based branching rule:

$$a_t(R, O, F) = \sum_{X \subseteq R} (-1)^{|X|-|R|+t} \binom{|X|}{|R| - t} a_0(\emptyset, O \cup (R \setminus X), F \cup X)$$

We will give a similar formula here, where $a_t(R, O, F)$ is replaced by $b_{i,j}(\mathcal{R}, \mathcal{B})$. Note that $\mathcal{B}$ now corresponds to the set of properties that initially are required properties. If we choose to make a subset $X \subseteq \mathcal{B}$ of these properties forbidden, then $\mathcal{R}$ is influenced by this choice of X as no subsets of $\mathcal{R}$ containing a vertex that is a neighbour of a forbidden vertex may be selected. In this way, we obtain the following formula summing over all sets of vertices $X \subseteq \mathcal{B}$ that correspond to the blue vertices that we forbid to be dominated:

$$\begin{aligned} b_{i,j}(\mathcal{R}, \mathcal{B}) &= \sum_{X \subseteq \mathcal{B}} (-1)^{|X|-|\mathcal{B}|+j} \binom{|X|}{|\mathcal{B}| - j} b_{i,0}(\mathcal{R} \setminus N(X), \emptyset) \\ &= \sum_{X \subseteq \mathcal{B}} (-1)^{|X|-|\mathcal{B}|+j} \binom{|X|}{|\mathcal{B}| - j} \binom{|\mathcal{R} \setminus N(X)|}{i} \end{aligned}$$

The second equality here follows from the definition of $b_{i,j}(\mathcal{R}, \mathcal{B})$: $\binom{|\mathcal{R} \setminus N(X)|}{i}$ equals the number of ways to choose i red vertices that have no forbidden neighbours.

If we now group the summands of the summation by the size of $\mathcal{R} \cap N(X)$ and the size of X, then we obtain the following equation:

$$\begin{aligned} b_{i,j}(\mathcal{R}, \mathcal{B}) &= \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} \sum_{\substack{X \subseteq \mathcal{B} \\ |\mathcal{R} \cap N(X)| = k, |X| = l}} (-1)^{l-|\mathcal{B}|+j} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} \\ &= \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} (-1)^{l-|\mathcal{B}|+j} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} \sum_{\substack{X \subseteq \mathcal{B} \\ |\mathcal{R} \cap N(X)| = k, |X| = l}} 1 \\ &= \sum_{k=0}^{|\mathcal{R}|} \sum_{l=0}^{|\mathcal{B}|} (-1)^{l+j-|\mathcal{B}|} \binom{l}{|\mathcal{B}| - j} \binom{|\mathcal{R}| - k}{i} b_{l,k}(\mathcal{B}, \mathcal{R}) \end{aligned}$$

The last equality follows again from the definition of $b_{j,i}(\mathcal{B}, \mathcal{R})$. □

This result is stated here mainly for aesthetic reasons: we will not use it in any of our algorithms. The symmetry, however, can be found in our algorithms. Namely, the presented algorithms will treat sets and elements (red and blue vertices) completely symmetric. This is different from, for example, Algorithm 8.2.

9.2.2. A Polynomial-Space Algorithm for Partial Dominating Set

We will now give a small modification of the polynomial-space algorithm for #Dominating Set in Section 8.3 to obtain a polynomial-space algorithm for Partial Dominating Set. The resulting algorithm computes the number of vertex sets of size κ that dominate exactly t vertices for each $0 \leq \kappa \leq n$ and $0 \leq t \leq n$. A curious fact is that the new algorithm solves a problem that is a more general problem (which is mostly studied separately) than #Dominating Set within the same running time as the algorithm for #Dominating Set, up to a polynomial (linear) factor.

Let us first consider the effect of the extended inclusion/exclusion-based branching rule when applied to compute the values $b_{i,j}(\mathcal{R}, \mathcal{B})$ defined in Section 9.2.1. Using the definition of the extended inclusion/exclusion-based branching rule, we directly obtain the following recurrence where $b_{i,j}(\mathcal{R}, \mathcal{B}) = 0$ if $j < 0$ or $j > |\mathcal{B}|$:

$$\begin{aligned} b_{i,j}(\mathcal{R}, \mathcal{B}) = {} & b_{i,j}(\mathcal{R} \setminus N(v), \mathcal{B} \setminus \{v\}) + \\ & (b_{i,j-1}(\mathcal{R}, \mathcal{B} \setminus \{v\}) - b_{i,j-1}(\mathcal{R} \setminus N(v), \mathcal{B} \setminus \{v\})) \end{aligned}$$

Theorem 9.2. *There exists an algorithm that solves* Partial Dominating Set *in* $\mathcal{O}(1.5673^n)$ *time and polynomial space.*

Proof. Consider the algorithm used in the proof of Theorem 8.5. This algorithm counts the number of dominating sets of each size κ with $0 \leq \kappa \leq n$. From an instance of #Dominating Set, the algorithm constructs the incidence graph of the related problem of counting the number of set covers of size κ and then applies Algorithm 8.1.

We will construct a very similar algorithm for Partial Dominating Set. Our algorithm will count the number of vertex sets of size κ that dominate exactly t vertices for each $0 \leq \kappa \leq n$ and $0 \leq t \leq n$. It will do so by counting the number of partial red-blue dominating sets of size κ that dominate exactly t vertices on the incidence graph I of the related variant of Set Cover, for each κ and t. This counting is done by a modification of Algorithm 8.1 applied to the incidence graph I: Algorithm 9.1.

First, notice that the annotation procedure of Algorithm 9.1 is identical to the same procedure in Algorithm 8.1. Because the annotations do not influence the counting, we can treat them in the same way as in Algorithm 8.1. Hence, any graph given to the procedure CPSC-DP has treewidth at most two by Lemma 8.3, and the annotations do not influence the correctness of the branching rules. As a result, CPSC-DP can be implemented in polynomial time using a standard dynamic programming algorithm on tree decompositions. For an idea of how such a dynamic programming algorithm computes the required values, see Chapter 11.

Next, consider the branching rules of Algorithm 9.1. When branching on a red vertex, the branching rule adds the numbers of partial red-blue dominating sets that contain the red vertex to the number of such sets that do not contain the vertex. The algorithm does so for all κ and t by adding up the two corresponding matrices after

Algorithm 9.1. An algorithm counting the number of partial red-blue dominating sets.

Input: a bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ and a set of annotated vertices A

Output: a matrix containing the number of partial red-blue dominating sets of size κ dominating exactly t blue vertices for each $0 \leq \kappa \leq |\mathcal{R}|$ and $0 \leq t \leq |\mathcal{B}|$

CPSC(G, A):

if there exists a vertex $v \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ of degree at most one in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ **then**
  return CPSC$(G, A \cup \{v\})$
else if there exist two vertices $v_1, v_2 \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ both of degree two in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ that have the same two neighbours **then**
  return CPSC$(G, A \cup \{v_1\})$
else
  Let $r \in \mathcal{R} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)$ is maximal
  Let $b \in \mathcal{B} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ is maximal
  if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) \leq 2$ **and** $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b) \leq 2$ **then**
    return CPSC-DP(G)
  else if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) > d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ **then**
    Let $M_{\text{take}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus N[r]], A \setminus N(r))$
    Let $M_{\text{discard}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus \{r\}], A)$
    Increase the cardinalities κ in M_{take} by one
    return $M_{\text{take}} + M_{\text{discard}}$
  else
    Let $M_{\text{optional}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b\}], A)$
    Let $M_{\text{not}} = M_{\text{forbidden}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus N[b]], A \setminus N(e))$
    Decrease the parameter t by one in M_{optional} and $M_{\text{forbidden}}$
    return $M_{\text{not}} + M_{\text{optional}} - M_{\text{forbidden}}$

shifting the values in M_{take} in order to take into account the fact that we have taken a red vertex. This is very similar to Algorithm 8.1. When branching on a blue vertex, the algorithm uses the extended inclusion/exclusion-based branching rule. The resulting matrix with the required values is computed through matrix additions and subtractions. Note that the extended inclusion/exclusion-based branching rule generates the same induced subgraphs as subproblems as the branching rule of Algorithm 8.1 does.

Because both modified branching rules generate the same subproblems as Algorithm 8.1, the same branching tree emerges from applying either algorithm. Therefore, we conclude that our algorithm for Partial Dominating Set runs in $\mathcal{O}(1.5673^n)$ time by the proof of Theorem 8.5. □

We now argue that we can also use the algorithm of Theorem 9.2 to construct a partial dominating set as follows. After finding the size of a minimum partial dominating set, select a vertex and put it in the partial dominating set. Then, repeat the algorithm on the corresponding Partial Red-Blue Dominating Set instance where the red vertex and all neighbouring blue vertices corresponding to the selected vertex are removed and where we update κ and t to account for the fact that we have taken this vertex in a solution. If this does not increase the size of the minimum size set that dominates at least t vertices, then repeat this approach by selecting the next vertex. Otherwise, the size of the minimum size set that dominates at least t vertices has increased. In this case, we conclude that the selected vertex is not part of the solution. Then, one puts the blue vertices back in the graph, but not the red vertex as we will no longer use it, and again update κ and t. Hereafter, we continue by selecting the next red vertex. By repeating this process, we construct a solution while using the algorithm a linear number of times.

9.2.3. An Exponential-Space Algorithm for Partial Dominating Set

We can improve the running time of the algorithm in Theorem 9.2 at the cost of using exponential space in a way similar to our improvement to the algorithm for #Dominating Set in Section 8.5. We do so by letting the algorithm that counts partial red-blue dominating sets (Algorithm 9.1) used in Theorem 9.2 switch to a dynamic-programming-based approach when the maximum degree in the incidence graph is low, yet larger than two. As a result, we will obtain an algorithm for Partial Dominating Set using $\mathcal{O}(1.5014^n)$ time and space.

We note that the algorithm is only slightly slower than the exponential-space algorithm for #Dominating Set in Section 8.5 while solving a more general counting problem. The main cause of the difference in running times is the fact that one reduction rule (subsumption) is no longer applicable in the new setting.

Consider Algorithm 9.1 applied to bipartite graphs G with red vertex partition $\mathcal{R}$ and blue vertex partition $\mathcal{B}$. We modify this algorithm such that it uses dynamic programming on a tree decomposition of G at an earlier stage; as a result, we obtain Algorithm 9.2. More precisely, we let Algorithm 9.2 switch to this dynamic programming approach when G is of maximum degree four and has no blue vertices in $\mathcal{B}$ that have a degree four neighbour in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ nor blue vertices in $\mathcal{B}$ that have four degree three neighbours in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$.

The rest of this section is very similar to the analysis of Algorithm 8.2 in Section 8.5.

Algorithm 9.2. An exponential-space algorithm counting partial red-blue dominating sets.

Input: a bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ and a set of annotated vertices A

Output: a matrix containing the number of partial red-blue dominating sets of size κ dominating exactly t blue vertices for each $0 \leq \kappa \leq |\mathcal{R}|$ and $0 \leq t \leq |\mathcal{B}|$

CPSC(G, A):

if there exists a vertex $v \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ of degree at most one in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ **then**
  return CPSC$(G, A \cup \{v\})$
else if there exist two vertices $v_1, v_2 \in (\mathcal{R} \cup \mathcal{B}) \setminus A$ both of degree two in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ that have the same two neighbours **then**
  return CPSC$(G, A \cup \{v_1\})$
else
  Let $r \in \mathcal{R} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r)$ is maximal
  Let $b \in \mathcal{B} \setminus A$ be a vertex such that $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ is maximal
  if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) \leq 4$ **and** $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b) \leq 4$ **and** there exist no vertex $b \in \mathcal{B}$ with a neighbour of degree four in $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ **and** there exist no vertex $b \in \mathcal{B}$ with four neighbours of degree three **then**
    return CPSC-DP(G)
  else if $d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r) > d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)$ **then**
    Let $M_{\text{take}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus N[r]], A \setminus N(r))$
    Let $M_{\text{discard}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus \{r\}], A)$
    Increase the cardinalities κ in M_{take} by one
    return $M_{\text{take}} + M_{\text{discard}}$
  else
    Let $M_{\text{optional}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus \{b\}], A)$
    Let $M_{\text{not}} = M_{\text{forbidden}} = \text{CPSC}(G[(\mathcal{R} \cup \mathcal{B}) \setminus N[b]], A \setminus N(e))$
    Decrease the parameter t by one in M_{optional} and $M_{\text{forbidden}}$
    return $M_{\text{not}} + M_{\text{optional}} - M_{\text{forbidden}}$

We first give the measure and weight functions used to analyse the algorithm. Then, we prove an upper bound on the treewidth of the graph of any subproblem to which we apply the dynamic programming phase of the algorithm. Finally, we prove the running time of $\mathcal{O}(1.5014^n)$ in Theorem 9.4.

To prove the bound on the running time, we again use measure and conquer [144] (see Section 5.2). To this end, we use the measure k used in the proof of Theorem 8.5 (implicitly also in Theorem 9.2) with different weights:

$$k := \sum_{b \in \mathcal{B}, b \notin A} v(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(b)) + \sum_{r \in \mathcal{R}, r \notin A} w(d_{(\mathcal{R} \cup \mathcal{B}) \setminus A}(r))$$

In this case, we use the following weight functions:

i	2	3	4	5	6	> 6
$v(i)$	0.498964	0.750629	0.913191	0.975726	0.999999	1.000000
$w(i)$	0.673168	1.046831	1.261070	1.352496	1.382025	1.386987

Now, we will give a bound on the treewidth of any generated subproblem that will be solved by dynamic programming by Algorithm 9.2.

Lemma 9.3. *For any $\epsilon > 0$, there exists an integer n_ϵ such that the following holds for any graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ associated with a subproblem of measure at most k that Algorithm 9.2 solves by dynamic programming: if $|\mathcal{R} \cup \mathcal{B}| > n_\epsilon$, then the treewidth of G is at most $(0.245614 + \epsilon)k$. A tree decomposition of this width can be computed in polynomial time.*

Proof. Let A be the set of annotated vertices in a subproblem with bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$. Furthermore, let $\epsilon > 0$ be fixed, and let n_ϵ be as in Proposition 2.16. Below, we will show that the pathwidth of $G[(\mathcal{R} \cup \mathcal{B}) \setminus A]$ is at most $(0.245614 + \epsilon)k$ and that a path decomposition of this width can be obtained in polynomial time using Proposition 2.16. The result then follows because a path decomposition also is a tree decomposition, and because we can add all vertices in A without increasing the treewidth by the argument in the proof of Lemma 8.3.

In a way similar to the proof of Lemma 8.12 (and [138]), we construct a linear program that computes the maximum width of a tree decomposition of G obtained in the above way. In this linear program, all variables have the domain $[0, \infty)$.

$$\begin{aligned}
\max \quad z &= \frac{1}{6}(x_3 + y_3) + \frac{1}{3}(x_4 + y_4) \qquad \text{such that:} \\
1 &= \sum_{i=2}^{4} w(i)x_i + \sum_{i=2}^{4} v(i)y_i && (9.1) \\
\sum_{i=2}^{4} i x_i &= \sum_{i=2}^{4} i y_i && (9.2) \\
y_4 &= p_0 + p_1 + p_2 + p_3 && (9.3) \\
x_2 &\geq \frac{4}{2}p_0 + \frac{3}{2}p_1 + \frac{2}{2}p_2 + \frac{1}{2}p_3 && (9.4) \\
x_3 &\geq \frac{1}{3}p_1 + \frac{2}{3}p_2 + \frac{3}{3}p_3 && (9.5)
\end{aligned}$$

The objective function represents the maximum pathwidth z of the graph $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$ per unit of the measure obtained using Proposition 2.16. Notice that, in a subproblem on which we start dynamic programming, $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$ can have vertices of degree at most four. The variables x_i and y_i represent the number of red vertices of degree i and blue vertices of degree i per unit of the measure in $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$, respectively. Since blue vertices of degree four can exist only if their neighbours have specific combinations of degrees, we introduce the variables p_i. Such a variable p_i represents the number of blue vertices of degree four that have i neighbours of degree three and $4-i$ neighbours of degree two per unit of the measure in $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$. Form the branching rules of Algorithm 9.2, we can directly see that we need p_i only for $i\in\{0,1,2,3\}$.

Consider the constraints of the above linear program. Constraint 9.1 guarantees that the variables x_k and y_k use exactly one unit of measure. Constraint 9.2 guarantees that the vertices in both partitions of the bipartite graph are incident to the same number of edges. Constraint 9.3 makes sure that the p_i sum up to y_4, which the total number of blue vertices for degree four per unit of the measure in $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$. Finally, Constraints 9.4 and 9.5 make sure that if any blue vertices of degree four exists in $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$, then sufficiently many red neighbours of the appropriate degrees the exists as required by definition of the variables p_i. That is, by definition of the variables p_i, a corresponding blue vertex of degree four in $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$ has i neighbours of degree three and $4-i$ neighbours of degree two per unit of the measure in $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$. Therefore, the number of edges incident to a degree two vertex $G[(\mathcal{R}\cup\mathcal{B})\setminus A]$ is at least $4p_0+3p_1+2p_2+1p_3$. Since, each such vertex of degree two has two endpoints, Constraints 9.4 follows. Constraint 9.5 follows similarly.

The solution to this linear program is $z=0.245614$ with all variables equal to zero, except: $x_4=0.589473$ and $y_3=0.442104$. The bound of $(0.245614+\epsilon)k$ now follows from Proposition 2.16 and the argument in the first paragraph of this proof. □

Theorem 9.4. *There exists an algorithm that solves* Partial Dominating Set *in* $\mathcal{O}(1.5014^n)$ *time and space.*

Proof. We use the same construction as in Theorem 9.2 that allows us to count partial red-blue dominating sets in order to solve Partial Dominating Set. More precisely, we count the number of of partial red-blue dominating sets of size κ that dominate exactly t vertices on the incidence graph I of the related variant of Set Cover, for each κ and t. To solve this counting problem, we use Algorithm 9.2.

Let $N_h(k)$ be the number of subproblems of measure h generated by Algorithm 9.2 through branching when starting from an input with measure k. Notice that due to Lemma 9.3, any subproblem of measure h that is solved by dynamic programming is solved in $\mathcal{O}^*(2^{(0.245614+\epsilon)h})=\mathcal{O}(1.1856^h)$ time, if we fix $\epsilon>0$ small enough.

To bound the number of subproblems generated by branching, we use the same recurrence relations as used in the proof of the running time of the polynomial-space algorithm; see Theorem 9.2 (we note that the recurrence relations are not given in the proof of Theorem 9.2 as they are identical to those in the proof of Theorem 8.5; they can be found there). Because we now consider using Algorithm 9.2 instead of Algorithms 9.1, we remove those recurrence relations that do not correspond to the modified branching rule. I.e., we remove any recurrence corresponding to branching on a vertex of degree three or four, unless it corresponds to branching on a blue vertex

of degree four with either a red neighbour of degree four or only red neighbours of degree three. If we solve the resulting set of recurrences with the measure defined above Lemma 9.3, we obtain: $N_h(k) \leq 1.1856^{k-h}$.

We now combine the analysis of the branching of the algorithm and the analysis of the dynamic programming and find that the total running time $T(k)$ on an input of measure k satisfies:

$$T(k) \leq \sum_{h \in H_k} N_h(k) \cdot 1.1856^h \leq \sum_{h \in H_k} 1.1856^{k-h} \cdot 1.1856^h = \sum_{h \in H_k} 1.1856^k$$

where H_k is the set of all possible measures of generated subproblems starting from a problem with measure k. We conclude that $T(k) = \mathcal{O}(1.1856^k)$ because we use only a finite number of weights, which makes $|H_k|$ polynomially bounded.

Let $v_{\max} = \max_{i \in \mathbb{N}} v(i)$ and $w_{\max} = \max_{i \in \mathbb{N}} w(i)$. As the input instance given to Algorithm 9.2 used to solve an instance of Partial Dominating Set contains n red vertices and n blue vertices, this proves a running time of $\mathcal{O}(1.1856^{(v_{\max}+w_{\max})n}) = \mathcal{O}(1.1856^{(1+1.386987)n}) = \mathcal{O}(1.5014^n)$. □

9.3. A Parameterised Algorithm for k-Set Splitting

As a second application of extended inclusion/exclusion-based branching, we will give a faster parameterised algorithm for k-Set Splitting. Both the parameterised and the unparameterised versions of this problem have been studied extensively, both combinatorially and algorithmically; see [7, 77, 106, 107, 125, 126, 229, 230, 231, 258, 327, 330].

Let us first define the problem formally. Consider a collection of sets $\mathcal{S}$ over a universe $\mathcal{U}$. In the k-Set Splitting problem, we are asked to divide the elements of $\mathcal{U}$ into two colour classes: red and green. In this setting, a set $S \in \mathcal{S}$ is said to be *split* if S contain at least one element of each of the two colour classes, i.e., at least one red element and at least one green element.

***k*-Set Splitting**
Input: A collection of sets $\mathcal{S}$ over a universe $\mathcal{U}$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Can $\mathcal{U}$ be partitioned into two colour classes such that at least k sets from $\mathcal{S}$ are split?

This problem is also known as k-Maximum Hypergraph 2-Colouring. Another related problem is the slightly more general k-Not-All-Equal Satisfiability. We note that our results also extend to this problem.

For the parameterised problem k-Set Splitting, there exists a long sequence of algorithms that each improve upon previous publications, see Table 9.1. Recently, an efficient kernel for this problem has been found by Lokshtanov and Saurabh [229]. This kernel has been used to obtain the previously fastest algorithms for this problem, namely an $\mathcal{O}^*(2^k)$-time algorithm using polynomial space and an $\mathcal{O}^*(1.9630^k)$-time algorithm using exponential space. We will use the same kernel and apply an algorithm based on our new branching rule to obtain our faster $\mathcal{O}^*(1.8213^k)$-time and polynomial-space algorithm.

Our algorithm uses the following result from [229]:

Authors		Time	Space usage
Dehne, Fellows, Rosamond	[106]	$\mathcal{O}^*(72^k)$	polynomial
Dehne, Fellows, Rosamond, Shaw	[107]	$\mathcal{O}^*(8^k)$	polynomial
Lokshtanov and Sloper	[230]	$\mathcal{O}^*(2.6499^k)$	polynomial
Chen and Lu (randomized algorithm)	[77]	$\mathcal{O}^*(2^k)$	polynomial
Lokshtanov and Saurabh	[229]	$\mathcal{O}^*(2^k)$	polynomial
Lokshtanov and Saurabh	[229]	$\mathcal{O}^*(1.9630^k)$	exponential
This chapter		$\mathcal{O}^*(1.8213^k)$	polynomial

Table 9.1. List of known results for k-Set Splitting.

Proposition 9.5 ([229]). *k-Set Splitting admits a kernel with at most $2k$ sets and a universe of at most k elements.*

This result allows us to perform the following preprocessing step in polynomial time: either we solve the instance directly, or we can transform it into an equivalent instance that has at most $2k$ sets and k elements.

After this first preprocessing step, we apply the following test that allows us to directly decide that instances containing sufficiently many large sets are Yes-instances.

Proposition 9.6 ([77]). *Let s_i be the number of sets of cardinality i in a given k-Set Splitting instance. The instance is a Yes-instance if:*

$$k \leq \frac{1}{2}s_2 + \frac{3}{4}s_3 + \frac{7}{8}s_4 + \frac{15}{16}s_5 + \frac{31}{32}s_6 + \frac{63}{64}s_7 + \cdots + \left(1 - \frac{1}{2^{i-1}}\right)s_i + \cdots$$

Proof. Consider colouring the elements of the instance either red or green, both with probability $\frac{1}{2}$ for each element independently. For any set of cardinality i, we consider the probability that it is not split by this random colouring. This probability is independent of the colour of the first element in the set, and, for each additional element, this probability is multiplied by $\frac{1}{2}$. Hence, a set of cardinality i has a probability of $\frac{1}{2^{i-1}}$ of not being split and a probability of $1 - \frac{1}{2^{i-1}}$ of being split.

By linearity of expectation, we see that the right hand side of the inequality in the proposition is the expected number of sets that will be split by this random colouring. Because there must exist a colouring that splits at least this expected number of sets, we can correctly decide that we have a Yes-instance if this instance satisfies the stated inequality. □

If the two described preprocessing steps do not solve our instance, then we apply a branch-and-reduce algorithm: Algorithm 9.3. This algorithm computes a list containing the number of ways to colour the elements such that exactly l sets are split, for each $0 \leq l \leq 2k$. Similar to our algorithms for variants of Dominating Set, Algorithm 9.3 will be applied to the incidence graph of a remaining k-Set Splitting instance. In this setting, the incidence graph is defined in the same way as the incidence graph of a Set Cover instance (see Section 5.1.1): we introduce a vertex for every set $S \in \mathcal{S}$ and every element $e \in \mathcal{U}$, and we add an edge between a vertex representing the set S and a vertex representing the element e if and only if $e \in S$.

Besides the incidence graph $I = (\mathcal{S} \cup \mathcal{U}, E)$ of the k-Set Splitting instance and a set of annotated vertices A, Algorithm 9.3 takes as input a partitioning of the vertices that represent sets into three sets $\mathcal{R}$, $\mathcal{G}$, and $\mathcal{W}$. This partitioning of $\mathcal{S}$ into these three sets is used to allow the algorithm to remove a vertex that represents an element as soon as the colour of this element is fixed by the algorithm. When a vertex representing an element $e \in \mathcal{U}$ is removed from I, then the partitioning will be updated to keep track of the fact that all sets that contained e now have an element of the colour given to e. More precisely, vertices that represent sets $S \in \mathcal{S}$ are put in $\mathcal{R}$ (*red*) or in $\mathcal{G}$ (*green*) if S now has an element that is given the colour red or green, respectively. Initially, all sets $S \in \mathcal{S}$ are in $\mathcal{W}$ (*white*); this partition contains all vertices corresponding to sets which elements have not been coloured thus far. If a set $S \in \mathcal{S}$ is split and thus contains elements of both colours, then the vertex that represents S is removed from I.

Algorithm 9.3 strongly resembles the polynomial-space algorithm that counts partial dominating sets from Section 9.2.2 (Algorithm 9.1). It uses the same annotation procedure that marks low degree vertices such that they will be ignored by the branching rules. Because this procedure is identical, we know by Lemma 8.3 that the subroutine SetSpl-DP$(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A)$ can be implemented in polynomial time using dynamic programming on tree decompositions (for more details, see Chapter 11).

Our algorithm uses three different branching rules, two of which use extended inclusion/exclusion-based branching as defined in Section 9.1.

1. **Branching on a vertex representing an element** (lines 11-15 of Algorithm 9.3). We recursively solve two subproblems, one in which the element is coloured red and one where the element is coloured green. After this, we shift the computed lists of numbers to update the number of split sets. Finally, we compute the required results by adding the two computed lists component wise.

2. **Branching on a vertex representing a set without coloured elements** (lines 16-21 of Algorithm 9.3). We apply the extended inclusion/exclusion-based branching rule as defined in Section 9.1. Because we are computing lists of numbers with the number of colourings splitting exactly l sets, for each $0 \leq l \leq 2k$, we need to compute only two of these lists: one corresponding to the subproblem where it is *optional* to split the set, and one corresponding to the subproblem where it is *forbidden* to split the set.

   In the optional branch, we remove the vertex representing the set. As a result, we are indifferent about splitting the corresponding set in this branch. In the forbidden branch, we remove the vertex representing the set and merge all vertices that represent the elements in the set to a single vertex. A solution to this instance corresponds to a solution of the original instance in which all elements in the set have the same colour, i.e., in which the set is not split.

3. **Branching on vertex representing a set with coloured elements** (lines 22-30 of Algorithm 9.3). Similar to branching on vertices that represent sets without coloured elements, we use the extended inclusion/exclusion-based branching rule and generate two subproblems. In the optional branch, we again remove the vertex corresponding to the set as we are indifferent about splitting it. In the forbidden branch, we generate an instance equivalent to making sure that the corresponding set will not be split. Since the vertex representing the set is already

Algorithm 9.3. An algorithm for counting two colourings that split exactly k sets.

Input: the incidence graph $I = (\mathcal{S} \cup \mathcal{U}, E)$ of a k-Set Splitting instance $(\mathcal{S}, \mathcal{U})$, a partitioning of the sets $\mathcal{S} = \mathcal{R} \cup \mathcal{G} \cup \mathcal{W}$ where $\mathcal{R}$ and $\mathcal{G}$ contain sets that contained an element that is coloured red and green, respectively, and $\mathcal{W}$ contains sets without coloured elements, and a set $A \subset (\mathcal{S} \cup \mathcal{U})$ of annotated vertices

Output: a list with, for each k, the number of colourings of $\mathcal{U}$ splitting exactly k sets

SetSpl$(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A)$:

if there exists a vertex $v \in (\mathcal{S} \cup \mathcal{U}) \setminus A$ of degree at most one in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ **then**
  return SetSpl$(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A \cup \{v\})$
else if there exist two vertices $v_1, v_2 \in (\mathcal{S} \cup \mathcal{U}) \setminus A$ both of degree two in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ that have the same two neighbours **then**
  return SetSpl$(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A \cup \{v_1\})$
else
  Let $e \in \mathcal{U} \setminus A$ be a vertex such that $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)$ is maximal
  Let $w \in \mathcal{W} \setminus A$ be a vertex such that $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w)$ is maximal
  Let $c \in (\mathcal{R} \cup \mathcal{G}) \setminus A$ be a vertex such that $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(c)$ is maximal
  if $\max\{d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e),\ d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w),\ d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(c)\} \leq 2$ **then**
    return SetSpl-DP$(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A)$
  else if $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e) = \max\{d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e),\ d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w),\ d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(c)\}$ **then**
    Let $L_{\text{red}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus N_{\mathcal{G} \cup \mathcal{U}}[e]], \mathcal{R} \cup N_{\mathcal{W}}(e), \mathcal{G} \setminus N(e), \mathcal{W} \setminus N(e), A \setminus N_{\mathcal{G}}(e))$
    Let $L_{\text{green}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus N_{\mathcal{R} \cup \mathcal{U}}[e]], \mathcal{R} \setminus N(e), \mathcal{G} \cup N_{\mathcal{W}}(e), \mathcal{W} \setminus N(e), A \setminus N_{\mathcal{R}}(e))$
    Update the number of split sets in L_{red} and L_{green}
    return $L_{\text{red}} + L_{\text{green}}$
  else if $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w) = \max\{d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e),\ d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w),\ d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(c)\}$ **then**
    $L_{\text{optional}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{w\}], \mathcal{R}, \mathcal{G}, \mathcal{W} \setminus \{w\}, A)$
    Merge all neighbours of w in I to a single vertex.
    $L_{\text{not}} = L_{\text{forbidden}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{w\}], \mathcal{R}, \mathcal{G}, \mathcal{W} \setminus \{w\}, A \setminus N(w))$
    Decrease the parameter k by one in L_{optional} and $L_{\text{forbidden}}$
    return $L_{\text{not}} + L_{\text{optional}} - L_{\text{forbidden}}$
  else // $d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(c) > d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e), d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w)$
    $L_{\text{optional}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus \{c\}], \mathcal{R} \setminus \{c\}, \mathcal{G} \setminus \{c\}, \mathcal{W}, A)$
    if $c \in \mathcal{R}$ **then**
      $L_{\text{not}} = L_{\text{forbidden}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus N^2_{\mathcal{U} \cup \mathcal{G}}[c]], (\mathcal{R} \setminus \{c\}) \cup N^2_{\mathcal{W}}(c), \mathcal{G} \setminus N^2(c), \mathcal{W} \setminus N^2(c), A \setminus N^2_{\mathcal{U} \cup \mathcal{G}}[c])$
    else // $c \in \mathcal{G}$
      $L_{\text{not}} = L_{\text{forbidden}} =$ SetSpl$(I[(\mathcal{S} \cup \mathcal{U}) \setminus N^2_{\mathcal{U} \cup \mathcal{R}}[c]], \mathcal{R} \setminus N^2(c), (\mathcal{G} \setminus \{c\}) \cup N^2_{\mathcal{W}}(c), \mathcal{W} \setminus N^2(c), A \setminus N^2_{\mathcal{U} \cup \mathcal{R}}[c])$
    Update the number of split sets in L_{not} and $L_{\text{forbidden}}$
    Decrease the parameter k by one in L_{optional} and $L_{\text{forbidden}}$
    return $L_{\text{not}} + L_{\text{optional}} - L_{\text{forbidden}}$

coloured, either red or green, this means that we must colour all elements in the set with the colour that is already present in this set. Note that other sets can be split as a result of this colouring operation.

For the running time analysis, we again use measure and conquer [144] (see Section 5.2). To analyse this k-Set Splitting algorithm, we use the following measure μ on a subproblem $(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A)$. Note that, in contrast to the rest of this thesis, we use μ instead of k for the measure to avoid confusion with the parameter k of k-Set Splitting.

$$\mu := \sum_{e \in \mathcal{U}, e \notin A} v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) + \sum_{w \in \mathcal{W}, w \notin A} x(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(w)) + \sum_{c \in (\mathcal{R} \cup \mathcal{G}), c \notin A} y(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(c))$$

Here, we use the following values for the weights:

i	≤ 1	2	3	4	5	6	> 6
$v(i)$	0.000000	0.726515	1.000000	1.000000	1.000000	1.000000	1.000000
$x(i)$	0.000000	0.525781	0.903938	1.054594	1.084859	1.084885	1.084885
$y(i)$	0.000000	0.387401	0.617482	0.758071	0.792826	0.792852	0.792852

We remind the reader that, similar to previous analyses, the weight functions v, x, and y have been chosen in such a way that the running time claimed in Theorem 9.8 is as fast as possible. See Section 5.6, for a general discussion of how we computed these weight functions.

Lemma 9.7. *Algorithm 9.3 runs in time $\mathcal{O}(1.31242^{\mu})$ on an input of measure μ.*

Proof. Let $N(\mu)$ be the number of subproblem generated on an input of measure μ. We generate a large set of recurrence relations of the form $N(\mu) \leq N(\mu - \Delta\mu_1) + N(\mu - \Delta\mu_2)$ representing all possible cases in which Algorithm 9.3 can branch.

Analogous to previous measure-and-conquer analyses in this thesis, we let $\Delta v(i) = v(i) - v(i-1)$, $\Delta x(i) = x(i) - x(i-1)$, and $\Delta y(i) = y(i) - y(i-1)$, and notice that the weight functions v, x, and y satisfy the following constraints:

1. $\Delta v(i), \Delta x(i), \Delta y(i) \geq 0$ for all i
2. $y(i) \leq x(i)$ for all i
3. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 2$
4. $\Delta x(i) \geq \Delta x(i+1)$ for all $i \geq 2$
5. $\Delta y(i) \geq \Delta y(i+1)$ for all $i \geq 2$
6. $2\Delta v(3) \leq v(2)$
7. $2 \min\{\Delta y(4),\ \Delta x(3)\} \leq y(2)$

Notice that annotating a vertex reduces the measure, and that annotated vertices have zero measure. Constraint 1 represents the fact that we want vertices with a higher degree in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ to contribute more to the measure of an instance. Constraint 2 represents the fact that moving a set from $\mathcal{W}$ to either $\mathcal{R}$ or $\mathcal{G}$ decreases the measure of an instance. Furthermore, Constraints 3, 4 and 5 are non-restricting steepness inequalities that make the formulation of the problem easier. Finally, the function of Constraints 6 and 7 is technical; this is explained in the proof below.

We first consider branching on a vertex e that represents an element. Let this vertex have w_i, r_i and g_i neighbours of degree i in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ that is contained in $\mathcal{W}$, $\mathcal{R}$, and $\mathcal{G}$, respectively.

Consider the branch where we colour the element *green*. In this branch, the measure decreases by $v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e))$ due to the fact that the vertex representing the element

is removed. Because the r_i sets in $\mathcal{R}$ containing the element are split, the measure decreases by an additional $r_i y(i)$. The g_i vertices of degree i in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ that represent sets in $\mathcal{G}$ that contain the element have their sizes reduced which decreases the measure by $g_i \Delta y(i)$. Furthermore, the w_i vertices of degree i in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ that represent sets in $\mathcal{W}$ are now put in $\mathcal{G}$ and reduced in size; this decreases the measure by an additional $w_i(x(i)-y(i-1))$. Finally, since the r_i vertices of degree i in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$ that represent sets in $\mathcal{R}$ containing the element are removed, other elements are reduced in frequency: this reduces the measure by an additional $\Delta v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) \sum_{i=2}^{\infty} r_i(i-1)$ by Constraints 3 and 6 and the fact that e is of maximum degree among the vertices that represent elements. Constraint 6 is used here to prevent that we remove too much measure in the following case. Notice that vertices of degree one have zero measure as they are annotated. Therefore, it could be the case that when we reduce the degree of a vertex of degree i at total of i times, then all its measure is removed after reducing it $i-1$ times. Constraint 6 prevents this by making sure that enough measure remains.

Let $\Delta\mu_{\text{red}}$ and $\Delta\mu_{\text{green}}$ be the decrease of the measure in the subproblem where the element is coloured red and green, respectively. Since the above analysis is symmetric if we colour e red instead of green, we have deduced the following inequalities:

$$\begin{aligned}
\Delta\mu_{\text{green}} &\geq v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) + \sum_{i=2}^{\infty} (r_i y(i) + g_i \Delta y(i) + w_i(x(i) - y(i-1))) \\
&\quad + \Delta v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) \sum_{i=2}^{\infty} r_i(i-1) \\
\Delta\mu_{\text{red}} &\geq v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) + \sum_{i=2}^{\infty} (g_i y(i) + r_i \Delta y(i) + w_i(x(i) - y(i-1))) \\
&\quad + \Delta v(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(e)) \sum_{i=2}^{\infty} g_i(i-1)
\end{aligned}$$

Now, consider branching on a vertex s representing a set in $\mathcal{R}$ or $\mathcal{G}$ containing e_i elements whose corresponding vertices have degree i in $I[(\mathcal{S} \cup \mathcal{U}) \setminus A]$. In the optional branch, s is removed and the vertices representing the elements contained in the set have their degrees reduced: this decreases the measure by $y(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s)) + \sum_{i=2}^{\infty} e_i \Delta v(i)$. In the forbidden branch, s is removed together with the vertices representing the elements contained in the set: this decreases the measure by $y(s) + \sum_{i=2}^{\infty} e_i v(i)$. Furthermore, removing the vertices that represent these elements reduces the degrees of other vertices that represent sets. Because of the branching order and Constraints 4 and 5, this decreases the measure by either at least $\Delta y(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s))$ if the set is in $\mathcal{R}$ or $\mathcal{G}$ and by at least $\Delta x(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s)-1)$ if the set is in $\mathcal{W}$. In total, this gives a decrease in the measure of at least $\min\{\Delta y(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s)),\ \Delta x(d_{(\mathcal{S} \cup \mathcal{U}) \setminus A}(s) - 1)\}$ for each time the degree of a vertex is reduced. Similar to how we used Constraint 6 when deriving the formula associated with branching on an element e, we use Constraint 7 here to make sure that we do not remove too much measure if we reduce the degree a vertex of degree i a total of i times.

Let $\Delta\mu_{\text{optional}}$ and $\Delta\mu_{\text{forbidden}}$ be the decrease of the measure in the subproblem

where splitting the set is made optional or forbidden, respectively. We have deduced:

$$\begin{aligned}\Delta\mu_{\text{optional}} &\geq y(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i \Delta v(i)\\ \Delta\mu_{\text{forbidden}} &\geq y(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i v(i)\\ &\quad + \min\{\Delta y(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)),\ \Delta x(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)-1)\} \cdot \sum_{i=2}^{\infty} e_i(i-1)\end{aligned}$$

Finally, we consider branching on a vertex representing a set in $\mathcal{W}$. Analogous to the above, we derive:

$$\begin{aligned}\Delta\mu_{\text{optional}} &\geq x(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s) + \sum_{i=2}^{\infty} e_i \Delta v(i)\\ \Delta\mu_{\text{forbidden}} &\geq x(d_{(\mathcal{S}\cup\mathcal{U})\setminus A}(s)) + \sum_{i=2}^{\infty} e_i v(i) - v(\sum_{i=2}^{\infty} e_i(i-1))\end{aligned}$$

The main difference in the derivation compared to the case where we branch on a vertex representing a set in $\mathcal{G}$ or $\mathcal{R}$ is the term $-v(\sum_{i=2}^{\infty} e_i(i-1))$. This term accounts for the fact that if we decide that it is forbidden to split a set from $\mathcal{W}$, then we do not remove the vertices representing the elements in this set, but merge these elements to a single vertex instead. This causes a new vertex of degree degree at most $\sum_{i=2}^{\infty} e_i(i-1)$ to be created which has measure at most $v(\sum_{i=2}^{\infty} e_i(i-1))$.

Having lower bounds on the decrease of the measure in every subproblem, we can now solve the resulting large set of recurrence relations with the given weights. We notice that we have to use only a finite set of recurrence relations since we use only a finite number of weights which causes recurrences corresponding to branching on large degree vertices to be dominated by recurrences corresponding to smaller degree vertices. We solve this set of recurrence relations and find that $N(\mu) \leq 1.31242^{\mu}$.

Since both the annotation procedure and the procedure SetSpl-DP$(I, \mathcal{R}, \mathcal{G}, \mathcal{W}, A)$ can be implemented in polynomial time, the running time is dominated by the exponentially number of generated subproblems. We conclude that the algorithm runs in $\mathcal{O}(1.31242^{\mu})$ time. □

We are now ready to prove the main result of this section.

Theorem 9.8. *There exists an $\mathcal{O}^*(1.8213^k)$-time and polynomial-space algorithm for k-Set Splitting.*

Proof. The claimed algorithm executes the following steps. First, it applies the preprocessing given by Propositions 9.5 and 9.6. Then, it applies Algorithm 9.3 to the incidence graph of the resulting instance $(\mathcal{S}, \mathcal{U})$. Algorithm 9.3 then produces a list containing the number of colourings of the elements in $\mathcal{U}$ that split exactly l sets from $\mathcal{S}$, for each $0 \leq l \leq 2k$. From this list, the algorithm can directly see if there exist a colouring splitting at least k sets.

The exponential part of the running time of this algorithm comes from the call to Algorithm 9.3 since the preprocessing is done in polynomial time. We use Lemma 9.7

to compute this running time. By Proposition 9.5, an instance that is given to Algorithm 9.3 can have at most k elements; these elements each have measure at most 1. By Proposition 9.6, an instance that is given to Algorithm 9.3 must satisfy $k > \sum_{i=2}^{\infty} \left(1 - \frac{1}{2^{i-1}}\right) s_i$ where s_i is the number of sets of cardinality i. Therefore, the maximum measure z of such an instance is bounded by the solution of:

$$z = k + \max \sum_{i=2}^{\infty} x(i)s_i \quad \text{with the restriction} \quad \sum_{i=2}^{\infty} \left(1 - \frac{1}{2^{i-1}}\right) s_i < k$$

We find that the maximum equals $z < 2.205251k$. The claimed running time follows from Lemma 9.7 since $1.31242^z < 1.31242^{2.205251k} < 1.8213^k$. □

We now show that our results extend to the slightly more general problem k-Not-All-Equal Satisfiability.

***k*-Not-All-Equal Satisfiability**
Input: A set of clauses C using a set of variables X.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a truth assignment of the variables X such that at least k clauses in C contain a literal set to true and a literal set to false?

Notice that this problem extends k-Set Splitting if one identifies clauses with sets and variables with elements. The difference is that variables can occur both as positive literals and as negative literals in the clauses of a k-Not-All-Equal Satisfiability instance.

Corollary 9.9. *There exists an $\mathcal{O}^*(1.8213^k)$-time and polynomial-space algorithm for k-Not-All-Equal Satisfiability.*

Proof. We note that the kernel of size $2k$ of Proposition 9.5 can be extended to k-Not-All-Equal Satisfiability, as claimed in [229]. Consider the proof of Proposition 9.6. The probabilistic argument here does not change due to the signs of literals, therefore the statement remains valid for instances of k-Not-All-Equal Satisfiability with s_i clauses of size i. From these two facts, we conclude that the claimed result follows from the proof of Theorem 9.8 if we can give a branch-and-reduce algorithm for k-Not-All-Equal Satisfiability whose branching behaviour equals that of Algorithm 9.3 when applied after these two preprocessing steps.

To create such an algorithm, we first notice that we can define the incidence graph of a k-Not-All-Equal Satisfiability instance analogous to the incidence graph of a k-Set Splitting instance: introduce a vertex for every variable $x \in X$ and every clause $c \in C$ and connect a vertex representing a variable x and a clause c if and only if $x \in c$ or $\neg x \in c$. Notice that the incidence graph of a k-Not-All-Equal Satisfiability instance is identical to the incidence graph of the related k-Set Splitting instance that is obtained through removing the negations from the literals and then replacing variables and clauses by elements and sets, respectively.

We will treat incidence graphs of k-Not-All-Equal Satisfiability instances in the same way as those of k-Set Splitting instances. That is, we remove any vertex representing a variable if the algorithm sets this variable to $True$ or $False$. We keep

track of these values in the clauses by letting the algorithm maintain a partitioning of the clauses into tree sets: the clauses that contain a variable set to $True$, the clauses that contain a variable set to $False$, and the clauses that contain no variables that are given an assignment thus far. Furthermore, we remove any vertex representing a clause that is satisfied, i.e., a clause that contains at least one literal that is set to $True$ and at least one literal that is set to $False$.

We will now continue by treating each aspect of the branch-and-reduce algorithm that we use to prove this corollary. This algorithm will closely resemble Algorithm 9.3. We will show that we can use the same annotation procedure as Algorithm 9.3, and branching rules with a behaviour that is similar to that of Algorithm 9.3.

The annotation procedure. We can use the same annotation procedure on incidence graphs as used in Algorithm 9.3 (and Algorithms 8.1, 9.1, and 9.2) because this procedure does not influence the correctness of the algorithm. Since the incidence graphs are equal to those of k-Set Splitting instances, the effect of this procedure will be identical to that of the procedure in Algorithm 9.3. Furthermore, if we let A be the set of annotated vertices, then we can solve subproblems corresponding to incidence graphs $I = (V, E)$ in which $I[V \setminus A]$ has maximum degree two in polynomial time by dynamic programming on a tree decomposition. This follows by Lemma 8.3 and standard dynamic programming algorithms on tree decompositions (see Chapter 11).

Branching rules - general introduction. It remains to show that we can give branching rules that cause the algorithm to generate at most $\mathcal{O}(1.8213^k)$ subproblems. We will give these branching rules below: one branching rule that branches on variables and two branching rules that branch on clauses. The branching rules are almost identical to those in Algorithm 9.3. Because of this analogy between the branching rules, we can let our algorithm use the same procedure for choosing which branching rule to apply as Algorithm 9.3 does for the branching rule on corresponding k-Set Splitting instances.

Below, we will show that the effect of these branching rules on the incidence graph of the k-Not-All-Equal Satisfiability instance is similar to the effect of the branching rules of Algorithm 9.3 on incidence graphs of k-Set Splitting instances. We note that the branching rules below will not give the same incidence graphs as generated subproblems when applied to a k-Not-All-Equal Satisfiability instance (C, X) compared to the subproblems that are generated by Algorithm 9.3 when applied to the k-Set Splitting instance obtained from (C, X) by removing the negations from all negative literals. The branching rules are similar only in the sense that we can use the same set of recurrence relations to prove an upper bound on the number of subproblems that they generate. The claimed running time then follows from the analysis of Algorithm 9.3 in the proof of Theorem 9.8.

Branching on a variable. Consider branching on a variable x in a k-Not-All-Equal Satisfiability instance: we set the variable to $True$ in one branch, and to $False$ in the other branch. In both branches, we perform the following operations to the incidence graph. First, we remove the vertex representing the variable x; as a result, vertices representing clauses that contain a literal of x have their degrees decreased by one. Secondly, we move vertices representing clauses that now contain literals with an assigned truth value to the partition that corresponds to the new assignment. Notice that, in this case, the sign of the literals of x have no effect on

the incidence graphs of the generated subproblems: the only thing that changes if we negate a literal is the partition in which the vertex representing the corresponding clause is put. Finally, vertices representing clauses containing literals with an assigned truth value are removed if the assignment causes the clause to be satisfied. For these clauses, the incidence graphs of the generated subproblems are affected by the sign of the literal of x occurring in C.

One can check that the set of recurrence relations in Theorem 9.8 that corresponds to branching on an element remains valid for this branching rule. The only real difference is the effect of the signs of the literals. However, the same set of recurrence relations can be used because changing the sign of a literal corresponds to considering a different recurrence relation where the partition of the clause is flipped between already having the value $True$ and already having the value $False$.

Branching on a set without literals with an assigned value. We now consider branching on a vertex representing a clause C that is in the partition corresponding to the fact that C does not contain literals with an assigned truth value. Now, we apply a form of extended inclusion/exclusion-based branching. In the optional branch, we remove the vertex representing the clause as we are now indifferent about satisfying it. In the forbidden branch, we prevent that the clause it satisfied by modifying the instance in the following way. First, we make sure that all literals in C are positive literals; this is done by flipping the sign of all literals of a variable in C if necessary. Note that this results in an equivalent instance: the same number of truth assignments satisfy exactly k clauses, for each k. Second, we replace all literals of the variables in C by literals of the same sign of a single variable x from C. For example, when we consider the clause $(x, y, \neg z)$, this results in (x, x, x) while all other clauses that contain y or z are modified such that y is replaced by x (and $\neg y$ by $\neg x$) and $\neg z$ is replaced by x (and z by $\neg x$). Notice that the effect of this modification is equal to merging the elements in a set in a k-Set Splitting instance.

For the above branching rule, the effects on the incidence graph are identical to the effects of the branching rule of Algorithm 9.3 that branches on a set in $\mathcal{W}$. Therefore, the set of recurrence relations in Theorem 9.8 that correspond to branching on a set in $\mathcal{W}$ remains valid for this branching rule.

Branching on a set containing literals with an assigned value. Finally, consider branching on a vertex representing a clause C that is in the partition corresponding to the fact that C contains literals which are either set to $True$ or $False$. For this case, we can also give a branching rule whose effects on an incidence graph are identical to the effects that a branching rule of Algorithm 9.8 has on the same graph if it would be an incidence graph of a k-Set Splitting instance. This branching rule also uses extended inclusion/exclusion-based branching and is similar to the branching rule of Algorithm 9.8 that branches on a set S in $\mathcal{R}$ or $\mathcal{G}$. It works in the following way. In the optional branch, the vertex representing the clause C is removed; this is identical to the removal of the vertex representing S by Algorithm 9.8. In the forbidden branch, the vertex representing the clause C is removed and all variables in C are given a value such that their literals in C gain the value corresponding to the partition that contains C. This results in the removal of the vertices representing these variables, and possibly some additional vertices representing clauses that have now become satisfied. We note that the effects on the incidence graph are identical to colouring all elements

in S with colour corresponding to the partition ($\mathcal{R}$ or $\mathcal{G}$) that contains S.

Since the effects on the incidence graph are identical in both branches, the set of recurrence relations in Theorem 9.8 that correspond to branching on a set in $\mathcal{R}$ or $\mathcal{G}$ remains valid for this branching rule.

We have proven that the number of subproblems generated by the algorithm satisfies the same recurrence relations as those used in the proof of Theorem 9.8. Hence, we conclude that the running time satisfies the upper bound claimed in Theorem 9.8. □

9.4. Concluding Remarks

In this chapter, we have shown that inclusion/exclusion-based branching can also be used in the setting where only a fixed number of requirements need to be satisfied. This resulted in our extended inclusion/exclusion-based branching rule. We demonstrated this approach on two problems in different settings. First, we have given faster exact exponential-time algorithms for Partial Dominating Set. Secondly, we have given a faster parameterised algorithm for the well-studied parameterised problem k-Set Splitting.

These two examples show that the extended inclusion/exclusion-based branching rule is quite a powerful tool. It would be interesting to see whether more problems exist for which this approach leads to faster algorithms, both in the field of exact exponential-time algorithms as in the field of parameterised algorithms. One example might be using extended inclusion/exclusion-based branching for faster algorithms for Partial Dominating Set on some restricted graph classes; this is similar to what we did in Section 8.6 for Dominating Set and #Dominating Set.

10 Partitioning a Graph Into Two Connected Subgraphs

There are several natural and elementary algorithmic problems that have the following form: given a graph G, does the structure of some fixed graph H appear as a pattern within G. One such structure that is well known is that of a *minor*: a graph H is a minor of G if it can be obtained from G by a series of vertex or edge deletions and edge contractions. The H-Minor Containment problem asks whether a given graph G contains H as a minor. A celebrated result by Robertson and Seymour [269] states that the H-Minor Containment problem can be solved in polynomial time for every fixed pattern graph H. They obtain this result by designing an algorithm that solves the following problem in polynomial time for instances with bounded $|Z_1|+|Z_2|+\cdots+|Z_k|$.

Disjoint Connected Subgraphs
Input: A graph $G = (V, E)$ and mutually disjoint non-empty sets $Z_1, Z_2, \ldots, Z_k \subseteq V$.
Question: Do there exist mutually vertex-disjoint connected subgraphs $G_1, G_2, \ldots, G_k$ of G (with $G_i = (V_i, E_i)$) such that Z_i is contained in V_i for every $1 \leq i \leq k$?

In this chapter, we consider the 2-Disjoint Connected Subgraphs problem which is a variant of Disjoint Connected Subgraphs where $k = 2$ and the vertex sets Z_1 and Z_2 may have arbitrary size. For an example instance, see Figure 10.1. One reason why this problem is interesting is that the technique of inclusion/exclusion-based branching can be applied to it. Thus, we will take a short leave of the main topic of this thesis, namely domination problems in graphs, to further explore

[†]This chapter is joint work with Daniel Paulusma. The chapter contains results of which a preliminary version has been presented at the 20th International Symposium on Algorithms and Computation (ISAAC 2009) [251].

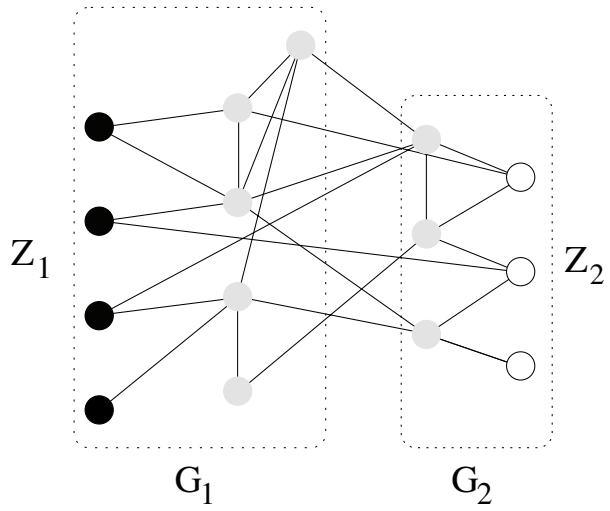


Figure 10.1. An instance of 2-Disjoint Connected Subgraphs. Z_1 contains the black vertices and Z_2 contains the white vertices. The partitioning showing that this is a Yes-instance is given by G_1 and G_2.

the inclusion/exclusion-based branching technique and give faster algorithms for 2-Disjoint Connected Subgraphs on some graph classes.

We will do so by considering previous results on 2-Disjoint Connected Subgraphs by van 't Hof et al. on some graph classes [313] and improve their algorithms using our inclusion/exclusion-based branching technique. Among others, van 't Hof et al. give $\mathcal{O}(1.5790^n)$-time algorithms on P_6-free graphs and split graphs. Here, we will improve these results to $\mathcal{O}(1.2051^n)$-time algorithms. We obtain this result by giving a faster algorithm for 2-Hypergraph 2-Colouring instances that arise from the approach of van 't Hof et al. [313] for 2-Disjoint Connected Subgraphs.

The result in this chapter demonstrates that it is sometimes useful to have two branching phases in an algorithm, where the first phase considers the decision variant of a problem and the second phase considers the counting variant of the same problem. This is different to the previous two chapters. It has the advantage that we can use more powerful reduction rules in the first phase since they need only to preserve the existence of a solution and do not need to count all solutions. The disadvantage, namely that we cannot use inclusion/exclusion-based branching, now holds only temporarily since we start counting in the second phase. This second phase counts only the number of solutions to the specific subproblem generated by the first phase. Although this helps to design fast algorithms, we note that the number of solutions computed in the second phase are not directly related to the number of solutions of the initial instance.

This chapter is organised as follows. First, we have a closer look at 2-Disjoint Connected Subgraphs in Section 10.1. In this section, we show how this problem relates to 2-Hypergraph 2-Colouring and state the main result of this chapter. This result is based on the algorithm for 2-Hypergraph 2-Colouring given in Section 10.2. Finally, we give some concluding remarks in Section 10.3.

10.1. The 2-Disjoint Connected Subgraphs Problem

Let us first look at 2-Disjoint Connected Subgraphs more closely.

2-Disjoint Connected Subgraphs
Input: A graph $G = (V, E)$ and two mutually disjoint non-empty sets $Z_1, Z_2 \subseteq V$.
Question: Do there exist two mutually vertex-disjoint connected subgraphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ of G such that Z_1 is contained in V_1 and Z_2 is contained in V_2?

Besides being a variant of Disjoint Connected Subgraphs, this problem also appears in other settings. For example, Gray et al. [173] motivate this problem from an application in computational geometry, namely finding a realization of an imprecise terrain that minimizes the total number of local minima and local maxima.

2-Disjoint Connected Subgraphs is $\mathcal{NP}$-complete even if one of the sets Z_1 and Z_2 has cardinality two [313], or when restricted to planar graphs [173]. When considering exact algorithms, no faster exact algorithm for 2-Disjoint Connected Subgraphs is known than the trivial brute-force $\mathcal{O}^*(2^n)$-time algorithm.

In an attempt to design fast exact exponential-time algorithms for this problem, van 't Hof et al. focus on restrictions of the problem to certain graph classes [313]. They show that 2-Disjoint Connected Subgraphs is already $\mathcal{NP}$-complete for P_5-free graphs and split graphs, whereas it is polynomially solvable for P_4-free graphs. They also give algorithms solving 2-Disjoint Connected Subgraphs faster than $\mathcal{O}^*(2^n)$ for graphs in the classes $\mathcal{G}^{k,r}$ which we define below. In particular, these classes include all P_l-free graphs.

Definition 10.1 (P_l-Free Graph). For any $l \geq 2$, a P_l-free graph is a graph that does not contain the path on l vertices as a subgraph.

Definition 10.2 (Split Graph). A split graph G is a graph of which the set of vertices V can be partitioned into two sets I and C such that I is an independent set and C is a clique.

Definition 10.3 (Graph Class $\mathcal{G}^{k,r}$). For any $k \in \mathbb{N}$ and $r \in \mathbb{N}$, $\mathcal{G}^{k,r}$ is the class of graphs in which all connected induced subgraphs have a connected distance-r dominating set of size at most k.

In other words, for each $G \in \mathcal{G}^{k,r}$ and $X \subset V$ such that $G[X]$ is connected, $G[X]$ contains a vertex set $Y \subseteq X$ with $|Y| \leq k$ and $G[Y]$ connected such that each $v \in X$ lies at distance at most r from Y in $G[X]$.

Somewhat surprisingly, for any fixed k, 2-Disjoint Connected Subgraphs on $\mathcal{G}^{k,r}$ can be solved in polynomial time if $r = 1$ or if one of the given sets of vertices Z_1 and Z_2 has fixed size [313]. However, for any fixed k and $r \geq 2$, the 2-Disjoint Connected Subgraphs on $\mathcal{G}^{k,r}$ problem is $\mathcal{NP}$-complete. For these graph classes, van 't Hof et al. present an algorithm that solves 2-Disjoint Connected Subgraphs in $\mathcal{O}^*(\alpha_r^n)$ time [313], where:

$$\alpha_r = \min_{0<c<0.5} \left\{ \max \left\{ \frac{1}{c^c(1-c)^{1-c}}, 2^{1-\frac{2c}{r-1}} \right\} \right\}$$

In particular, their algorithm solves the problem in $\mathcal{O}(1.5790^n)$ time on P_6-free graphs and in $\mathcal{O}(1.7737^n)$ time on P_7-free graphs. Also, this algorithm runs in $\mathcal{O}(1.5790^n)$ time on split graphs.

These results follow from the following observation that is a direct consequence of the characterizations of P_l-free graphs for $l = 6$ [310] and for $l \geq 7$ [12].

Proposition 10.4 ([12, 313]). *The class of split graphs and the class of P_l-free graphs for $l \in \{5, 6\}$ belong to $\mathcal{G}^{4,2}$. The class of P_l-free graphs for $l \geq 7$ belongs to $\mathcal{G}^{1,l-3}$.*

10.1.1. Relation to 2-Hypergraph 2-Colouring

We will give an algorithm for 2-Disjoint Connected Subgraphs restricted to the case where Z_1 and Z_2 both contain a set of vertices that is both connected in G and that dominates $V \setminus (Z_1 \cup Z_2)$ in G. We then use this algorithm to obtain our results on P_6-free graphs and split graphs. Our approach is as follows: first, we translate 2-Disjoint Connected Subgraphs to the 2-Hypergraph 2-Colouring problem; then, we give an exact algorithm for the latter problem.

Let us first introduce the 2-Hypergraph 2-Colouring problem which is a variant of Hypergraph 2-Colouring. A *hypergraph* $H = (Q, \mathcal{S})$ consists of a set $Q = \{q_1, q_2, \ldots, q_n\}$ of elements together with a set $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of Q called *hyperedges*. A *2-colouring* of H is a partition of Q into two sets Q_1, Q_2 such that each set $S \in \mathcal{S}$ contains at least one element from each of the partitions. Notice the equivalence to the k-Set Splitting problem, as defined in Section 9.3, with $k = m$.

These notions can be generalized as follows. A *2-hypergraph* $H = (Q, \mathcal{L}, \mathcal{R})$ consists of a set $Q = \{q_1, q_2, \ldots, q_n\}$ of elements together with two (not necessarily disjoint) sets $\mathcal{L} = \{L_1, L_2, \ldots, L_s\}$ and $\mathcal{R} = \{R_1, R_2 \ldots, R_t\}$ of subsets of Q. We call $\mathcal{L}$ and $\mathcal{R}$ the *hyperedge classes* of H. With every 2-hypergraph $H = (Q, \mathcal{L}, \mathcal{R})$, we associate an *incidence graph* I which is a bipartite graph on $Q \cup \mathcal{L} \cup \mathcal{R}$ that contains an edge between a vertex representing an element $q \in Q$ and a vertex representing a hyperedge $S \in \mathcal{L} \cup \mathcal{R}$ if and only if $q \in S$. Let the *dimension* d of a 2-hypergraph $H = (Q, \mathcal{L}, \mathcal{R})$ be $d = |Q| + |\mathcal{L}| + |\mathcal{R}|$, that is, the number of vertices in the incidence graph of H. A *2-colouring* of H is a partition of Q into two sets Q_l, Q_r such that each hyperedge $L \in \mathcal{L}$ contains an element from Q_l and each hyperedge $R \in \mathcal{R}$ contains an element from Q_r. This leads to the following decision problem.

2-Hypergraph 2-Colouring
Input: A 2-hypergraph $H = (Q, \mathcal{L}, \mathcal{R})$.
Question: Does H have a 2-colouring?

Note that a hypergraph $H = (Q, \mathcal{S})$ is 2-colourable if and only if the 2-hypergraph $H' = (Q, \mathcal{S}, \mathcal{S})$ is 2-colourable. The problem Hypergraph 2-Colouring asks if a hypergraph is 2-colourable; this problem is $\mathcal{NP}$-complete [162]. Thus, we can directly observe that 2-Hypergraph 2-Colouring is $\mathcal{NP}$-complete as well.

Observe that 2-Hypergraph 2-Colouring stays $\mathcal{NP}$-complete if we require that the 2-hypergraph $H = (Q, \mathcal{L}, \mathcal{R})$ contains a hyperedge equal to Q in both hyperedge classes $\mathcal{L}$ and $\mathcal{R}$. We consider the incidence graph I of such a 2-hypergraph that contains a hyperedge equal to Q in both $\mathcal{L}$ and $\mathcal{R}$. Notice that, in this incidence graph, there exists a connected vertex set $S \subseteq \mathcal{L}$ (and one $S' \subseteq \mathcal{R}$) in I that dominates

$V \setminus (\mathcal{L} \cup \mathcal{R})$: take the connected singleton vertex set containing the vertex corresponding to the hyperedge that equals Q.

Now, let (G, Z_1, Z_2) be an instance of 2-Disjoint Connected Subgraphs where Z_1 and Z_2 both contain a connected vertex set in G that dominates $V \setminus (Z_1 \cup Z_2)$. The first observation we make is that this special case of 2-Disjoint Connected Subgraphs stays $\mathcal{NP}$-complete. This follows from the fact that if I is the incidence graph of a 2-Hypergraph 2-Colouring instance with a hyperedge equal to Q in both $\mathcal{L}$ and $\mathcal{R}$, then this is a Yes-instance of 2-Hypergraph 2-Colouring if and only if $(I, \mathcal{L}, \mathcal{R})$ is a Yes-instance of the given variant of 2-Disjoint Connected Subgraphs.

In Section 10.2, we will give an $\mathcal{O}(1.2051^d)$-time algorithm for 2-Hypergraph 2-Colouring instances of dimension d, see Lemma 10.10. Based on this result, we can derive the following lemma.

Lemma 10.5. *There is an algorithm that solves* 2-Disjoint Connected Subgraphs *in* $\mathcal{O}(1.2051^n)$ *time on instances* (G, Z_1, Z_2) *in which* Z_1 *and* Z_2 *both contain a connected vertex set in* G *that dominates* $V \setminus (Z_1 \cup Z_2)$.

Proof. Let n be the number of vertices in G, and let $U = V \setminus (Z_1 \cup Z_2)$. We transform each connected component of $G[Z_1]$ and $G[Z_2]$ into a single vertex by performing a series of edge contractions in G. Let G' be the resulting graph, with independent vertex sets Z_1' and Z_2' obtained from Z_1 and Z_2, respectively. Now, there exist vertices $z_1 \in Z_1'$ and $z_2 \in Z_2'$ that are adjacent to all vertices in U in G', because both Z_1 and Z_2 contained a connected vertex set in G that dominates U.

Notice that the new instance G' with Z_1' and Z_2' is equivalent to the instance G with Z_1 and Z_2. Because both z_1 and z_2 are adjacent to every vertex in U, this equivalence still holds after we remove all edges between different vertices in U from G'. The resulting bipartite graph G'' is the incidence graph of the 2-Hypergraph 2-Colouring instance (U, Z_1', Z_2'), and this instance has dimension at most n.

Now, the result follows by using the $\mathcal{O}(1.2051^d)$-time algorithm of Lemma 10.10 on the 2-Hypergraph 2-Colouring instance (U, Z_1', Z_2'), and noticing that $d \leq n$. □

10.1.2. Our Algorithm

We will now give our algorithm for 2-Disjoint Connected Subgraphs on graphs in $\mathcal{G}^{k,2}$ for any fixed $k \in \mathbb{N}$.

Theorem 10.6. *For any fixed integer* $k \geq 1$, *there exists an* $\mathcal{O}(1.2051^n)$*-time algorithm for* 2-Disjoint Connected Subgraphs *on graphs in* $\mathcal{G}^{k,2}$.

Proof. Let (G, Z_1, Z_2) be an instance of 2-Disjoint Connected Subgraphs with $G \in \mathcal{G}^{k,2}$, and let $U = V \setminus (Z_1 \cup Z_2)$. By definition of $\mathcal{G}^{k,2}$, any solution (G_1, G_2) of this instance of 2-Disjoint Connected Subgraphs is such that G_1 contains a connected vertex set D_1 of size at most k and G_2 contains a connected vertex set D_2 of size at most k such that any vertex $v \in V$ lies at distance at most two from both D_1 and D_2.

Our algorithm tries all combinations of the $\mathcal{O}(n^k)$ sets $D_1 \subseteq Z_1 \cup U$ of up to k vertices and all the $\mathcal{O}(n^k)$ sets $D_2 \subseteq Z_2 \cup U$ of up to k vertices. For such a combination, the algorithm checks whether $D_1 \cap D_2 = \emptyset$ and whether $G[D_1]$ and $G[D_2]$ are both connected. If one of these conditions fails, we continue with the next combination of sets. Otherwise, we keep D_1 and D_2 and form a new instance (G', Z_1', Z_2'), where:

- G' is the subgraph of G obtained after removing all vertices from U that are neither adjacent to D_1 nor to D_2.

  The reason that we may remove these vertices is because they are redundant in any possible solution (G_1, G_2) in which any vertex $v \in V$ is at distance at most two from both D_1 and D_2.

- $Z_1' = Z_1 \cup D_1 \cup \{u \in U \mid u \text{ is adjacent to } D_1 \text{ but not to } D_2\}$ and
  $Z_2' = Z_2 \cup D_2 \cup \{u \in U \mid u \text{ is adjacent to } D_2 \text{ but not to } D_1\}$.

This instance (G', Z_1', Z_2') is equivalent to $(G, Z_1 \cup D_1, Z_2 \cup D_2)$ while both Z_1' and Z_2' contain a connected vertex set of size at most k that lies at distance at most two from any vertex in G'. As a result we can apply Lemma 10.5 to each generated instance.

Since, for any fixed $k \in \mathbb{N}$, the algorithm of Lemma 10.5 is executed on a polynomially bounded number of instances with at most n vertices, the result follows. □

Corollary 10.7. *There exist an $\mathcal{O}(1.2051^n)$-time algorithm for* 2-Disjoint Connected Subgraphs *on P_6-free graphs and split graphs.*

Proof. Combine Theorem 10.6 with Proposition 10.4. □

10.2. A 2-Hypergraph 2-Colouring Algorithm

We will now give our algorithm for 2-Hypergraph 2-Colouring. This algorithm has been used in Section 10.1.1 to prove Lemma 10.5 and Theorem 10.6. We present this algorithm to illustrate the approach of using a branching algorithm that has two phases; one where we consider the decision problem, and one where we consider the associated counting problem. Because of these two phases, the first phase can use more powerful reduction rules that preserve the existence of a solution but not the number of solutions, while the second phase can use inclusion/exclusion-based branching.

Our algorithm will also use dynamic programming on tree decompositions to solve sparse instances. This is similar to the approach used in our exponential-space algorithms in Chapters 8 and 9. This can be seen as a third phase of our algorithm. Consequently, we present the algorithm as an algorithm with three phases.

Throughout the description of the algorithm, we denote the 2-hypergraph under consideration by $H = (Q, \mathcal{L}, \mathcal{R})$ and its incidence graph by $I = (Q \cup \mathcal{L} \cup \mathcal{R}, E)$. Here, H contains vertices for the elements which have no colour yet and vertices for the hyperedges which have no element of the appropriate colour yet (colour l for $L \in \mathcal{L}$ and colour r for $R \in \mathcal{R}$); all other vertices representing elements and hyperedges are removed. From now on, if we say that an element in Q or a hyperedge in $\mathcal{L} \cup \mathcal{R}$ has a certain *degree*, we mean its degree in I.

Phase 1: Branch and Reduce on the Decision Problem. We exhaustively apply two reduction rules and branch on the elements $q \in Q$: either give q colour l or colour r. We go to Phase 2 with a 2-hypergraph if every remaining element appears in at most three hyperedges in $\mathcal{L} \cup \mathcal{R}$. We go into more details below.

Phase 1 uses the following two reduction rules which are clearly correct:

Reduction Rule 10.1 (Elements in Hyperedges of at Most One Hyperedge Class). Let q be an element of H that occurs in at most one hyperedge class. If q has degree zero, remove q. If q occurs only in $\mathcal{L}$, then colour it with l. Otherwise, if q occurs only in $\mathcal{R}$, then colour it with r. Hereafter, remove q and all hyperedges containing q, and if H becomes empty this way, return YES.

Reduction Rule 10.2 (Hyperedges of Degree One). Let $S = \{q\}$ be a hyperedge of degree one. If $S \in \mathcal{L}$, colour q with colour l, otherwise, colour q with colour r. Next, remove the element q, the hyperedge S, and all other hyperedges in $\mathcal{L} \cup \mathcal{R}$ that have received their appropriate colour. If this results in $\emptyset \in \mathcal{L} \cup \mathcal{R}$, then return NO.

When Rules 10.1 and 10.2 cannot be applied, we select an element $q \in Q$ of maximum degree in I. If q has degree at most three, then we send this subproblem to Phase 2. Otherwise, we branch on q. In one branch, we colour q with l and remove q and all hyperedges in $\mathcal{L}$ containing q. In the other branch, we colour q with r and remove q and all hyperedges in $\mathcal{R}$ containing q. We solve both generated subproblems recursively and return YES if at least one of the recursive calls on the subproblems returns YES and NO otherwise. If a subproblem is given to Phase 2, then we return YES if the recursive call returns that there exists at least one solution and NO otherwise.

Phase 2: Inclusion/Exclusion-Based Branching on the Counting Problem. The algorithm now switches to the counting variant of our problem: we compute the number of 2-colourings of $H = (Q, \mathcal{L}, \mathcal{R})$. Because of this, we need to use a different set of reduction rules as Reduction Rule 10.1 does not preserve the number of solutions. We replace the reduction rules for this phase by an annotation procedure as used in the algorithms in Chapters 8 and 9. If no vertex can be annotated, the algorithm applies inclusion/exclusion-based branching to a hyperedge in $\mathcal{L}$ or $\mathcal{R}$. If this results in an instance that is sparse enough (as defined below), we go to Phase 3.

Let A be the set of annotated vertices in I. Note that all elements in Q in an instance in Phase 2 have degree two or three.

Reduction Rule 10.3 (Annotation). If there exists a vertex $v \in (Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A$ that has degree one in $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$, then put this vertex in the set of annotated vertices A.

Note that vertices that are annotated can correspond to elements in Q as well as hyperedges in $\mathcal{L}$ or $\mathcal{R}$. From now on, the degree of a vertex in I is considered to be its degree in $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$ unless explicitly specified otherwise.

When Reduction Rule 10.3 cannot be applied, the algorithm branches on a hyperedge. It uses the following order of selecting a hyperedge to branch on. Here, we let $s_i(S)$ be the number of elements in the hyperedge S of degree i in $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$, and we let $o(S)$ be the total number of appearances that elements in S have in the

hyperedge class not containing S. Also, let $\mathcal{E} \subseteq \mathcal{L} \cup \mathcal{R}$ be the set of hyperedges S with either $d_{(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A}(S) = 5$ and $s_3(S) \geq 3$, or $d_{(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A}(S) = s_3(S) = 4$.

1. If the maximum degree in $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$ is at least six, then choose a hyperedge S of maximum degree.
2. If the maximum degree in $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$ is at most five and $\mathcal{E} \neq \emptyset$, choose $S \in \mathcal{E}$ with $o(S)$ maximum over all $S \in \mathcal{E}$.
3. Otherwise, solve the subproblem in Phase 3.

The algorithm branches on the selected hyperedge S through inclusion/exclusion-based branching, see Section 8.1. The *optional* branch computes the number of 2-colourings that are indifferent about colouring S, i.e., in which S may or may not have received its right colour. In this branch, we remove only S. The *forbidden* branch computes the number of 2-colourings that do not colour S with the right colour. To do so, we colour all elements of S with the colour of the hyperedge class that does not contain S. As a result, we remove S, all elements of S, and all hyperedges that contain an element of S and are in the hyperedge class not containing S: these hyperedges have now received their corresponding colour.

After each branching, we solve both generated subproblems recursively. That is, we recursively apply the procedure of Phase 2 to them. We now compute the number of 2-colourings that correctly colour S by subtracting the result from the forbidden branch from the result from the optional branch.

Phase 3: Dynamic Programming on a Tree Decomposition. The algorithm solves generated subproblems of the counting variant of the problem by dynamic programming on a tree decomposition of I.

Note that all elements have degree two or three and all hyperedges have degree at most five, with some additional constraints on the elements contained in hyperedge in case their degree is four or five. We now compute a tree decomposition of $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$ using Proposition 2.16 [138]. We then remove the annotations in reverse order of the moment of annotation. By Proposition 8.2, this allows us to obtain a tree decomposition of I of the same width. Using this tree decomposition, we can count the number of 2-colourings and return these values to the appropriate leaves of the branching tree generated by Phases 1 and 2.

10.2.1. Analysis of the Running Time

We analyse the described algorithm using measure and conquer [144]. To this end, we introduce weight functions $v, w : \mathbb{N} \to \mathbb{R}_+$ and use the following measure k on a subproblem (H, A) with $H = (Q, \mathcal{L}, \mathcal{R})$:

$$k := \sum_{q \in Q \setminus A} v(d_{(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A}(q)) + \sum_{S \in (\mathcal{L} \cup \mathcal{R}) \setminus A} w(d_{(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A}(S))$$

In this case, we use the following weight functions. We choose these weight functions in order to minimise the running time resulting from the analysis below, that is, we computed these values as described in Section 5.6.

i	2	3	4	5	6	> 6
$v(i)$	0.448902	0.767484	0.934782	0.992583	1.000000	1.000000
$w(i)$	0.809607	0.963013	0.996566	1.000000	1.000000	1.000000

Similar to previous exponential-space inclusion/exclusion-based branching algorithms, we divide the running-time analysis of the described algorithm over a series of lemmas. First, Lemma 10.8 gives an upper bound on the number of subproblems generated by the branching in Phases 1 and 2. Thereafter, Lemma 10.9 gives an upper bound on the time that Phase 3 uses to solve such a generated subproblem. Finally, Lemma 10.10 combines both lemmas and proves an upper bound on the running time of the algorithm.

Lemma 10.8. *Let d be the dimension of an instance to which we apply the described algorithm. For each $h \leq d$, at most 1.20509^{d-h} subproblems of measure h are solved in Phase 3.*

Proof. Let $\Delta v(i) = v(i) - v(i-1)$ and $\Delta w(i) = w(i) - w(i-1)$ as before. Notice that the weight functions satisfy the following constraints:

1. $v(0) = v(1) = 0$
2. $\Delta v(i) \geq 0$ for all $i \geq 2$
3. $\Delta v(i) \geq \Delta v(i+1)$ for all $i \geq 2$
4. $w(0) = w(1) = 0$
5. $\Delta w(i) \geq 0$ for all $i \geq 2$
6. $\Delta w(i) \geq \Delta w(i+1)$ for all $i \geq 2$
7. $w(2) \geq 2\Delta w(5)$

Constraints 1 and 4 set the weights of elements and hyperedges that are removed or annotated by the reduction rules to zero. Constraints 2 and 5 ensure that the measure of an instance does not increase when we decrease the degree of an element or hyperedge during the branching in Phase 1 or 2, and Constraints 3 and 6 are the steepness inequalities that make the formulation of the problem easier. The role of Constraint 7 becomes clear from the analysis below.

Consider branching on an element q in an instance $(Q, \mathcal{L}, \mathcal{R})$ of measure k in Phase 1. Let I be the incidence graph of $(Q, \mathcal{L}, \mathcal{R})$, and notice that we have $A = \emptyset$ in this phase. Let l_i and r_i be the number of hyperedges in $\mathcal{L}$ and $\mathcal{R}$, respectively, that are of degree i in I and that contain q. Let Δk_l and Δk_r be the decrease in measure in the branch where we colour q with colour l and the branch where we colour q with colour r, respectively. We derive the following lower bounds on Δk_l and Δk_r:

$$\begin{aligned}
\Delta k_l &\geq v(d(q)) + \sum_{i=2}^{\infty}(l_i w(i) + r_i \Delta w(i)) + \Delta v(d(q)) \sum_{i=2}^{\infty}(i-1)l_i \\
&\quad + [r_2 > 0](v(2) - \Delta v(d(q))) \\
\Delta k_r &\geq v(d(q)) + \sum_{i=2}^{\infty}(r_i w(i) + l_i \Delta w(i)) + \Delta v(d(q)) \sum_{i=2}^{\infty}(i-1)r_i \\
&\quad + [l_2 > 0](v(2) - \Delta v(d(q)))
\end{aligned}$$

We show how to derive the first lower bound below; the second one can be derived similarly. In the branch where we assign colour l to q, the element q is removed, all hyperedges in $\mathcal{L}$ that contain q are removed, and all hyperedges in $\mathcal{R}$ that contain q have

their degree reduced. This explains the first two terms. Since we remove hyperedges in $\mathcal{L}$, other elements may have their degrees reduced also. Because of the steepness inequalities (Constraint 3) and the fact that q is of maximum degree, we can bound the decrease of the measure for each time that we reduce the degree of an element by $\Delta v(d(q))$. This gives an additional decrease of the measure of $\Delta v(d(q)) \sum_{i=2}^{\infty}(i-1)l_i$ explaining the third term.

The last term follows from the following arguments. If $r_2 > 0$, then there are hyperedges in $\mathcal{R}$ that get degree one after removing q. Consequently, Reduction Rule 10.2 will can be applied after q has been removed. Let q' be the unique element remaining in such a hyperedge and notice that q' will be assigned colour r, and thus will be removed. This leads to an additional decrease of the measure. In the worst case, all other occurrences of q' are in hyperedges in $\mathcal{L}$ that also contained q and thus have been removed already. Note that we have already included $\Delta v(d(q))$ per occurrence of q' in such a hyperedge in $\mathcal{L}$ in the bound on the decrease of the measure. To better bound the decrease in the measure due to removal of the element q', we observe that at most $(d(q')-1)\Delta v(d(q))$ measure is already taken from the total measure of $v(d(q'))$ that q' has. This leads to the additional decrease of the measure of $v(2) - \Delta v(d(q))$ since:

$$v(d(q')) = v(2) + \Delta v(3) + \ldots + \Delta v(d(q'))$$

which, due to Constraint 3, gives us:

$$v(d(q')) - (d(q')-1)\Delta v(d(q)) \geq v(2) - \Delta v(d(q))$$

This completes the proof of the correctness of Δk_l and Δk_r.

In Phase 2, the algorithm branches on hyperedges $S \in (\mathcal{L} \cup \mathcal{R})$. Recall that $s_i(S)$ denotes the elements in S of degree i in $I[(Q \cup \mathcal{L} \cup \mathcal{R}) \setminus A]$. For simplicity, we write $s_i = s_i(S)$. Let $\Delta k_{\text{optional}}$ be the decrease in measure in the optional branch, and $\Delta k_{\text{forbidden}}$ be the decrease in measure in the forbidden branch. Similar to the above, we find that $\Delta k_{\text{optional}}$ and $\Delta k_{\text{forbidden}}$ can be bounded by below as follows:

$$\begin{aligned}
\Delta k_{\text{optional}} &\geq w(d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S)) + \sum_{i=2}^{3} s_i \Delta v(i) \\
\Delta k_{\text{forbidden}} &\geq w(d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S)) + \sum_{i=2}^{3} s_i v(i) + \Delta w^* \sum_{i=2}^{3}(i-1)s_i
\end{aligned}$$

Here, $\Delta w^* = \Delta w(d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S))$ if $d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) \geq 6$ and $\Delta w^* = \Delta v(5)$ if $4 \leq d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) \leq 5$. We use this w^* because if the algorithm branches on a hyperedge S of degree four in $I[(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A]$, then S is not necessarily of maximum degree as hyperedges of degree five may still exist. We note that, in the above inequalities, we use Constraint 7 to obtain the third term in the formula with $\Delta k_{\text{forbidden}}$; this constraint makes sure that if the degree of a hyperedge S' is reduced to zero, then we still obtain $w(d(S')) - w(0) = \Delta w(d(S')) + \ldots + \Delta w(2) \geq (d(S')-2)\Delta w^* + \Delta w(2) \geq d(S')\Delta w^*$ as required.

For some specific cases, we will slightly increase the lower bound on $\Delta k_{\text{forbidden}}$ as described above. This is based on the following argument. Since we are in Phase 2, it may happen that elements in S occur only in hyperedges of the same hyperedge class

as S. In that case, however, $d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) \geq 6$ must hold by the selection criteria of the branching rule. This can be seen as follows. Suppose that $4 \leq d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) \leq 5$, then S must contain elements of degree three and these elements must already have been of degree three at the start of Phase 2. Consequently, they occur in hyperedges of both hyperedge classes. Let $\mathcal{T}(S)$ be the set of hyperedges that contain elements of S and that are in the hyperedge class that does not contain S. We conclude that in this case, $\mathcal{T}(S) \neq \emptyset$.

This means that in these specific cases where $4 \leq d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) \leq 5$, we can increase the values of $\Delta k_{\text{forbidden}}$. We remind the reader that $o(S)$ is the total number of appearances that elements in S have in the hyperedge class not containing S.

First, suppose $d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) = 5$. If $s_2 = 0$ and $s_3 = 5$, then $\mathcal{T}(S)$ cannot consist of a single hyperedge T as this would give us $10 = o(T) > 5 = o(S)$, and we would have branched on T instead of on S. Because of this, $\mathcal{T}(S)$ contains at least two hyperedges (that are all of degree at least two); Furthermore, if $\mathcal{T}(S)$ contains exactly two hyperedges, then one hyperedge must be of degree at least three. Since $3w(2) = \Delta w(2) + 2w(2) \geq \Delta w(3) + 2w(2) = w(2) + w(3)$, the latter case is the worst case. If $s_2 = 1$ and $s_3 = 4$, then, by a similar argument as above, we find that $\mathcal{T}(S)$ contains at least two hyperedges that are of degree at least two. If $s_2 = 2$ and $s_3 = 3$ then $\mathcal{T}(S)$ is guaranteed to have a hyperedge of degree at least three or two hyperedges of degree at least two. Since $2w(2) = \Delta w(2) + w(2) \geq \Delta w(3) + w(2) = w(3)$, the first case is the worst case. We note that these three subcases form the complete collection of cases that need to be considered with $d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) = 5$ as the branching rule will not branch on any other cases.

Finally, suppose $d_{(Q\cup\mathcal{L}\cup\mathcal{R})\setminus A}(S) = 4$. By the selection criteria of the branching rule, we must now have $s_2 = 0$ and $s_3 = 4$. We again find that $\mathcal{T}(S)$ is guaranteed to contain two hyperedges of degree at least two.

Summarizing, after correcting the double counting, we can add the following quantities to the lower bound for $\Delta k_{\text{forbidden}}$ in the following cases:

1. If $d(S) = 5, s_2 = 0, s_3 = 5$, then add $w(2) + w(3) - 5w^*$.
2. If $d(S) = 5, s_2 = 1, s_3 = 4$, then add $w(2) + w(2) - 4w^*$.
3. If $d(q) = 5, s_2 = 2, s_3 = 3$, then add $w(3) - 3w^*$.
4. If $d(q) = 4, s_2 = 0, s_3 = 4$, then add $w(2) + w(2) - 4w^*$.

This completes the description of the recurrence relations related to the branching.

Let $N_h(k)$ denote the number of subproblems of measure h created due to the branching in Phases 1 and 2 on an instance of measure k. We have:

$$\begin{aligned} N_h(k) &\leq N_h(k - \Delta k_l) + N_h(k - \Delta k_r) \\ N_h(k) &\leq N_h(k - \Delta k_{\text{optional}}) + N_h(k - \Delta k_{\text{forbidden}}). \end{aligned}$$

Using the given weight functions, we find an upper bound on the solution of this set of recurrence relations that satisfies $N_h(k) < 1.20509^{k-h}$.

Since $k \leq d$, this proves the bound of 1.20509^{d-h} on the number of subproblems of measure h. □

Next, we prove a bound on the running time used by Phase 3 to solve an instance generated by the branching in Phases 1 and 2. Recall that, in Phase 3, instance are solved by dynamic programming on a tree decomposition. This tree decomposition

is obtained by applying Proposition 2.16 [138] to $I[V \setminus A]$. Thereafter, this tree decomposition is modified to obtain a tree decomposition of I of the same width. This is done by removing the annotations in reverse order of the moment of annotation and using Proposition 8.2. Given such a tree decomposition of width t, we can solve such an instance in $\mathcal{O}^*(2^t)$ time by standard dynamic programming techniques on tree decompositions; see for example Chapter 11.

To prove a bound on the width of this tree decomposition, we use the formula in Proposition 2.16 [138] in a linear program to bound the pathwidth of $I[V \setminus A]$ in a generated instance with incidence graph I and set of annotated vertices A.

Lemma 10.9. *The number of 2-colourings of each 2-hypergraph H of measure h in Phase 3 can be computed in $\mathcal{O}(1.1904^h)$ time.*

Proof. As argued above, we can count the number of 2-colourings of H in $\mathcal{O}^*(2^t)$ time if we are given a tree decomposition of $I[V \setminus A]$ of width t where I is the incidence graph of an instance (H, A). Therefore, we prove this lemma by giving an upper bound on the pathwidth computed using Proposition 2.16 on an instance in Phase 3. This is done in a similar way as in Lemma 8.12 or in Theorem 9.4.

Let x_i and y_i represent the number of elements and hyperedges of degree i per unit the measure k in a worst case instance, respectively. Using Proposition 2.16, we can compute a path decomposition of $I[V \setminus A]$ of pathwidth at most $z \cdot h$ where z is the solution to the following linear program:

$$\begin{aligned}
\max \quad z &= \frac{1}{6}(x_3 + y_3) + \frac{1}{3}y_4 + \frac{13}{30}y_5 \qquad \text{such that:} \\
1 &= \sum_{i=2}^{3} v(i)x_i + \sum_{i=2}^{5} w(i)y_i && (10.1) \\
\sum_{i=2}^{3} i x_i &= \sum_{i=2}^{5} i y_i && (10.2) \\
x_2 &\geq \frac{1}{2}y_4 + \frac{3}{2}y_5 && (10.3)
\end{aligned}$$

In this linear program, all variables have the domain $[0, \infty)$. Recall that all unannotated elements are of degree 2 or 3 and that all unannotated hyperedges are of degree 2, 3, 4, or 5 in Phase 3. Hence, the given set of variables and the given objective function suffice to compute the required upper bound.

We can impose the given constraints due to the following argument. Constraint 10.1 guarantees that the variables use exactly one unit of the measure. Constraint 10.2 guarantees that both partitions of the bipartite incidence graph I are incident to the same number of edges, and Constraint 10.3 guarantees that the variables model sufficiently many elements of small degree if hyperedges of degree four or five are used. We can impose this constraint because certain hyperedges of degree four or five in $I[V \setminus A]$ are branched on in Phase 2 while others are allowed to pass to Phase 3 depending on the degrees of the elements contained in the hyperedge; see the definition of the branching rule in Phase 2 in Section 10.2. The constraint corresponds to the fact that every hyperedge of degree four contains at least one element of degree two, and every hyperedge of degree five contains at least three elements of degree two.

The solution to this linear program is $z = 0.251446$ with $x_2 = 0.251446$, $x_3 = 0.502892$, $y_4 = 0.502892$ and $y_2 = y_3 = y_5 = 0$. As a result, the computed tree composition has width at most $t \leq (0.251446 + \epsilon)h$, for any fixed $\epsilon > 0$. We choose ϵ sufficiently small such that the term $2^{\epsilon h}$ can be neglected due to the decimal rounding. We conclude that Phase 3 runs in $\mathcal{O}^*(2^{zh}) = \mathcal{O}(1.1904^h)$ time. □

Combining Lemmas 10.8 and 10.9 gives the result we wanted to prove.

Lemma 10.10. *There exists an algorithm that solves* 2-Hypergraph 2-Colouring *instances of dimension d in $\mathcal{O}(1.2051^d)$ time and space.*

Proof. Let $N_h(d)$ denote the number of subproblems of measure h generated by the branching of Phases 1 and 2 on an input of dimension d, and let H_d be the set of all possible measures of the subproblems that exist at the start of Phase 3.

If we combine the analysis of the branching in Phases 1 and 2 of Lemma 10.8 with the analysis of the dynamic programming in Phase 3 of Lemma 10.9, we find that the total running time $T(d)$ on an input of measure d equals:

$$T(d) \leq \sum_{h \in H_d} N_h(d) \cdot 1.1904^h \leq \sum_{h \in H_k} 1.2051^{d-h} \cdot 1.1904^h = \sum_{h \in H_d} 1.2051^d$$

We conclude that $T(d) = \mathcal{O}(1.2051^d)$ because we use only a finite number of weights which makes $|H_d|$ polynomially bounded. □

10.3. Concluding Remarks

We have given an $\mathcal{O}(1.2051^n)$-time algorithm for the 2-Disjoint Connected Subgraphs problem restricted to instances (G, Z_1, Z_2) where both Z_1 and Z_2 contain a connected set in G that dominates $V \setminus (Z_1 \cup Z_2)$. We have also showed how to use this algorithm to solve this problem within the same time bound on graphs in the class $\mathcal{G}^{k,2}$ for any fixed $k \geq 1$ and, in particular, for split graphs and P_6-free graphs

We leave it as an open question how to obtain a faster algorithm for graphs in the classes $\mathcal{G}^{k,r}$ with $r \geq 3$. Another natural question is to study the class of instances (G, Z_1, Z_2) where only one of the subsets, say Z_1, contains a connected set in G that dominates $U = V \setminus (Z_1 \cup Z_2)$. For solving this problem, a similar approach as in [313] may be followed, where brute force techniques are applied depending on the size of Z_1 and Z_2. Another approach would be to apply an algorithm that lists all minimal set covers (similar to [146]). By using such an approach one can enumerate all sets $U' \subseteq U$ that are minimal with respect to dominating Z_1. For each choice of U' one can check in polynomial time if $G[Z_2 \cup (U' \setminus U)]$ is connected. We note that this approach also works for instances where both Z_1 and Z_2 contain a connected set in G that dominates $V \setminus (Z_1 \cup Z_2)$, but this leads to much worse running times than presented in this chapter.

The main open question is to find an exact algorithm for the 2-Disjoint Connected Subgraphs problem on general graphs that is faster than the trivial $\mathcal{O}^*(2^n)$ algorithm. For solving this problem, new techniques that deal with the connectivity issue are necessary.

Another interesting issue is whether there are more problems for which we can give faster exact exponential-time algorithms by using the two-phase approach used in this chapter. That is, are there more problems for which it is beneficial to first consider a branch-and-reduce procedure on the decision problem and, in a later phase, consider a branch-and-reduce procedure on the counting problem using inclusion/exclusion-based branching. This approach allows the combination of the use of more powerful reduction rules that do not apply to the counting problem with the approach of using inclusion/exclusion-based branching.

IV

Dynamic Programming Algorithms on Graph Decompositions

11 Fast Dynamic Programming on Tree Decompositions

Width parameters of graphs and their related graph decompositions are important in the theory of graph algorithms. Many investigations show that problems that are $\mathcal{NP}$-hard on general graphs become polynomial or even linear-time solvable when restricted to the class of graphs in which a given width parameter is bounded. However, the constant factors involved in the upper bound on the running times of such algorithms are often large and depend on the parameter. Therefore, it is often useful to find algorithms where these factors grow as slow as possible as a function of the graph parameter k.

In this thesis, we consider such algorithms involving three prominent graph-width parameters and their related decompositions. We consider *treewidth* and *tree decompositions* in this chapter; Chapter 12 considers *branchwidth* and *branch decompositions*; and, Chapter 13 considers *cliquewidth* and *k-expressions* or *clique decompositions*. These three graph-width parameters are probably the most commonly used ones in the literature. However, other parameters such as rankwidth [248] or booleanwidth [62] and their related decompositions also exist.

Most algorithms solving combinatorial problems using a graph-width parameter consist of two steps:

1. Find a graph decomposition of the input graph of small width.
2. Solve the problem by dynamic programming on this graph decomposition.

In this chapter and the next two, we will focus on the second of these steps and improve the running time of many known algorithms on all three discussed types of graph decomposition as a function of the width parameter.

[†]This chapter is joint work with Hans L. Bodlaender and Peter Rossmanith. The chapter contains results of which a preliminary version has been presented at the 17th Annual European Symposium on Algorithms (ESA 2009) [305].

Algorithms for Tree Decompositions. Concerning the first step of the general two-step approach above, we note that finding a tree decomposition of minimum treewidth is $\mathcal{NP}$-hard [10]. For fixed k, one can find a tree decomposition of width at most k in linear time, if such a decomposition exists [37]. However, the constant factor involved in this algorithm is very high. On the other hand, tree decompositions of small width can be obtained efficiently for special graph classes [39], and there are also several good heuristics that often work well in practice [45]. When tree decompositions need to be constructed for obtaining fast exact exponential-time algorithms, Theorem 2.14 [147], Proposition 2.16 [138], and Proposition 2.18 [204] can be used. Also, approximation algorithms for treewidth that construct tree decompositions can be found in [6, 43, 52].

Concerning the second step of this two-step approach, many $\mathcal{NP}$-hard problems can be solved in polynomial time on a graph G whose treewidth is bounded by a constant. If we assume that a graph G is given with a tree decomposition T of G of width k, then the running time of such an algorithm is typically polynomial in the size of graph G, but exponential in the treewidth k. Examples of such algorithms include many kinds of vertex partitioning problems (including many graph domination problems such as the $[\rho, \sigma]$-domination problems) [296], edge colouring problems such as Chromatic Index [35], or other problems such as Steiner Tree [211].

There are several recent results about the running time of algorithms on tree decompositions, with special considerations for the running time as function of the width of the tree decomposition k. For several vertex partitioning problems, Telle and Proskurowski showed that there are algorithms that, given a graph with a tree decomposition of width k, solve these problems in $\mathcal{O}(c^k n)$ time [296], where c is a constant that depends only on the problem at hand. For Dominating Set, Alber and Niedermeier gave an improved algorithm that runs in $\mathcal{O}(4^k n)$ time [3]. Similar results are given in [2] for related problems: Independent Dominating Set, Total Dominating Set, Perfect Dominating Set, Perfect Code, Total Perfect Dominating Set, Red-Blue Dominating Set and weighted versions of these problems.

If the input graph is planar, then other improvements are possible. Dorn showed that Dominating Set on planar graphs given with a tree decomposition of width k can be solved in $\mathcal{O}^*(3^k)$ time [112]; he also gave similar improvements for other problems. We obtain the same result without requiring planarity.

Our Results. In this chapter, we show that the number of dominating sets of each given size in any graph can be counted in $\mathcal{O}^*(3^k)$ time. After some modifications, this gives an $\mathcal{O}(nk^2 3^k)$-time algorithm for Dominating Set. We also show that one can count the number of perfect matchings in a graph in $\mathcal{O}^*(2^k)$ time, and we generalise these results to the $[\rho, \sigma]$-domination problems (see Section 1.6).

For these $[\rho, \sigma]$-domination problems, we show that they can be solved in $\mathcal{O}^*(s^k)$ time, where s is the natural number of states required to represent a partial solution. The only restriction that we impose on these problems is that we require both ρ and σ to be either finite or cofinite. That such an assumption is necessary follows from Chappelle's recent result [70]: he shows that $[\rho, \sigma]$-domination problems are $\mathcal{W}[1]$-hard when parameterised by the treewidth of the graph if σ is allowed to have arbitrarily large gaps between consecutive elements and ρ is cofinite. The problems to which our results apply include Strong Stable Set, Independent Dominating Set, Total Dominating Set, Total Perfect Dominating Set, Perfect Code, Induced

p-Regular Subgraph, and many others. Our results also extend to other similar problems such as Red-Blue Dominating Set.

Finally, we also define families of problems that we call γ-clique covering, packing, or partitioning problems: these families generalise standard problems like Minimum Clique Partition in the same way as the $[\rho, \sigma]$-domination problems generalise Dominating Set. The resulting families of problems include Maximum Triangle Packing, Partition Into l-Cliques for fixed l, the problem to determine the minimum number of odd-size cliques required to cover G, and many others. For these γ-clique covering, packing, or partitioning problems, we give $\mathcal{O}^*(2^k)$-time algorithms.

Optimality, Polynomial Factors, and Model of Computation. We note that our results attain, or are very close to, intuitive natural lower bounds for the problems considered, namely a polynomial times the amount of space used by any dynamic programming algorithm for these problems on graph decompositions. Similarly, it makes sense to think about the number of states necessary to represent partial solutions as the best possible base of the exponent in the running time: this equals the space requirements. Currently, this is $\mathcal{O}^*(3^k)$ for Dominating Set on tree decompositions.

Very recently, this intuition has been strengthened by a result of Lokshtanov et al. [227]. They prove that it is impossible to improve the exponential part of the running time for a number of tree-decomposition-based algorithms that we present in this chapter, unless the *Strong Exponential-Time Hypothesis* fails. That is, unless there exist an algorithm for the general Satisfiability problem running in $\mathcal{O}((2-\epsilon)^n)$ time for any $\epsilon > 0$; see Section 3.2. In particular, this holds for our algorithms for Dominating Set and Partition Into Triangles.

Because of these seemingly optimal exponential factors in the running times of our algorithms, we spend quite some effort to make the polynomial factors involved as small as possible. Also, because many of our algorithms use numbers which require more than a constant number of bits to represent (often n-bit numbers are involved), the time and space required to represent these numbers and perform arithmetic operations on these numbers affects the polynomial factors in the running times of our algorithms. We will always include these factors and highlight them using a special notation.

Notation $(i_+(n), i_\times(n))$. We denote the time required to add and multiply n-bit numbers by $i_+(n)$ and $i_\times(n)$, respectively.

Currently, $i_\times(n) = n\log(n)2^{\mathcal{O}(\log^*(n))}$ due to Fürer's algorithm [159], and $i_+(n) = \mathcal{O}(n)$.

In this chapter and the next two, we use the *Random Access Machine* (RAM) model with $\mathcal{O}(k)$-bit word size [157] for the analysis of our algorithms. In this model, memory access can performed in constant time for memory of size $\mathcal{O}(c^k)$ for any constant c. We consider addition and multiplication operations on $\mathcal{O}(k)$-bit numbers to be unit-time operations ($i_+(k) = i_\times(k) = 1$). For an overview of this model, see for example [176].

We use this computational model because we do not want the table look-up operations to influence the polynomial factors of the running time. Since the tables have size $\mathcal{O}^*(s^k)$, for a problem-specific integer $s \geq 2$, these operations are constant-time operations in this model. We note that the word size used in the computational model was not an issue before in this thesis, because we were not interested in the polynomial factors involved in the exact exponential-time algorithms in previous chapters.

Fast Subset Convolution. We obtain our results by using variants of the *covering product* and the *fast subset convolution* algorithm [28] in conjunction with known techniques on graph decompositions. An important aspect of our results is an implicit generalisation of the fast subset convolution algorithm that is able to use multiple states. This contrasts to the set formulation in which the covering product and subset convolution are defined: this formulation is equivalent to using two states (in and out). Moreover, the fast subset convolution algorithm uses ranked Möbius transforms, while we obtain our results by using transformations that use multiple states and multiple ranks. It is interesting to note that the state-based convolution technique that we use reminds of the technique used in Strassen's algorithm for fast matrix multiplication [289].

Given a set U and functions $f, g : 2^U \to \mathbb{Z}$, their *subset convolution* $(f * g)$ is defined as follows:

$$(f * g)(S) = \sum_{X \subseteq S} f(X)g(S \setminus X)$$

The fast subset convolution algorithm by Björklund et al. can compute this convolution using $\mathcal{O}(k^2 2^k)$ arithmetic operations [28].

Similarly, Björklund et al. define the *covering product* $(f *_c g)$ and the *packing product* $(f *_p g)$ of f and g in the following way:

$$(f *_c g)(S) = \sum_{\substack{X,Y \subseteq S \\ X \cup Y = S}} f(X)g(Y) \qquad (f *_p g)(S) = \sum_{\substack{X,Y \subseteq S \\ X \cap Y = \emptyset}} f(X)g(Y)$$

These products can be computed using $\mathcal{O}(k2^k)$ arithmetic operations [28].

The fast subset convolution algorithm and similar algorithms for the covering and packing products have been used to speed up other dynamic programming algorithms before, but not in the setting of graph-decompositions. Examples include Steiner Tree [28, 244], graph motif problems [20], and graph recolouring problems [254].

In this thesis, we will not directly use the algorithms of Björklund et al. as subroutines. Instead, we present their algorithms based on what we will call *state changes*. This approach does exactly the same as using the algorithms by Björklund et al. as subroutines. We choose to present it in our own way because this allows us to easily generalise the fast subset convolution algorithms to a more complex setting than functions with domain 2^U for some set U.

Organisation of the Chapter. This chapter is organised in the following way. We begin with an introduction to dynamic programming on tree decompositions in Section 11.1. In this section, we give some definitions and an example algorithm for Dominating Set. Thereafter, we define what we call the *de Fluiter property* for treewidth and discuss its relations to other properties given in the literature in Section 11.2. In the following sections, we give a series of faster dynamic programming algorithms on tree decompositions. We give a faster algorithm for Dominating Set in Section 11.3, a faster algorithm for #Perfect Matching in Section 11.4, faster algorithms for the $[\rho, \sigma]$-domination problems in Section 11.5, and faster algorithms for a series of clique covering, packing, and partitioning problems in Section 11.6. Finally, we give some concluding remarks in Section 11.7.

11.1. Introduction to Treewidth-Based Algorithms

Tree-decomposition-based algorithms can be used to effectively solve combinatorial problems on graphs of small treewidth both in theory and in practice. Practical algorithms exist for problems like partial constraint satisfaction [212]. Furthermore, tree-decomposition-based algorithms are used as subroutines in many areas such as approximation algorithms [109, 121], parameterised algorithms [108, 237, 297], exact exponential-time algorithms [138, 284, 307] (see also Section 2.2.2 and Chapters 8-10), and subexponential-time algorithms [51, 153]. For an overview of tree decompositions and dynamic programming on tree decompositions, see [44, 184].

In this section, we introduce the reader to some important ideas of treewidth-based algorithms. First, we give some definitions in Section 11.1.1. Thereafter, we present an example of a dynamic programming algorithm of graphs given with a tree decomposition of width k in Section 11.1.2. This example algorithm will be the basis for all other algorithms presented in this chapter.

11.1.1. Definitions

The notions of tree decomposition and treewidth were introduced by Robertson and Seymour [267]. We recall the definition of a tree decomposition from Section 2.2.2. We note that, for a decomposition tree T, we often identify T with the set of nodes in T, and we write $E(T)$ for the edges of T.

Definition 11.1 (Tree Decomposition). A *tree decomposition* of a graph $G = (V, E)$ consists of a tree T in which each node $x \in T$ has an associated set of vertices $X_x \subseteq V$ (called a *bag*) such that $\bigcup_{x \in T} X_x = V$ and the following properties hold:

1. for each $\{u, v\} \in E$, there exists an X_x such that $\{u, v\} \subseteq X_x$.
2. if $v \in X_x$ and $v \in X_y$, then $v \in X_z$ for all nodes z on the path from node x to node y in T.

The *width* $tw(T)$ of a tree decomposition T is the size of the largest bag of T minus one. The treewidth $tw(G)$ of a graph G is the minimum width over all possible tree decompositions of G. Note that treewidth of trees is one. In this chapter, we will always assume that tree decompositions of the appropriate width are given.

Dynamic programming algorithms on tree decompositions are often presented on nice tree decompositions, which were introduced by Kloks [202]. We give a slightly different definition of a nice tree decomposition.

Definition 11.2 (Nice Tree Decomposition). A *nice tree decomposition* is a tree decomposition with one special node z called the *root* with $X_z = \emptyset$ and in which each node is of one of the following types:

1. *Leaf node*: a leaf x of T with $X_x = \{v\}$ for some vertex $v \in V$.
2. *Introduce node*: an internal node x of T with one child node y; this type of node has $X_x = X_y \cup \{v\}$, for some $v \notin X_y$. The node is said to *introduce* the vertex v.
3. *Forget node*: an internal node x of T with one child node y; this type of node has $X_x = X_y \setminus \{v\}$, for some $v \in X_y$. The node is said to *forget* the vertex v.
4. *Join node*: an internal node x with two child nodes l and r; this type of node has $X_x = X_r = X_l$.

We note that this definition is slightly different from the usual definition. In our definition, we have the extra requirements that a bag X_x associated with a leaf x of T consists of a single vertex v ($X_x = \{v\}$), and that the bag X_z associated with the root node Z is empty ($X_z = \emptyset$).

Given a tree decomposition consisting of $O(n)$ nodes, a nice tree decomposition of equal width and also consisting of $\mathcal{O}(n)$ nodes can be found in $O(n)$ time [202]. By adding a series of forget nodes to the old root, and by adding a series of introduce nodes below an old leaf node if its associated bag contains more than one vertex, we can easily modify any nice tree decomposition to have our extra requirements within the same running time.

By fixing the root of T, we associate with each node x in a tree decomposition T a vertex set $V_x \subseteq V$: a vertex v belongs to V_x if and only if there exists a bag y with $v \in X_y$ such that either $y = x$ or y is a descendant of x in T. Furthermore, we associate with each node x of T the induced subgraph $G_x = G[V_x]$ of G. I.e., G_x is the following graph:

$$G_x = G\Big[\bigcup\{X_y \mid y = x \text{ or } y \text{ is a descendant of } x\}\Big]$$

In the algorithms given in this chapter, we often associate a table A_x with each node $x \in T$. We denote the number of entries in A_x by $|A_x|$.

11.1.2. An Example Algorithm for Dominating Set

Algorithms solving $\mathcal{NP}$-hard problems in polynomial time on graphs of bounded treewidth are often dynamic programming algorithms of the following form. The tree decomposition T is traversed in a bottom-up manner. For each node $x \in T$ visited, the algorithm constructs a table with partial solutions on the subgraph G_x, that is, the induced subgraph on all vertices that are in a bag X_y where $y = x$ or y is a descendant of x in T. Let an *extension* of such a partial solution be a solution on G that contains the partial solution on G_x, and let two such partial solutions P_1, P_2 have the same *characteristic* if any extension of P_1 also is an extension of P_2 and vice versa. The table for a node $x \in T$ does not store all possible partial solutions on G_x: it stores a set of solutions such that it contains exactly one partial solution for each possible characteristic. While traversing the tree T, the table for a node $x \in T$ is computed using the tables that had been constructed for the children of x in T.

This type of algorithm typically has a running time of the form $\mathcal{O}(f(k)poly(n))$ or even $\mathcal{O}(f(k)n)$, for some function f that grows at least exponentially. This is because the size of the computed tables often is (at least) exponentially in the treewidth k of T, but polynomial (or even constant) in the size of the graph G.

We now give a simple dynamic programming algorithm for Dominating Set; see Proposition 11.3. We note that a faster algorithm for this problem exists [2, 3], and an even faster algorithm will be given in Section 11.3. The presented algorithm follows from standard techniques for treewidth-based algorithms. Many of the details of the algorithm described below also apply to other algorithms described in this chapter. We will not repeat these details: for the other algorithms, we will specify only how to compute the tables associated with each node of a nice tree decomposition, for all four kinds of nodes.

state	meaning
1	this vertex is in the dominating set.
0_1	this vertex is not in the dominating set and has already been dominated.
0_0	this vertex is not in the dominating set and has not yet been dominated.
$0_?$	this vertex is not in the dominating set and may or may not be dominated.

Table 11.1. Vertex states for the DOMINATING SET problem.

Proposition 11.3. *There is an algorithm that, given a tree decomposition of a graph G of width k, computes the size of a minimum dominating set in G in $\mathcal{O}(n5^k i_+(\log(n)))$ time.*

Proof. First, we construct a nice tree decomposition T of G of width k from the given tree decomposition in $\mathcal{O}(n)$ time.

Similar to Telle and Proskurowski [296], we introduce vertex states 1, 0_1, and 0_0 that characterise the 'state' of a vertex with respect to a vertex set D that is a partial solution of the DOMINATING SET problem: v has state 1 if $v \in D$; v has state 0_1 if $v \notin D$ but v is dominated by D, i.e., there is a $d \in D$ with $\{v, d\} \in E$; and, v has state 0_0 if $v \notin D$ and v is not dominated by D; see also Table 11.1.

For each node x in the nice tree decomposition T, we consider partial solutions $D \subseteq V_x$, such that all vertices in $V_x \setminus X_x$ are dominated by D. We characterise these sets D by the states of the vertices in X_x and the size of D. More precisely, we will compute a table A_x with an entry $A_x(c) \in \{0, 1, \ldots, n\} \cup \{\infty\}$ for each $c \in \{1, 0_1, 0_0\}^{|X_x|}$. We call $c \in \{1, 0_1, 0_0\}^{|X_x|}$ a *colouring* of the vertices in X_x. A table entry $A_x(c)$ represents the size of the partial solution D of DOMINATING SET in the induced subgraph G_x associated with the node x of T that satisfies the requirements defined by the states in the colouring c, or infinity if no such set exists. That is, the table entry gives the size of the smallest partial solution D in G_x that contains all vertices in X_x with state 1 in c and that dominates all vertices in G_x except those in X_x with state 0_0 in c, or infinity if no such set exists. Notice that these $3^{|X_x|}$ colourings correspond to $3^{|X_x|}$ partial solutions with different characteristics, and that it contains a partial solution for each possible characteristic.

We now show how to compute the table A_x for the next node $x \in T$ while traversing the nice tree decomposition T in a bottom-up manner. Depending on the type of the node x (see Definition 11.2), we do the following:

Leaf node: Let x be a leaf node in T. The table consists of three entries, one for each possible colouring $c \in \{1, 0_1, 0_0\}$ of the single vertex v in X_x.

$$A_x(\{1\}) = 1 \qquad A_x(\{0_1\}) = \infty \qquad A_x(\{0_0\}) = 0$$

Here, $A_x(c)$ corresponds to the size of the smallest partial solution satisfying the requirements defined by the colouring c on the single vertex v.

Introduce node: Let x be an introduce node in T with child node y. We assume that when the l-th coordinate of a colouring of X_x represents a vertex u, then the same coordinate of a colouring of X_y also represents u, and that the last coordinate of a

colouring of X_x represents the newly introduced vertex v. Now, for any colouring $c \in \{1, 0_1, 0_0\}^{|X_y|}$:

$$A_x(c \times \{0_1\}) = \begin{cases} A_y(c) & \text{if } v \text{ has a neighbour with state 1 in } c \\ \infty & \text{otherwise} \end{cases}$$

$$A_x(c \times \{0_0\}) = \begin{cases} A_y(c) & \text{if } v \text{ has no neighbour with state 1 in } c \\ \infty & \text{otherwise} \end{cases}$$

For colourings with state 1 for the introduced vertex, we say that a colouring c_x of X_x *matches* a colouring c_y of X_y if:

- For all $u \in X_y \setminus N(v)$: $c_x(u) = c_y(u)$.
- For all $u \in X_y \cap N(v)$: either $c_x(u) = c_y(u) = 1$, or $c_x(u) = 0_1$ and $c_y(u) \in \{0_1, 0_0\}$.

Here, $c(u)$ is the state of the vertex u in the colouring c. We compute $A_x(c)$ by the following formula:

$$A_x(c \times \{1\}) = \begin{cases} \infty & \text{if } c(u) = 0_0 \text{ for some } u \in N(v) \\ 1 + \min\{A_y(c') \mid c' \text{ matches } c\} & \text{otherwise} \end{cases}$$

It is not hard to see that $A_x(c)$ now corresponds to the size of the partial solution satisfying the requirements imposed on X_x by the colouring c..

Forget node: Let x be a forget node in T with child node y. Again, we assume that when the l-th coordinate of a colouring of X_x represents a vertex u, then the same coordinate of a colouring of X_y also represents u, and that the last coordinate of a colouring of X_y represents vertex v that we are forgetting.

$$A_x(c) = \min\{A_y(c \times \{1\}), A_y(c \times \{0_1\})\}$$

Now, $A_x(c)$ corresponds to the size of the smallest partial solution satisfying the requirements imposed on X_x by the colouring c as we consider only partial solutions that dominate the forgotten vertex.

Join node: Let x be a join node in T and let l and r be its child nodes. As $X_x = X_l = X_r$, we can assume that the same coordinates represent the same vertices in a colouring of each of the three bags.

Let $c_x(v)$ be the state that represents the vertex v in colouring c_x of X_x. We say that three colourings c_x, c_l, and c_r of X_x, X_l, and X_r, respectively, *match* if for each vertex $v \in X_x$:

- either $c_x(v) = c_l(v) = c_r(v) = 1$,
- or $c_x(v) = c_l(v) = c_r(v) = 0_0$,
- or $c_x(v) = 0_1$ while $c_l(v)$ and $c_r(v)$ are 0_1 or 0_0, but not both 0_0.

Notice that three colourings c_x, c_l, and c_r match if for each vertex v the requirements imposed by the states are correctly combined from the states in the colourings on both child bags c_l and c_r to the states in the colourings of the parent bag c_x. That is, if a vertex is required by c_x to be in the vertex set of a partial solution, then it is also required to be so in c_l and c_r; if a vertex is required to be undominated in c_x, then it is also required to be undominated in c_l and c_r; and, if a vertex is required to be not in the partially constructed dominating set but it is required to be dominated in c_x,

then it is required not to be in the vertex sets of the partial solutions in both c_l and c_r, but it must be dominated in one of both partial solutions.

The new table A_x can be computed by the following formula:

$$A_x(c_x) = \min_{c_x, c_l, c_r \text{ match}} A_l(c_l) + A_r(c_r) - \#_1(c_x)$$

Here, $\#_1(c)$ stands for the number of 1-states in the colouring c. This number needs to be subtracted from the total size of the partial solution because the corresponding vertices are counted in each entry of $A_l(c_l)$ as well as in each entry of $A_r(c_r)$. One can easily check that this gives a correct computation of A_x.

After traversing the nice tree decomposition T, we end up in the root node $z \in T$. As $X_z = \emptyset$ and thus $G_z = G$, we find the size of the minimum dominating set in G in the single entry of A_z.

It is not hard to see that the algorithm stores the size of the smallest partial solution of Dominating Set in A_x for each possible characteristic on X_x for every node $x \in T$. Hence, the algorithm is correct.

For the running time, observe that, for a leaf or forget node, $\mathcal{O}(3^{|X_x|} i_+(\log(n)))$ time is required since we work with $\log(n)$-bit numbers. In an introduce node, we need more time as we need to inspect multiple entries from A_y to compute A_x. For a vertex u outside $N(v)$, we have three possible combinations of states, and for a vertex $u \in N(v)$ we have four possible combinations we need to inspect: the table entry with $c_x(u) = c_y(u) = 0_0$, colourings with $c_x(u) = c_y(u) = 1$, and colourings with $c_x(u) = 0_1$ while $c_y(u) = 0_0$ or $c_y(u) = 0_1$. This leads to a total time of $\mathcal{O}(4^{|X_x|} i_+(\log(n)))$ for an introduce node. In a join node, five combinations of states need to be inspected per vertex requiring $\mathcal{O}(5^{|X_x|} i_+(\log(n)))$ time in total. As the largest bag has size at most $k+1$ and the tree decomposition T has $\mathcal{O}(n)$ nodes, the running time is $\mathcal{O}(n 5^k i_+(\log(n)))$. □

We notice that the above algorithm computes only the size of a minimum dominating set in G, not the dominating set itself. To construct a minimum dominating set D, the tree decomposition T can be traversed in top-down order (reverse order compared to the algorithm of Proposition 11.3). We start by selecting the single entry in the table of the root node, and then, for each child node y of the current node x, we select an the entry in A_y which was used to compute the selected entry of A_x. More specifically, we select the entry that was either used to copy into the selected entry of A_x, or we select one, or in a join node two, entries that lead to the minimum that was computed for A_x. In this way, we trace back the computation path that computed the size of D. During this process, we construct D by adding each vertex that is not yet in D and that has state 1 in c to D. As we use only colourings that lead to a minimum dominating set, this process gives us a minimum dominating set in G.

11.2. De Fluiter Property for Treewidth

Before we give a series of new, fast dynamic programming algorithms for a broad range of problems, we need the following definition. We use it to improve the polynomial factors involved in the running times of our algorithms in the rest of this chapter.

Definition 11.4 (De Fluiter Property for Treewidth). Given a graph-optimisation problem Π, consider a method to represent the different characteristics of partial solutions used in an algorithm that performs dynamic programming on a tree decomposition to solve Π. Such a representation of partial solutions has the *de Fluiter property for treewidth* if the difference between the objective values of any two partial solutions of Π that are associated with a different characteristic and can both still be extended to an optimal solution is at most $f(k)$, for some non-negative function f that depends only on treewidth k.

This property is named after Babette van Antwerpen-de Fluiter, as this property implicitly plays an important role in her work reported in [49, 103]. Note that although we use the value ∞ in our dynamic programming tables, we do not consider such entries since they can never be extended to an optimal solution. Hence, these entries do not influence the de Fluiter property. Furthermore, we say that a problem has the *linear de Fluiter property for treewidth* if f is a linear function in k.

Consider the representation used in Proposition 11.3 for the Dominating Set problem. This representation has the de Fluiter property for treewidth with $f(k) = k + 1$ because any table entry that is more than $k + 1$ larger than the smallest value stored in the table cannot lead to an optimal solution. This holds because any partial solution of Dominating Set D that is more than $k+1$ larger than the smallest value stored in the table cannot be part of a minimum dominating set. Namely, we can obtain a partial solution that is smaller than D and that dominates the same vertices or more by taking the partial solution corresponding to the smallest value stored in the table and adding all vertices in X_x to it.

The de Fluiter property for treewidth is highly related to the concept *finite integer index* as defined in [49]. Finite integer index is a property used in reduction algorithms for optimisation problems on graphs of small treewidth [49] and is also used in meta-results in the theory of kernelisation [42]. We will conclude this section by explaining the relation between the de Fluiter property and finite integer index. We note that we treat this relation only to link the de Fluiter property to the literature; one does not need to understand the details of this relation to understand the new dynamic programming algorithms on tree decompositions presented in this chapter.

Let a *terminal graph* be a graph G together with an ordered set of distinct vertices $X = \{x_1, x_2, \ldots, x_l\}$ with each $x_i \in V$. The vertices $x_i \in X$ are called the *terminals* of G. For two terminal graphs G_1 and G_2 with the same number of terminals, the addition operation $G_1 + G_2$ is defined to be the operation that takes the disjoint union of both graphs, then identifies each pair of terminals with the same number $1, 2, \ldots, t$, and finally removes any double edges created.

For a graph optimisation problem Π, Bodlaender and van Antwerpen-de Fluiter [49] define an equivalence relation $\sim_{\Pi,l}$ on terminal graphs with l terminals: $G_1 \sim_{\Pi,l} G_2$ if and only if there exists an $i \in \mathbb{Z}$ such that for all terminal graphs H with l terminals:

$$\pi(G_1 + H) = \pi(G_2 + H) + i$$

Here, the function $\pi(G)$ assigns the objective value of an optimal solution of the optimisation problem Π to the input graph G.

Definition 11.5 (Finite Integer Index). An optimisation problem Π is of *finite integer index* if $\sim_{\Pi,l}$ has a finite number of equivalence classes for each fixed l.

When one proves that a problem has finite integer index, one often gives a representation of partial solutions that has the de Fluiter property for treewidth; see for example [103]. This is correct as one can see from the following proposition.

Proposition 11.6. *If a problem Π has a representation of its partial solutions of different characteristics that can be used in an algorithm that performs dynamic programming on tree decompositions and that has the de Fluiter property for treewidth, then Π is of finite integer index.*

Proof. Let l be fixed, and consider an l-terminal graph G. Construct a tree decomposition T of G such that the bag associated with the root of T equals the set of terminals X of G. Note that this is always possible since we have not specified a bound on the treewidth of T. For an l-terminal graph H, one can construct a tree decomposition of $G + H$ by making a similar tree decomposition of H and identifying the roots, which both have the same vertex set X.

Let G_1, G_2 be two l-terminal graphs to which we both add another l-terminal graph H through addition, i.e, $G_i + H$, and let T_1, T_2 be tree decompositions of these graphs obtained in the above way. For both graphs, consider the dynamic programming table constructed for the node x_X associated with the vertex set X by a dynamic programming algorithm for Π that has the de Fluiter property for treewidth. For these tables, we assume that the induced subgraph associated with x_X of the decompositions equals G_i, that is, the bags of the nodes below x_X contain all vertices in G_i and vertices in H occur only in bags associated with nodes that are not descendants of x_X in T_i.

Clearly, $\pi(G_1 + H) = \pi(G_2 + H)$ if both dynamic programming tables are the same and $G_1[X] = G_2[X]$, that is, if the tables are equal and both graphs have the same edges between their terminals. Let us now consider a more general case where we first normalise the dynamic programming tables such that the smallest-valued entry equals zero, and all other entries contain the difference in value to this smallest entry. In this case, it is not hard to see that if both normalised dynamic programming tables are equal and $G_1[X] = G_2[X]$, then there must exist an $i \in \mathbb{Z}$ such that $\pi(G_1 + H) = \pi(G_2 + H) + i$.

The dynamic programming algorithm for the problem Π can compute only finite-size tables. Moreover, as the representation used by the algorithm has the de Fluiter property for treewidth, the normalised tables can have only values in the range $0, 1, \ldots, f(k)$. Therefore, there are only a finite number of different normalised tables and a finite number of possible induced subgraphs on l vertices (terminals). We conclude that the relation $\sim_{\Pi,l}$ has a finite number of equivalence classes. □

The converse of Proposition 11.6 is not necessarily true.

An example of a problem for which there is no representation of partial solutions that has the de Fluiter property for treewidth is Independent Dominating Set. It is not hard to see that the equivalence relation $\sim_{\Pi,l}$ corresponding to this problem can have an infinite amount of equivalence classes. That no representation of partial solutions that has the de Fluiter property exists follows from this observation.

11.3. Minimum Dominating Set

Alber et al. showed that one can improve the classical result of Proposition 11.3 by choosing a different set of states to represent characteristics of partial solutions [2, 3]: they obtained an $\mathcal{O}^*(4^k)$ algorithm using the set of states $\{1, 0_1, 0_?\}$ (see Table 11.1 in Section 11.1.2). In this section, we obtain an $\mathcal{O}^*(3^k)$ algorithm by using yet another set of states, namely $\{1, 0_0, 0_?\}$.

Note that $0_?$ represents a vertex v that is not in the vertex set D of a partial solution of Dominating Set, while we do not specify whether v is dominated, i.e., given D, vertices with state 0_1 and with state 0_0 could also have state $0_?$. In particular, there is no longer a unique colouring of X_x with states for a specific partial solution: a partial solution can correspond to several such colourings. Below, we discuss in detail how we can handle this situation and how it can lead to faster algorithms.

Since the state 0_0 represents an undominated vertex and the state $0_?$ represents a vertex that may or may not be dominated, one may think that it is impossible to guarantee that a vertex is dominated using these states. We circumvent this problem by not just computing the *size* of a minimum dominating set, but by computing the *number* of dominating sets of each fixed size κ with $0 \leq \kappa \leq n$. This approach does not store (the size of) a solution per characteristic of the partial solutions, but counts the number of partial solutions of each possible size per characteristic. We note that the algorithm of Proposition 11.3 can straightforwardly be modified to also count the number of (minimum) dominating sets.

For our next algorithm, we use dynamic programming tables in which an entry $A_x(c, \kappa)$ represents the number of partial solutions of Dominating Set on G_x of size exactly κ that satisfy the requirements defined by the states in the colouring c. That is, the table entries give the number of partial solution in G_x of size κ that dominate all vertices in $V_x \setminus X_x$ and all vertices in X_x with state 0_1, and that do not dominate the vertices in X_x with state 0_0. This approach leads to the following result.

Theorem 11.7. *There is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(n^3 3^k i_\times(n))$ time.*

Proof. We will show how to compute the table A_x for each type of node x in a nice tree decomposition T. Recall that an entry $A_x(c, \kappa)$ counts the number of partial solution of Dominating Set of size exactly κ in G_x satisfying the requirements defined by the states in the colouring c.

Leaf node: Let x be a leaf node in T with $X_x = \{v\}$. We compute A_x in the following way:

$$A_x(\{1\}, \kappa) = \begin{cases} 1 & \text{if } \kappa = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$A_x(\{0_0\}, \kappa) = \begin{cases} 1 & \text{if } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$A_x(\{0_?\}, \kappa) = \begin{cases} 1 & \text{if } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

Notice that this is correct since there is exactly one partial solution of size one that contains v, namely $\{v\}$, and exactly one partial solution of size zero that does not contain v, namely $\emptyset$.

Introduce node: Let x be an introduce node in T with child node y that introduces the vertex v, and let $c \in \{1, 0_1, 0_0\}^{|X_y|}$. We compute A_x in the following way:

$$\begin{aligned} A_x(c \times \{1\}, \kappa) &= \begin{cases} 0 & \text{if } v \text{ has a neighbour with state } 0_0 \text{ in } c \\ 0 & \text{if } \kappa = 0 \\ A_y(c, \kappa - 1) & \text{otherwise} \end{cases} \\ A_x(c \times \{0_0\}, \kappa) &= \begin{cases} 0 & \text{if } v \text{ has a neighbour with state 1 in } c \\ A_y(c, \kappa) & \text{otherwise} \end{cases} \\ A_x(c \times \{0_?\}, \kappa) &= A_y(c, \kappa) \end{aligned}$$

As the state $0_?$ is indifferent about domination, we can copy the appropriate value from A_y. With the other two states, we have to set $A_x(c, \kappa)$ to zero if a vertex with state 0_0 can be dominated by a vertex with state 1. Moreover, we have to update the size of the set if v gets state 1.

Forget node: Let x be a forget node in T with child node y that forgets the vertex v. We compute A_x in the following way:

$$A_x(c, \kappa) = A_y(c \times \{1\}, \kappa) + A_y(c \times \{0_?\}, \kappa) - A_y(c \times \{0_0\}, \kappa)$$

The number of partial solutions of size κ in G_x satisfying the requirements defined by c equals the number of partial solutions of size κ that contain v plus the number of partial solutions of size κ that do not contain v but where v is dominated. This last number can be computed by subtracting the number of such solutions in which v is not dominated (state 0_0) from the total number of partial solutions in which v may be dominated or not (state $0_?$). This shows the correctness of the above formula.

The computation in the forget node is a simple illustration of the principle of inclusion/exclusion and the related Möbius transform; see for example [33].

Join node: Let x be a join node in T and let l and r be its child nodes. Recall that $X_x = X_l = X_r$.

If we are using the set of states $\{1, 0_0, 0_?\}$, then we do not have the consider colourings with matching states in order to compute the join. Namely, we can compute A_x using the following formula:

$$A_x(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + \#_1(c)} A_l(c, \kappa_l) \cdot A_r(c, \kappa_r)$$

The fact that this formula does not need to consider multiple matching colourings per colouring c (see Proposition 11.3) is the main reason why the algorithm of this theorem is faster than previous results.

To see that the formula is correct, recall that any partial solution of Dominating Set on G_x counted in the table A_x can be constructed from combining partial solutions G_l and G_r that are counted in A_l and A_r, respectively. Because an entry in A_x where a vertex v that has state 1 in a colouring of X_x counts partial solutions with v in the vertex set of the partial solution, this entry must count combinations of partial

$\times$	1	0_1	0_0
1	1		
0_1		0_1	0_1
0_0		0_1	0_0

$\times$	1	0_1	$0_?$
1	1		
0_1			0_1
$0_?$		0_1	$0_?$

$\times$	1	$0_?$	0_0
1	1		
$0_?$		$0_?$	
0_0			0_0

Figure 11.1. Join tables for the DOMINATING SET problem. From left to right they correspond to Proposition 11.3, the algorithm from [2, 3], and Theorem 11.7.

solutions in A_l and A_r where this vertex is also in the vertex set of these partial solutions and thus also has state 1. Similarly, if a vertex v has state 0_0, we count partial solutions in which v is undominated; hence v must be undominated in both partial solutions we combine and also have state 0_0. And, if a vertex v has state $0_?$, we count partial solutions in which v is not in the vertex set of the partial solution and we are indifferent about domination; hence, we can get all combinations of partial solutions from G_l and G_r if we also are indifferent about domination in A_l and A_r which is represented by the state $0_?$. All in all, if we fix the sizes of the solutions from G_l and G_r that we use, then we only need to multiply the number of solutions from A_r and A_l of this size which have the same colouring on X_x. The formula is correct as it combines all possible combinations by summing over all possible sizes of solutions on G_l and G_r that lead to a solution on G_x of size κ. Notice that the term $\#_1(c)$ under the summation sign corrects the double counting of the vertices with state 1 in c.

After the execution of this algorithm, the number of dominating sets of G of size κ can be found in the table entry $A_z(\emptyset, \kappa)$, where z is the root of T.

For the running time, we observe that in a leaf, introduce, or forget node x, the time required to compute A_x is linear in the size of the table A_x. The computations involve n-bit numbers because there can be up to 2^n dominating sets in G. Since $c \in \{1, 0_0, 0_?\}^{|X_x|}$ and $0 \leq \kappa \leq n$, we can compute each table A_x in $\mathcal{O}(n3^k i_+(n))$ time. In a join node x, we have to perform $\mathcal{O}(n)$ multiplications to compute an entry of A_x. This gives a total of $\mathcal{O}(n^2 3^k i_\times(n))$ time per join node. As the nice tree decomposition has $\mathcal{O}(n)$ nodes, the total running time is $\mathcal{O}(n^3 3^k i_\times(n))$. □

The algorithm of Theorem 11.7 is exponentially faster in the treewidth k compared to the previous fastest algorithm of Alber et al. [2, 3]. Also, no exponentially faster algorithm exists unless the Strong Exponential-Time Hypothesis fails [227] (see Section 3.2). The exponential speed-up comes from the fact that we use a different set of states to represent the characteristics of partial solutions; a set of states that allows us to perform the computations in a join node much faster. We note that although the algorithm of Theorem 11.7 uses underlying ideas of the covering product, no transformations associated with such an algorithm are used directly.

To represent the characteristics of the partial solutions of the DOMINATING SET problem, we can use any of the following three sets of states: $\{1, 0_1, 0_0\}$, $\{1, 0_1, 0_?\}$, $\{1, 0_0, 0_?\}$. Depending on which set we choose, the number of combinations that we need to inspect in a join node differ. We give an overview of this in Figure 11.1: each table represents a join using a different set of states, and each state in an entry of such a table represents a combination of the states in the left and right child nodes

that need to be inspected the create this new state. The number of non-empty entries now shows how many combinations have to be considered per vertex in a bag of a join node. Therefore, one can easily see that a table in a join node can be computed in $\mathcal{O}^*(5^k)$, $\mathcal{O}^*(4^k)$, and $\mathcal{O}^*(3^k)$ time, respectively, depending on the set of states used. These tables correspond to the algorithm of Proposition 11.3, the algorithm of Alber et al. [2, 3], and the algorithm of Theorem 11.7, respectively.

The way in which we obtain the third table in Figure 11.1 from the first one reminds us of Strassen's algorithm for matrix multiplication [289]: the speed-up in this algorithm comes from the fact that one multiplication can be omitted by using a series of extra additions and subtractions. Here, we do something similar by adding up all entries with a 0_1-state or 0_0-state together in the $0_?$-state and computing the whole block of four combinations at once. We then reconstruct the values we need by subtracting the combinations with two 0_0-states.

The exponential speed-up obtained by the algorithm of Theorem 11.7 comes at the cost of extra polynomial factors in the running time. This is n^2 times the factor due to the fact that we work with n-bit numbers. Since we compute the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, instead of computing a minimum dominating set, some extra polynomial factors in n seem unavoidable. However, the ideas of Theorem 11.7 can also be used to count only *minimum* dominating sets. Using that Dominating Set has the de Fluiter property for treewidth (see Section 11.2), this leads to the following result, where the factor n^2 is replaced by the much smaller factor k^2.

Corollary 11.8. *There is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of minimum dominating sets in G in $\mathcal{O}(nk^2 3^k i_\times(n))$ time.*

Proof. We notice that the representation of the different characteristics of partial solutions used in Theorem 11.7 has the linear de Fluiter property when used to count the number of minimum dominating sets. More explicitly, when counting the number of minimum dominating sets, we need to store only the number of partial solutions of each different characteristic that are at most $k+1$ larger in size than the smallest partial solution with a non-zero entry. This holds, as larger partial solutions can never lead to a minimum dominating set since taking any set corresponding to this smallest non-zero entry and adding all vertices in X_x leads to a smaller partial solution that dominates at least the same vertices.

In this way, we can modify the algorithm of Theorem 11.7 such that, in each node $x \in T$, we store a number ξ_x representing the size of the smallest partial solution and a table A_x with the number of partial solutions $A_x(c, \kappa)$ with $\xi_x \leq \kappa \leq \xi_x + k + 1$.

In a leaf node x, we simply set $\xi_x = 0$. In an introduce or forget node x with child node y, we first compute the entries $A_x(c, \kappa)$ for $\xi_y \leq \kappa \leq \xi_y + k + 1$ and then set ξ_x to the value of κ corresponding to the smallest non-zero entry of A_x. While computing A_x, the algorithm uses $A_y(c, \kappa) = 0$ for any entry $A_y(c, \kappa)$ that falls outside the given range of κ. Finally, in a join node x with child nodes r and l, we do the same as in Theorem 11.7, but we compute only the entries with κ in the range $\xi_l + \xi_r - (k+1) \leq \kappa \leq \xi_l + \xi_r + (k+1)$. Furthermore, as all terms of the sum with κ_l or κ_r outside the range of A_l and A_r evaluate to zero, we now have to evaluate only $\mathcal{O}(k)$ terms of the sum. It is not hard to see that all relevant combinations of partial

solutions from the two child nodes l and r fall in this range of κ.

The modified algorithm computes $\mathcal{O}(n)$ tables of size $\mathcal{O}(k3^k)$, and the computation of each entry requires at most $\mathcal{O}(k)$ multiplications of n-bit numbers. Therefore, the running time is $\mathcal{O}(nk^2 3^k i_\times(n))$. □

A disadvantage of the direct use of the algorithm of Corollary 11.8 compared to Proposition 11.3 is that we cannot reconstruct a minimum dominating set in G by directly tracing back the computation that gave the size of a minimum domination set. However, as we show below, we can transform the tables computed by Theorem 11.7 and Corollary 11.8 that use the states $\{1, 0_0, 0_?\}$ in $\mathcal{O}^*(3^k)$ time into tables using any of the other sets of states. These transformations have two applications. First of all, they allow us to easily construct a minimum dominating set in G from the computation of Corollary 11.8 by transforming the computed tables into different tables as used in Proposition 11.3 and thereafter traverse the tree in a top-down fashion as we have discussed earlier. Secondly, they can be used to switch from using n-bit numbers to $\mathcal{O}(k)$-bit numbers, further improving the polynomial factors of the running time if we are interested only in solving the Dominating Set problem.

Lemma 11.9. *Let x be a node of a tree decomposition T and let A_x be a table with entries $A_x(c, \kappa)$ representing the number of partial solutions of* Dominating Set *of G_x of each size κ, for some range of κ, corresponding to each colouring c of the bag X_x with states from one of the following sets:*

$$\{1, 0_1, 0_0\} \qquad \{1, 0_1, 0_?\} \qquad \{1, 0_0, 0_?\} \qquad \textit{(see Table 11.1)}$$

The information represented in the table A_x does not depend on the choice of the set of states from the options given above. Moreover, there exist transformations between tables using representations with different sets of states using $\mathcal{O}(|X_x||A_x|)$ arithmetic operations.

Proof. We will transform A_x such that it represents the same information using a different set of states. The transformation will be given for fixed κ and can be repeated for each κ in the given range.

The transformations work in $|X_x|$ steps. In step i, we assume that the first $i-1$ coordinates of the colouring c in our table A_x use the initial set of states, and the last $|X_x| - i$ coordinates use the set of states to which we want to transform. Using this as an invariant, we change the set of states used for the i-th coordinate at step i.

Transforming from $\{1, 0_1, 0_0\}$ to $\{1, 0_0, 0_?\}$ can be done using the following formula in which $A_x(c, \kappa)$ represents our table for colouring c, c_1 is a subcolouring of size $i-1$ using states $\{1, 0_1, 0_0\}$, and c_2 is a subcolouring of size $|X_x| - i$ using states $\{1, 0_0, 0_?\}$.

$$A_x(c_1 \times \{0_?\} \times c_2, \kappa) = A_x(c_1 \times \{0_1\} \times c_2, \kappa) + A_x(c_1 \times \{0_0\} \times c_2, \kappa)$$

We keep entries with states 1 and 0_0 on the i-th vertex the same, and we remove entries with state 0_1 on the i-th vertex after computing the new value. In words, the above formula counts the number partial solutions that do not containing the i-th vertex v in their vertex sets by adding the number of partial solutions that do not contain v in their vertex sets and dominate it to the number of partial solutions that do not

contain v in the vertex sets and do not dominate it. This completes the description of the transformation.

To see that the new table contains the same information, we can apply the reverse transformation from the set of states $\{1, 0_0, 0_?\}$ to the set $\{1, 0_1, 0_0\}$ by using the same transformation with a different formula to introduce the new state:

$$A_x(c_1 \times \{0_1\} \times c_2, \kappa) = A_x(c_1 \times \{0_?\} \times c_2, \kappa) - A_x(c_1 \times \{0_0\} \times c_2, \kappa)$$

A similar argument applies here: the number of partial solutions that dominate but do not contain the i-th vertex v in their vertex sets equals the total number of partial solutions that do not contain v in their vertex sets minus the number of partial solutions in which v is undominated.

The other four transformations work similarly. Each transformation keeps the entries of one of the three states 0_1, 0_0, and $0_?$ intact, computes the entries for the new state by a coordinate-wise addition or subtraction of the other two states, and removes the entries using the third state from the table. To compute an entry with the new state, either the above two formula can be used if the new state is 0_1 or $0_?$, or the following formula can be used if the new state is 0_0:

$$A_x(c_1 \times \{0_0\} \times c_2, \kappa) = A_x(c_1 \times \{0_?\} \times c_2, \kappa) - A_x(c_1 \times \{0_1\} \times c_2, \kappa)$$

For the above transformations, we need $|X_x|$ additions or subtractions for each of the $|A_x|$ table entries. Hence, a transformation requires $\mathcal{O}(|X_x||A_x|)$ arithmetic operations. □

We are now ready to give our final improvement for Dominating Set.

Corollary 11.10. *There is an algorithm that, given a tree decomposition of a graph G of width k, computes the size of a minimum dominating set in G in $\mathcal{O}(nk^2 3^k)$ time.*

We could give a slightly shorter proof than the one given below. This proof would directly combine the algorithm of Proposition 11.3 with the ideas of Theorem 11.7 using the transformations from Lemma 11.9. However, combining our ideas with the computations in the introduce and forget nodes in the algorithm of Alber et al. [2, 3] gives a more elegant solution, which we prefer to present.

Proof. On leaf, introduce, and forget nodes, our algorithm is exactly the same as the algorithm of Alber et al. [2, 3], while on a join node it is similar to Corollary 11.8. We give the full algorithm for completeness.

For each node $x \in T$, we compute a table A_x with entries $A_x(c)$ containing the size of a smallest partial solution of Dominating Set that satisfies the requirements defined by the colouring c using the set of states $\{1, 0_1, 0_?\}$.

Leaf node: Let x be a leaf node in T. We compute A_x in the following way:

$$A_x(\{1\}) = 1 \qquad A_x(\{0_1\}) = \infty \qquad A_x(\{0_?\}) = 0$$

Introduce node: Let x be an introduce node in T with child node y introducing the

vertex v. We compute A_x in the following way:

$$\begin{aligned} A_x(c \times \{0_1\}) &= \begin{cases} A_y(c) & \text{if } v \text{ has a neighbour with state 1 in } c \\ \infty & \text{otherwise} \end{cases} \\ A_x(c \times \{0_?\}) &= A_y(c) \\ A_x(c \times \{1\}) &= 1 + A_y(\phi_{N(v):0_1 \to 0_?}(c)) \end{aligned}$$

Here, $\phi_{N(v):0_1 \to 0_?}(c)$ is the colouring c with every occurrence of the state 0_1 on a vertex in $N(v)$ replaced by the state $0_?$.

Forget node: Let x be a forget node in T with child node y forgetting the vertex v. We compute A_x in the following way:

$$A_x(c) = \min\{A_y(c \times \{1\}), A_y(c \times \{0_1\})\}$$

Correctness of the operations on a leaf, introduce, and forget node are easy to verify and follow from [2, 3].

Join node: Let x be a join node in T and let l and r be its child nodes. We first create two tables A'_l and A'_r. For $y \in \{l, r\}$, we let $\xi_y = \min\left\{A_y(c') \mid c' \in \{1, 0_1, 0_?\}^{|X_y|}\right\}$ and let A'_y have entries $A'_y(c, \kappa)$ for all $c \in \{1, 0_1, 0_?\}^{|X_y|}$ and κ with $\xi_y \leq \kappa \leq \xi_y + k + 1$:

$$A'_y(c, \kappa) = \begin{cases} 1 & \text{if } A_y(c) = \kappa \\ 0 & \text{otherwise} \end{cases}$$

After creating the tables A'_l and A'_r, we use Lemma 11.9 to transform the tables A'_l and A'_r such that they use colourings c with states from the set $\{1, 0_0, 0_?\}$. The initial tables A'_y do not contain the actual number of partial solutions; they contain a 1-entry if a corresponding partial solution exists. In this case, the tables obtained after the transformation count the number 1-entries in the tables before the transformation. In the table A'_x computed for the join node x, we now count the number of combinations of these 1-entries. This suffices since any smallest partial solution in G_x that is obtained by joining partial solutions from both child nodes must consist of minimum solutions in G_l and G_r.

We can compute A'_x by evaluating the formula for the join node in Theorem 11.7 for all κ with $\xi_l + \xi_r - (k+1) \leq \kappa \leq \xi_l + \xi_r + (k+1)$ using the tables A'_l and A'_r. If we do this in the same way as in Corollary 11.8, then we consider only the $\mathcal{O}(k)$ terms of the formula where κ_l and κ_r fall in the specified ranges for A_l and A_r, respectively, as other terms evaluate to zero. In this way, we obtain the table A'_x in which entries are marked by colourings with states from the set $\{1, 0_0, 0_?\}$. Finally, we use Lemma 11.9 to transform the table A'_x such that it again uses colourings with states from the set $\{1, 0_1, 0_?\}$. This final table gives the number of combinations of 1-entries in A_l and A_r that lead to partial solutions of each size that satisfy the associated colourings. Since we are interested only in the size of the smallest partial solution of Dominating Set of each characteristic, we can extract these values in the following way:

$$A_x(c) = \min\{\kappa \mid A'_x(c, \kappa) \geq 1;\ \xi_l + \xi_r - (k+1) \leq \kappa \leq \xi_l + \xi_r + (k+1)\}$$

For the running time, we first consider the computations in a join node. Here, each state transformation requires $\mathcal{O}(k^2 3^k)$ operations by Lemma 11.9 since the tables

have size $\mathcal{O}(k3^k)$. These operations involve $\mathcal{O}(k)$-bit numbers since the number of 1-entries in A_l and A_r is at most 3^{k+1}. Evaluating the formula that computes A'_x from the tables A'_l and A'_r costs $\mathcal{O}(k^2 3^k)$ multiplications. If we do not store a $\log(n)$-bit number for each entry in the tables A_x in any of the four kinds of nodes of T, but store only the smallest entry using a $\log(n)$-bit number and let A'_x contain the difference to this smallest entry, then all entries in any of the A'_x can also be represented using $\mathcal{O}(k)$-bit numbers. Since there are $\mathcal{O}(n)$ nodes in T, this gives a running time of $\mathcal{O}(nk^2 3^k)$. Note that the time required to multiply the $\mathcal{O}(k)$-bit numbers disappears in the computational model with $\mathcal{O}(k)$-bit word size that we use. □

Corollary 11.10 gives the currently fastest algorithm for Dominating Set on graphs given with a tree decomposition of width k. Essentially, what the algorithm does is fixing the 1-states and applying the covering product of Björklund et al. [28] on the 0_1-states and $0_?$-states, where the 0_1-states need to be covered by the same states from both child nodes. We chose to present our algorithm in a way that does not use the covering product directly, because reasoning with states allows us to generalise our results in Section 11.5.

We conclude by stating that we can directly obtain similar results for similar problems using exactly the same techniques:

Proposition 11.11. *For each of the following problems, there is an algorithm that solves them, given a tree decomposition of a graph G of width k, using the following running times:*

- Independent Dominating Set *in* $\mathcal{O}(n^3 3^k)$ *time.*
- Total Dominating Set *in* $\mathcal{O}(nk^2 4^k)$ *time.*
- Red-Blue Dominating Set *in* $\mathcal{O}(nk^2 2^k)$ *time.*

Proof (Sketch). Use the same techniques as in the rest of this section. We emphasise only the following details.

With Independent Dominating Set, the factor n^3 comes from the fact that this (minimisation) problem does not have the de Fluiter property for treewidth. However, we can still use $\mathcal{O}(k)$-bit numbers. This is because even though the expanded tables A'_l and A'_r have size at most $n3^k$, they still contain the value one only once for each of the 3^k characteristic before applying the state changes. Therefore, the total sum of the values in the table, and thus also the maximum values of an entry in these tables after the state transformations is 3^k; these can be represented by $\mathcal{O}(k)$-bit numbers.

With Total Dominating Set, the running time is linear in n while the extra polynomial factor is k^2. This is because this problem does have the linear de Fluiter property for treewidth.

With Red-Blue Dominating Set, an exponential factor of 2^k suffices as we can use two states for the red vertices (in the red-blue dominating set or not) and two different states on the blue vertices (dominated or not). □

×	0	1
0	0	1
1	1	

×	0	?
0	0	
?		?

Figure 11.2. Join tables for counting the number of perfect matchings. We used the symbol ? in the last table because the direct combination of two ?-states can lead to matching a vertex twice.

11.4. Counting the Number of Perfect Matchings

The next problem we consider is the problem of computing the number of perfect matchings in a graph. We give an $\mathcal{O}^*(2^k)$ time algorithm for this problem. This requires a slightly more complicated approach than the approach of the previous section. The main difference is that here every vertex needs to be matched *exactly* once, while previously we needed to dominate every vertex *at least* once. After introducing state transformations similar to Lemma 11.9, we will introduce some extra counting techniques to overcome this problem.

The obvious tree-decomposition-based dynamic programming algorithm uses the set of states $\{0, 1\}$, where 1 means this vertex is matched and 0 means that it is not. It then computes, for every node $x \in T$, a table A_x with entries $A_x(c)$ containing the number of matchings in G_x with the property that the only vertices that are not matched are exactly the vertices in the current bag X_x with state 0 in c. This algorithm will run in $\mathcal{O}^*(3^k)$ time; this running time can be derived from the join table in Figure 11.2. Similar to Lemma 11.9 in the previous section, we will prove that the table A_x contains exactly the same information independent of whether we use the set of states $\{0, 1\}$ or $\{0, ?\}$, where ? represents a vertex for which we do not specify whether it is matched or not. I.e., for a colouring c, we count the number of matchings in G_x, where all vertices in $V_x \setminus X_x$ and all vertices in X_x with state 1 in c are matched, all vertices in X_x with state 0 in c are unmatched, and all vertices in X_x with state ? can either be matched or not.

Lemma 11.12. *Let x be a node of a tree decomposition T and let A_x be a table with entries $A_x(c)$ representing the number of matchings in G_x matching all vertices in $V_x \setminus X_x$ and corresponding to each colouring c of the bag X_x with states from one of the following sets:*

$$\{1, 0\} \qquad \{1, ?\} \qquad \{0, ?\}$$

The information represented in the table A_x does not depend on the choice of the set of states from the options given above. Moreover, there exist transformations between tables using representations with different sets of states using $\mathcal{O}(|X_x||A_x|)$ arithmetic operations.

If one defines a vertex with state 1 or ? to be in a set S, and a vertex with state 0 not to be in S, then the state changes essentially are Möbius transforms and inversions, see [28]. The transformations in the proof below essentially are the fast evaluation algorithms from [28].

Proof. The transformations work almost identical to those in the proof of Lemma 11.9. In step $1 \leq i \leq |X_x|$, we assume that the first $i-1$ coordinates of the colouring c in our table use one set of states, and the last $|X_x| - i$ coordinates use the other set of states. Using this as an invariant, we change the set of states used for the i-th coordinate at step i.

Transforming from $\{0, 1\}$ to $\{0, ?\}$ or $\{1, ?\}$ can be done using the following formula. In this formula, $A_x(c)$ represents our table for colouring c, c_1 is a subcolouring of size $i - 1$ using states $\{0, 1\}$, and c_2 is a subcolouring of size $|X_x| - i$ using states $\{0, ?\}$:

$$A_x(c_1 \times \{?\} \times c_2) = A_x(c_1 \times \{0\} \times c_2) + A_x(c_1 \times \{1\} \times c_2)$$

In words, the number of matchings that may contain some vertex v equals the sum of the number of matchings that do and the number of matchings that do not contain v.

The following two similar formulas can be used for the other four transformations:

$$A_x(c_1 \times \{1\} \times c_2) = A_x(c_1 \times \{?\} \times c_2) - A_x(c_1 \times \{0\} \times c_2)$$

$$A_x(c_1 \times \{0\} \times c_2) = A_x(c_1 \times \{?\} \times c_2) - A_x(c_1 \times \{1\} \times c_2)$$

In these transformations, we need $|X_x|$ additions or subtractions for each of the $|A_x|$ table entries. Hence, a transformation requires $\mathcal{O}(|X_x||A_x|)$ arithmetic operations. □

Although we can transform our dynamic programming tables such that they use different sets of states, this does not directly help us in obtaining a faster algorithm for counting the number of perfect matchings. Namely, if we would combine two partial solutions in which a vertex v has the ?-state in a join node, then it is possible that v is matched twice in the combined solution: once in each child node. This would lead to incorrect answers, and this is why we put a ⁇ instead of a ? in the join table in Figure 11.2. We overcome this problem by using some additional counting tricks that can be found in the proof below.

Theorem 11.13. *There is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of perfect matchings in G in $\mathcal{O}(nk^2 2^k i_\times(k \log(n)))$ time.*

Proof. For each node $x \in T$, we compute a table A_x with entries $A_x(c)$ containing the number of matchings that match all vertices in $V_x \backslash X_x$ and that satisfy the requirements defined by the colouring c using states $\{1, 0\}$. We use the extra invariant that vertices with state 1 are matched only with vertices outside the bag, i.e., vertices that have already been forgotten by the algorithm. This prevents vertices being matched within the bag and greatly simplifies the presentation of the algorithm.

Leaf node: Let x be a leaf node in T. We compute A_x in the following way:

$$A_x(\{1\}) = 0 \qquad A_x(\{0\}) = 1$$

The only matching in the single vertex graph is the empty matching.

Introduce node: Let x be an introduce node in T with child node y introducing the vertex v. The invariant on vertices with state 1 makes the introduce operation trivial:

$$A_x(c \times \{1\}) = 0 \qquad A_x(c \times \{0\}) = A_y(c)$$

Forget node: Let x be a forget node in T with child node y forgetting the vertex v. If the vertex v is not matched already, then it must be matched to an available neighbour at this point:

$$A_x(c) = A_y(c \times \{1\}) + \sum_{u \in N(v), c(u)=1} A_y(\phi_{u:1\to 0}(c) \times \{0\})$$

Here, $c(u)$ is the state of u in c and $\phi_{u:1\to 0}(c)$ is the colouring c where the state of u is changed from 1 to 0. This formula computes the number of matchings corresponding to c, by adding the number of matchings in which v is matched already to the number of matchings of all possibly ways of matching v to one of its neighbours. We note that, because of our extra invariant, we have to consider only neighbours in the current bag X_x. Namely, if we would match v to an already forgotten vertex u, then we could have matched v to u in the node where u was forgotten.

Join node: Let x be a join node in T and let l and r be its child nodes.

The join is the most interesting operation. As discussed before, we cannot simply change the set of states to $\{0, ?\}$ and perform the join similar to Dominating Set as suggested by Table 11.2. We use the following method: we expand the tables and index them by the number of matched vertices in X_l or X_r, i.e., the number of vertices with state 1. Let $y \in \{l, r\}$, then we compute tables A'_l and A'_r as follows:

$$A'_y(c, i) = \begin{cases} A_y(c) & \text{if } \#_1(c) = i \\ 0 & \text{otherwise} \end{cases}$$

Next, we change the state representation in both tables A'_y to {0,?} using Lemma 11.12. These tables do not use state 1, but are still indexed by the number of 1-states used in the previous representation. Then, we join the tables by combining all possibilities that arise from i 1-states in the previous representation using states $\{0, 1\}$ (stored in the index i) using the following formula:

$$A'_x(c, i) = \sum_{i_l + i_r = i} A'_l(c, i_l) \cdot A'_r(c, i_r)$$

As a result, the entries $A'_x(c, i)$ give us the total number of ways to combine partial solutions from G_l and G_r such that the vertices with state 0 in c are unmatched, the vertices with state ? in c can be matched in zero, one, or both partial solutions used, and the total number of times the vertices with state ? are matched is i.

Next, we change the states in the table A'_x back to $\{0, 1\}$ using Lemma 11.12. It is important to note that the 1-state can now represent a vertex that is matched twice because the ?-state used before this second transformation represented vertices that could be matched twice as well. However, we can find those entries in which no vertex is matched twice by applying the following observation: the total number of 1-states in c should equal the sum of those in its child tables, and this sum is stored in the index i. Therefore, we can extract the number of perfect matchings for each colouring c using the following formula:

$$A_x(c) = A'_x(c, \#_1(c))$$

In this way, the algorithm correctly computes the tables A_x for a join node $x \in T$. This completes the description of the algorithm.

The computations in the join nodes again dominate the running time. In a join node, the transformations of the states in the tables cost $\mathcal{O}(k^2 2^k)$ arithmetic operations each, and the computations of A'_x from A'_l and A'_r also costs $\mathcal{O}(k^2 2^k)$ arithmetic operations. We will now show that these arithmetic operations can be implemented using $\mathcal{O}(k \log(n))$-bit numbers. For every vertex, we can say that the vertex is matched to another vertex at the time when it is forgotten in T, or when its matching neighbour is forgotten. When it is matched at the time that it is forgotten, then it is matched to one of its at most $k+1$ neighbours. This leads to at most $k+2$ choices per vertex. As a result, there are at most $\mathcal{O}(k^n)$ perfect matchings in G, and the described operations can be implemented using $\mathcal{O}(k \log(n))$-bit numbers.

Because a nice tree decomposition has $\mathcal{O}(n)$ nodes, the running time of the algorithm is $\mathcal{O}(nk^2 2^k i_\times(k \log(n)))$. □

The above theorem gives the currently fastest algorithm for counting the number of perfect matchings in graphs with a given tree decompositions of width k. The algorithm uses ideas from the fast subset convolution algorithm of Björklund et al. [28] to perform the computations in the join node.

11.5. $[\rho, \sigma]$-Domination Problems

We have shown how to solve two elementary problems in $\mathcal{O}^*(s^k)$ time on graphs of treewidth k, where s is the number of states per vertex used in representations of partial solutions. In this section, we generalise our result for Dominating Set to the $[\rho, \sigma]$-domination problems; for a definition of these problems, see Section 1.6. We show that we can solve all $[\rho, \sigma]$-domination problems with finite or cofinite ρ and σ in $\mathcal{O}^*(s^k)$ time. This includes the existence (decision), minimisation, maximisation, and counting variants of these problems.

For the $[\rho, \sigma]$-domination problems, one can also use colourings with states to represent the different characteristics of partial solutions. Let D be the vertex set of a partial solution of a $[\rho, \sigma]$-domination problem. One set of states that we use involves the states ρ_j and σ_j, where ρ_j and σ_j represent vertices not in D, or in D, that have j neighbours in D, respectively. For finite ρ, σ, we let $p = \max\{\rho\}$ and $q = \max\{\sigma\}$. In this case, we have the following set of states: $\{\rho_0, \rho_1, \ldots, \rho_p, \sigma_0, \sigma_1, \ldots, \sigma_q\}$. If ρ or σ are cofinite, we let $p = 1 + \max\{\mathbb{N} \setminus \rho\}$ and $q = 1 + \max\{\mathbb{N} \setminus \sigma\}$. In this case, we replace the last state in the given sets by $\rho_{\geq p}$ or $\rho_{\geq q}$, respectively. This state represents a vertex in the vertex set D of the partial solution of the $[\rho, \sigma]$-domination problem that has at least p neighbours in D, or a vertex not in D with at least q neighbours in D, respectively. Let $s = p + q + 2$ be the number of states involved.

Dynamic programming tables for the $[\rho, \sigma]$-domination problems can also be represented using different sets of states that contain the same information. In this section, we will use three different sets of states. These sets are defined as follows.

Definition 11.14. Let State Set I, II, and III be the following sets of states:

- State Set I: $\{\rho_0, \rho_1, \rho_2, \ldots, \rho_{p-1}, \rho_p/\rho_{\geq p}, \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{q-1}, \sigma_q/\sigma_{\geq q}\}$.
- StateSet II: $\{\rho_0, \rho_{\leq 1}, \rho_{\leq 2}, \ldots, \rho_{\leq p-1}, \rho_{\leq p}/\rho_{\mathbb{N}}, \sigma_0, \sigma_{\leq 1}, \sigma_{\leq 2}, \ldots, \sigma_{\leq q-1}, \sigma_{\leq q}/\sigma_{\mathbb{N}}\}$.
- State Set III: $\{\rho_0, \rho_1, \rho_2, \ldots \rho_{p-1}, \rho_p/\rho_{\geq p-1}, \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{q-1}, \sigma_q/\sigma_{\geq q-1}\}$.

The meaning of all the states is self-explanatory: $\rho_{condition}$ and $\sigma_{condition}$ consider the number of partial solutions of the $[\rho,\sigma]$-domination problem that do not contain (ρ-state) or do contain (σ-state) this vertex with a number of neighbours in the corresponding vertex sets satisfying the *condition*. The subscript $\mathbb{N}$ stands for no condition at all, i.e., $\rho_{\mathbb{N}} = \rho_{\geq 0}$: all possible number of neighbours in $\mathbb{N}$. We note that the notation $\rho_p/\rho_{\geq p}$ in Definition 11.14 is used to indicate that this set uses the state ρ_p if ρ is finite and $\rho_{\geq p}$ if ρ is cofinite.

Lemma 11.15. *Let x be a node of a tree decomposition T and let A_x be a table with entries $A_x(c,\kappa)$ representing the number of partial solutions of size κ to the $[\rho,\sigma]$-domination problem in G_x corresponding to each colouring c of the bag X_x with states from any of the three sets from Definition 11.14. The information represented in the table A_x does not depend on the choice of the set of states from the options given in Definition 11.14. Moreover, there exist transformations between tables using representations with different sets of states using $\mathcal{O}(s|X_x||A_x|)$ arithmetic operations.*

Proof. We apply transformations that work in $|X_x|$ steps and are similar to those in the proofs of Lemmas 11.9 and 11.12. In the i-th step, we replace the states at the i-th coordinate of c. We use the following formulas to create entries with a new state.

We will give only the formulas for the ρ-states. The formulas for the σ-states are identical, but with ρ replaced by σ and p replaced by q. We note that we slightly abuse notation below since we use that $\rho_{\leq 0} = \rho_0$.

To obtain states from State Set I not present in State Set II or III, we can use:

$$\begin{aligned}
A_x(c_1 \times \{\rho_j\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\leq j}\} \times c_2, \kappa) - A_x(c_1 \times \{\rho_{\leq j-1}\} \times c_2, \kappa) \\
A_x(c_1 \times \{\rho_{\geq p}\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\mathbb{N}}\} \times c_2, \kappa) - A_x(c_1 \times \{\rho_{\leq p-1}\} \times c_2, \kappa) \\
A_x(c_1 \times \{\rho_{\geq p}\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\geq p-1}\} \times c_2, \kappa) - A_x(c_1 \times \{\rho_{p-1}\} \times c_2, \kappa)
\end{aligned}$$

To obtain states from State Set II not present in State Set I or III, we can use:

$$\begin{aligned}
A_x(c_1 \times \{\rho_{\leq j}\} \times c_2, \kappa) &= \sum_{l=0}^{j} A_x(c_1 \times \{\rho_l\} \times c_2, \kappa) \\
A_x(c_1 \times \{\rho_{\mathbb{N}}\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\geq p}\} \times c_2, \kappa) + \sum_{l=0}^{p-1} A_x(c_1 \times \{\rho_l\} \times c_2, \kappa) \\
A_x(c_1 \times \{\rho_{\mathbb{N}}\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\geq p-1}\} \times c_2, \kappa) + \sum_{l=0}^{p-2} A_x(c_1 \times \{\rho_l\} \times c_2, \kappa)
\end{aligned}$$

To obtain states from State Set III not present in State Set I or II, we can use the same formulas used to obtain states from State Set I in combination with the following formulas:

$$\begin{aligned}
A_x(c_1 \times \{\rho_{\geq p-1}\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\geq p}\} \times c_2, \kappa) + A_x(c_1 \times \{\rho_{p-1}\} \times c_2, \kappa) \\
A_x(c_1 \times \{\rho_{\geq p-1}\} \times c_2, \kappa) &= A_x(c_1 \times \{\rho_{\mathbb{N}}\} \times c_2, \kappa) - A_x(c_1 \times \{\rho_{\leq p-2}\} \times c_2, \kappa)
\end{aligned}$$

As the transformations use $|X_x|$ steps in which each entry is computed by evaluating a sum of less than s terms, the transformations require $\mathcal{O}(|X_x||A_x|)$ arithmetic operations. □

Remark 11.1. We note that similar transformations can also be used to transform a table into a new table that uses different sets of states on different vertices in a bag X_x. For example, we can use State Set I on the first two vertices (assuming some ordering) and State Set III on the other $|X_x| - 2$ vertices. We will use a transformation of this type in the proof of Theorem 11.17.

To prove our main result for the $[\rho, \sigma]$-domination problems, we will also need more involved state transformations than those given above. We need to generalise the ideas of the proof of Theorem 11.13. In this proof, we expanded the tables A_l and A_r of the two child nodes l and r such that they contain entries $A_l(c, i)$ and $A_r(c, i)$, where i was an index indicating the number of 1-states used to create the ?-states in c. We will generalise this to the states used for the $[\rho, \sigma]$-domination problems.

Below, we often say that a colouring c of a bag X_x using State Set I from Definition 11.14 is *counted* in a colouring c' of X_x using State Set II. We let this be the case when, all partial solutions counted in the entry with colouring c in a table using State Set I are also counted in the entry with colouring c' in the same table when transformed such that it uses State Set II. I.e., when, for each vertex $v \in X_x$, $c(v)$ and $c'(v)$ are both σ-states or both ρ-states, and if $c(v) = \rho_i$ or $c(v) = \sigma_i$, then $c'(v) = \rho_{\leq j}$ or $c'(v) = \sigma_{\leq j}$ for some $j \geq i$.

Consider the case where ρ and σ are finite. We introduce an *index vector* $\vec{i} = (i_{\rho 1}, i_{\rho 2}, \ldots, i_{\rho p}, i_{\sigma 1}, i_{\sigma 2}, \ldots, i_{\sigma q})$ that is used in combination with states from State Set II from Definition 11.14. In this index vector, $i_{\rho j}$ and $i_{\sigma j}$ represent the sum over all vertices with state $\rho_{\leq j}$ and $\sigma_{\leq j}$ of the number of neighbours of the vertex in D, respectively. We say that a solution corresponding to a colouring c using State Set I from Definition 11.14 *satisfies* a combination of a colouring c' using State Set II and an index vector $\vec{i}$ if: c is counted in c', and for each $i_{\rho j}$ or $i_{\sigma j}$, the sum over all vertices with state $\rho_{\leq j}$ and $\sigma_{\leq j}$ in c' of the number of neighbours of the vertex in D equals $i_{\rho j}$ or $i_{\sigma j}$, respectively.

We clarify this with an example. Suppose that we have a bag of size three and a dynamic programming table indexed by colourings using the set of states $\{\rho_0, \rho_1, \rho_2, \sigma_0\}$ (State Set I) that we want to transform to one using the set states $\{\rho_0, \rho_{\leq 1}, \rho_{\leq 2}, \sigma_0\}$ (State Set II): thus $\vec{i} = (i_{\rho 1}, i_{\rho 2})$. Notice that a partial solution corresponding to the colouring $c = (\rho_0, \rho_1, \rho_2)$ will be counted in both $c'_1 = (\rho_0, \rho_{\leq 2}, \rho_{\leq 2})$ and $c'_2 = (\rho_{\leq 1}, \rho_{\leq 1}, \rho_{\leq 2})$. In this case, c satisfies the combination $(c'_1, \vec{i} = (0, 3))$ since the sum of the subscripts of the states in c of the vertices with state $\rho_{\leq 1}$ in c'_1 equals zero and this sum for the vertices with state $\rho_{\leq 2}$ in c'_1 equals three. Also, c satisfies no combination of c'_1 with an other index vector. Similarly, c satisfies the combination $(c'_2, \vec{i} = (1, 2))$ and no other combination involving c'_2.

In the case where ρ or σ are cofinite, the index vectors are one shorter: we do not count the sum of the number of neighbours in D of the vertices with state $\rho_{\mathbb{N}}$ and $\sigma_{\mathbb{N}}$.

What we will need is a table containing, for each possible combination of a colouring using State Set II with an index vector, the number of partial solutions that satisfy these. We can construct such a table using the following lemma.

Lemma 11.16. *Let x be a node of a tree decomposition T of width k. There exists an algorithm that, given a table A_x with entries $A_x(c, \kappa)$ containing the number of partial solutions of size κ to the $[\rho, \sigma]$-domination problem corresponding to the colouring c on*

the bag X_x using State Set I from Definition 11.14, computes in $\mathcal{O}(n(sk)^{s-1}s^{k+1}i_+(n))$ time a table A'_x with entries $A'_x(c,\kappa,\vec{i})$ containing the number partial solutions of size κ to the $[\rho,\sigma]$-domination problem satisfying the combination of a colouring using State Set II and the index vector $\vec{i}$.

Proof. We start with the following table A'_x using State Set I:

$$A'_x(c,\kappa,\vec{i}) = \begin{cases} A_x(c,\kappa) & \text{if } \vec{i} \text{ is the all-0 vector} \\ 0 & \text{otherwise} \end{cases}$$

Since there are no colourings with states $\rho_{\leq j}$ and $\sigma_{\leq j}$ yet, the sum of the number of neighbours in the vertex set D of the partial solutions of vertices with these states is zero.

Next, we change the states of the j-th coordinate at step j similar to Lemma 11.15, but now we also updates the index vector $\vec{i}$:

$$\begin{aligned} A'_x(c_1 \times \{\rho_{\leq j}\} \times c_2,\kappa,\vec{i}) &= \sum_{l=0}^{j} A'_x(c_1 \times \{\rho_l\} \times c_2,\kappa,\vec{i}_{i_{\rho j}\to(i_{\rho j}-l)}) \\ A'_x(c_1 \times \{\sigma_{\leq j}\} \times c_2,\kappa,\vec{i}) &= \sum_{l=0}^{j} A'_x(c_1 \times \{\sigma_l\} \times c_2,\kappa,\vec{i}_{i_{\sigma j}\to(i_{\sigma j}-l)}) \end{aligned}$$

Here, $\vec{i}_{i_{\rho j}\to(i_{\rho j}-l)}$ denotes the index vector $\vec{i}$ with the value of $i_{\rho j}$ set to $i_{\rho j}-l$.

If ρ or σ are cofinite, we simply use the formula in Lemma 11.15 for every fixed index vector $\vec{i}$ for the $\rho_{\mathbb{N}}$-states and $\sigma_{\mathbb{N}}$-states. We do so because we do not need to keep track of any index vectors for these states.

For the running time, note that each index $i_{\rho j}$, $i_{\sigma j}$ can have only values between zero and sk because there can be at most k vertices in X_x that each have at most s neighbours in D when considered for a state of the form $\rho_{\leq j}$ or $\sigma_{\leq j}$, as $j < p$ or $j < q$, respectively. The new table has $\mathcal{O}(n(sk)^{s-2}s^{k+1})$ entries since we have s^{k+1} colourings, $n+1$ sizes κ, and $s-2$ indices that range over sk values. Since the algorithm uses at most $k+1$ steps in which it computes a sum with less than s terms for each entry using n-bit numbers, this gives a running time of $\mathcal{O}(n(sk)^{s-1}s^{k+1}i_+(n))$. □

We are now ready to prove our main result of this section.

Theorem 11.17. *Let $\rho,\sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q and s be the values associated with the corresponding $[\rho,\sigma]$-domination problem. There is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of $[\rho,\sigma]$-dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(n^3(sk)^{2(s-2)}s^{k+1}i_\times(n))$ time.*

Notice that, for any given $[\rho,\sigma]$-domination problem, s is a fixed constant. Hence, Theorem 11.17 gives us $\mathcal{O}^*(s^k)$-time algorithms for these problems.

Proof. Before we give the computations involved for each type of node in a nice tree decomposition T, we slightly change the meaning of the subscript of the states $\rho_{condition}$ and $\sigma_{condition}$. In our algorithm, we let the subscripts of these states count only

the number of neighbours in the vertex sets D of the partial solution of the $[\rho, \sigma]$-domination problem that have already been forgotten by the algorithm. This prevents us from having to keep track of any adjacencies within a bag during a join operation. We will update these subscripts in the forget nodes. This modification is similar to the approach for counting perfect matchings in the proof of Theorem 11.13, where we matched vertices in a forget node to make sure that we did not have to deal with vertices that are matched within a bag when computing the table for a join node.

We will now give the computations for each type of node in a nice tree decomposition T. For each node $x \in T$, we will compute a table $A_x(c, \kappa)$ containing the number of partial solutions of size κ in G_x corresponding to the colouring c on X_x for all colourings c using State Set I from Definition 11.14 and all $0 \leq \kappa \leq n$. During this computation, we will transform to different sets of states using Lemmas 11.15 and 11.16 when necessary.

Leaf node: Let x be a leaf node in T.

Because the subscripts of the states count only neighbours in the vertex set of the partial solutions that have already been forgotten, we use only the states ρ_0 and σ_0 on a leaf. Furthermore, the number of σ-states must equal κ. As a result, we can compute A_x in the following way:

$$A_x(c, \kappa) = \begin{cases} 1 & \text{if } c = \{\rho_0\} \text{ and } \kappa = 0 \\ 1 & \text{if } c = \{\sigma_0\} \text{ and } \kappa = 1 \\ 0 & \text{otherwise} \end{cases}$$

Introduce node: Let x be an introduce node in T with child node y introducing the vertex v.

Again, the entries where v has the states ρ_j or σ_j, for $j \geq 1$, will be zero due to the definition of the (subscripts of) the states. Also, we must again keep track of the size κ. Let ς be the state of the introduced vertex. We compute A_x in the following way:

$$A_x(c \times \{\varsigma\}, \kappa) = \begin{cases} A_y(c, \kappa) & \text{if } \varsigma = \rho_0 \\ A_y(c, \kappa - 1) & \text{if } \varsigma = \sigma_0 \text{ and } \kappa \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Forget node: Let x be a forget node in T with child node y forgetting the vertex v.

The operations performed in the forget node are quite complicated. Here, we must update the states such that they are correct after forgetting the vertex v, and we must select those solutions that satisfy the constraints imposed on v by the specific $[\rho, \sigma]$-domination problem. We will do this in three steps: we compute intermediate tables A_1, A_2 in the first two steps and finally A_x in step three. Let $c(N(v))$ be the subcolouring of c restricted to vertices in $N(v)$.

Step 1: We update the states used on the vertex v. We do so to include the neighbours in D that the vertex v has inside the bag X_x in the states used to represent the different characteristics. Notice that after including these neighbours, the subscripts of the states on v represent the total number of neighbours that v has in D. The result will be the table A_1, which we compute using the following formulas where $\#_\sigma(c)$

stands for the number of σ-states in the colouring c:

$$A_1(c \times \{\rho_j\}, \kappa) = \begin{cases} A_y(c \times \{\rho_{j-\#_\sigma(c(N(v)))}\}, \kappa) & \text{if } j \geq \#_\sigma(c(N(v))) \\ 0 & \text{otherwise} \end{cases}$$

$$A_1(c \times \{\sigma_j\}, \kappa) = \begin{cases} A_y(c \times \{\sigma_{j-\#_\sigma(c(N(v)))}\}, \kappa) & \text{if } j \geq \#_\sigma(c(N(v))) \\ 0 & \text{otherwise} \end{cases}$$

If ρ or σ are cofinite, we also need the following formulas:

$$A_1(c \times \{\rho_{\geq p}\}, \kappa) = A_y(c \times \{\rho_{\geq p}\}, \kappa) + \sum_{i=p-\#_\sigma(c(N(v)))}^{p-1} A_y(c \times \{\rho_i\}, \kappa)$$

$$A_1(c \times \{\sigma_{\geq q}\}, \kappa) = A_y(c \times \{\sigma_{\geq q}\}, \kappa) + \sum_{i=q-\#_\sigma(c(N(v)))}^{q-1} A_y(c \times \{\sigma_i\}, \kappa)$$

Correctness of these formulas is easy to verify.

Step 2: We update the states representing the neighbours of v such that they are according to their definitions after forgetting v. All the required information to do this can again be read from the colouring c.

We apply Lemma 11.15 and change the state representation for the vertices in $N(v)$ to State Set III (Definition 11.14) obtaining the table $A'_1(c, \kappa)$; we do not change the representation of other vertices in the bag. That is, if ρ or σ are cofinite, we replace the last state $\rho_{\geq p}$ or $\sigma_{\geq q}$ by $\rho_{\geq p-1}$ or $\sigma_{\geq q-1}$, respectively, on vertices in $X_y \cap N(v)$. We can do so as discussed in Remark 11.1.

This state change allows us to extract the required values for the table A_2, as we will show next. We introduce the function ϕ that will send a colouring using State Set I to a colouring that uses State Set I on the vertices in $X_y \setminus N(v)$ and State Set III on the vertices in $X_y \cap N(v)$. This function updates the states used on $N(v)$ assuming that we would put v in the vertex set D of the partial solution. We define ϕ in the following way: it maps a colouring c to a new colouring with the same states on vertices in $X_y \setminus N(v)$ while it applies the following replacement rules on the states on vertices in $X_y \cap N(v)$: $\rho_1 \mapsto \rho_0$, $\rho_2 \mapsto \rho_1$, $\ldots$, $\rho_p \mapsto \rho_{p-1}$, $\rho_{\geq p} \mapsto \rho_{\geq p-1}$, $\sigma_1 \mapsto \sigma_0$, $\sigma_2 \mapsto \sigma_1$, $\ldots$, $\sigma_q \mapsto \sigma_{q-1}$, $\sigma_{\geq q} \mapsto \sigma_{\geq q-1}$. Thus, ϕ lowers the counters in the conditions that index the states by one for states representing vertices in $N(v)$. We note that $\phi(c)$ is defined only if $\rho_0, \sigma_0 \notin c$.

Using this function, we can easily update our states as required:

$$A_2(c \times \{\sigma_j\}, \kappa) = \begin{cases} A'_1(\phi(c) \times \{\sigma_j\}, \kappa) & \text{if } \rho_0, \sigma_0 \notin c(N(v)) \\ 0 & \text{otherwise} \end{cases}$$

$$A_2(c \times \{\rho_j\}, \kappa) = A'_1(c \times \{\rho_j\}, \kappa)$$

In words, for partial solutions on which the vertex v that we will forget has a σ-state, we update the states for vertices in $X_y \cap N(v)$ such that the vertex v is counted in the subscript of the states. Entries in A_2 are set to 0 if the states count no neighbours in D while v has a σ-state in c and thus a neighbour in D in this partial solution.

Notice that after updating the states using the above formula the colourings c in A_2 again uses State Set I from Definition 11.14.

Step 3: We select the solutions that satisfy the constraints of the specific $[\rho, \sigma]$-domination problem on v and forget v.

$$A_x(c, \kappa) = \left(\sum_{i \in \rho} A_2(c \times \{\rho_i\}, \kappa) \right) + \left(\sum_{i \in \sigma} A_2(c \times \{\sigma_i\}, \kappa) \right)$$

We slightly abuse our notation here when ρ or σ are cofinite. Following the discussion of the construction of the table A_x, we conclude that this correctly computes the required values.

Join node: Let x be a join node in T and let l and r be its child nodes. Computing the table A_x for the join node x is the most interesting operation.

First, we transform the tables A_l and A_r of the child nodes such that they use State Set II (Definition 11.14) and are indexed by index vectors using Lemma 11.16. As a result, we obtain tables A'_l and A'_r with entries $A'_l(c, \kappa, \vec{g})$ and $A'_r(c, \kappa, \vec{h})$. These entries count the number of partial solutions of size κ corresponding to the colouring c such that the sum of the number of neighbours in D of the set of vertices with each state equals the value that the index vectors $\vec{g}$ and $\vec{h}$ indicate. Here, D is again the vertex set of the partial solution involved. See the example above the statement of Lemma 11.16.

Then, we compute the table $A_x(c, \kappa, \vec{i})$ by combining identical states from A'_l and A'_r using the formula below. In this formula, we sum over all ways of obtaining a partial solution of size κ by combining the sizes in the tables of the child nodes and all ways of obtaining index vector $\vec{i}$ from $\vec{i} = \vec{g} + \vec{h}$.

$$A'_x(c, \kappa, \vec{i}) = \sum_{\kappa_l + \kappa_r = \kappa + \#_\sigma(c)} \left(\sum_{i_{\rho 1} = g_{\rho 1} + h_{\rho 1}} \cdots \sum_{i_{\sigma q} = g_{\sigma q} + h_{\sigma q}} A'_l(c, \kappa_l, \vec{g}) \cdot A'_r(c, \kappa_r, \vec{h}) \right)$$

We observe the following: a partial solution D in A'_x that is a combination of partial solutions from A'_l and A'_r is counted in an entry in $A'_x(c, \kappa, \vec{i})$ if and only if it satisfies the following three conditions.

1. The sum over all vertices with state $\rho_{\leq j}$ and $\sigma_{\leq j}$ of the number of neighbours of the vertex in D of this combined partial solution equals $i_{\rho j}$ or $i_{\sigma j}$, respectively.
2. The number of neighbours in D of each vertex with state $\rho_{\leq j}$ or $\sigma_{\leq j}$ of both partial solutions used to create this combined solution is at most j.
3. The total number of vertices in D in this joined solution is κ.

Let $\Sigma^l_\rho(c)$, $\Sigma^l_\sigma(c)$ be the weighted sums of the number of ρ_j-states and σ_j-states with $0 \leq j \leq l$ in c, respectively, defined by:

$$\Sigma^l_\rho(c) = \sum_{j=1}^{l} j \cdot \#_{\rho_j}(c) \qquad \Sigma^l_\sigma(c) = \sum_{j=1}^{l} j \cdot \#_{\sigma_j}(c)$$

We note that $\Sigma^1_\rho(c) = \#_{\rho_1}(c)$ and $\Sigma^1_\sigma(c) = \#_{\sigma_1}(c)$.

Now, using Lemma 11.15, we change the states used in the table A'_x back to State Set I. If ρ and σ are finite, we extract the values computed for the final table A_x in the following way:

$$A_x(c, \kappa) = A'_x \left(c,\ \kappa,\ (\Sigma^1_\rho(c), \Sigma^2_\rho(c), \ldots, \Sigma^p_\rho(c), \Sigma^1_\sigma(c), \Sigma^2_\sigma(c), \ldots, \Sigma^q_\sigma(c)) \right)$$

If ρ or σ are cofinite, we use the same formula but omit the components $\Sigma^p_\rho(c)$ or $\Sigma^q_\sigma(c)$ from the index vector of the extracted entries, respectively.

Below, we will prove that the entries in A_x are exactly the values that we want to compute. We first give some intuition. In essence, the proof is a generalisation of how we performed the join operation for #Perfect Matching in the proof of Theorem 11.13. State Set II has the role of the ?-states in the proof of Theorem 11.13. These states are used to count possible combinations of partial solutions from A_l and A_r. These combinations include incorrect combinations in the sense that a vertex can have more neighbours in D than it should have; this is analogous to #Perfect Matching, where combinations were incorrect if a vertex is matched twice. The values $\Sigma^l_\rho(c)$ and $\Sigma^l_\sigma(c)$ represent the total number of neighbours in D of the vertices with a ρ_j-states or σ_j-states with $0 \leq j \leq l$ in c, respectively. The above formula uses these $\Sigma^l_\rho(c)$ and $\Sigma^l_\sigma(c)$ to extract exactly those values from the table A'_x that correspond to correct combinations. That is, in this case, correct combinations for which the number of neighbours of a vertex in D is also correctly represented by the new states.

We will now prove that the computation of the entries in A_x gives the correct values. An entry in $A_x(c, \kappa)$ with $c \in \{\rho_0, \sigma_0\}^k$ is correct: these states are unaffected by the state changes and the index vector is not used. The values of these entries follow from combinations of partial solutions from both child nodes corresponding to the same states on the vertices.

Now consider an entry in $A_x(c, \kappa)$ with $c \in \{\rho_0, \rho_1, \sigma_0\}^k$. Each ρ_1-state comes from a $\rho_{\leq 1}$-state in $A'_x(c, \kappa, \vec{i})$ and is a combination of partial solutions from A_l and A_r with the following combinations of states on this vertex: (ρ_0, ρ_0), (ρ_0, ρ_1), (ρ_1, ρ_0), (ρ_1, ρ_1). Because we have changed states back to State Set I, each (ρ_0, ρ_0) combination is counted in the ρ_0-state on this vertex, and thus subtracted from the combinations used to form state ρ_1: the other three combinations remain counted in the ρ_1-state. Since we consider only those solutions with index vector $i_{\rho_1} = \Sigma^1_\rho(c)$, the total number of ρ_1-states used to form this joined solution equals $\Sigma^1_\rho(c) = \#_{\rho_1}(c)$. Therefore, no (ρ_1, ρ_1) combination could have been used, and each partial solution counted in $A(c, \kappa)$ has exactly one neighbour in D on each of the ρ_1-states, as required.

We can now inductively repeat this argument for the other states. For $c \in \{\rho_0, \rho_1, \rho_2, \sigma_0\}^k$, we know that the entries with only ρ_0-states and ρ_1-states are correct. Thus, when a ρ_2-state is formed from a $\rho_{\leq 2}$-state during the state transformation of Lemma 11.15, all nine possibilities of getting the state $\rho_{\leq 2}$ from the states ρ_0, ρ_1, and ρ_2 in the child bags are counted, and from this number all three combinations that should lead to a ρ_0 and ρ_1 in the join are subtracted. What remains are the combinations (ρ_0, ρ_2), (ρ_1, ρ_2), (ρ_2, ρ_2), (ρ_1, ρ_1), (ρ_1, ρ_2), (ρ_2, ρ_0). Because of the index vector of the specific the entry we extracted from A'_x, the total sum of the number of neighbours in D of these vertices equals Σ^2_ρ, and hence only the combinations (ρ_0, ρ_2), (ρ_1, ρ_1), and (ρ_2, ρ_0) could have been used. Any other combination would raise the component $i_{\rho 2}$ of $\vec{i}$ to a number larger than Σ^2_ρ.

If we repeat this argument for all states involved, we conclude that the above computation correctly computes A_x if ρ and σ are finite. If ρ or σ are cofinite, then the argument can also be used with one small difference. Namely, the index vectors are one component shorter and keep no index for the states $\rho_{\mathbb{N}}$ and $\sigma_{\mathbb{N}}$. That is, at the point in the algorithm where we introduce these index vectors and transform to State Set II using Lemma 11.16, we have no index corresponding to the sum of the number

of neighbours in the vertex set D of the partial solution of the vertices with states $\rho_{\mathbb{N}}$ and $\sigma_{\mathbb{N}}$. However, we do not need to select entries corresponding to having p or q neighbours in D for the states $\rho_{\geq p}$ and $\sigma_{\geq q}$ since these correspond to all possibilities of getting at least p or q neighbours in D. When we transform the states back to State Set I just before extracting the values for A_x from A'_x, entries that have the state $\rho_{\geq p}$ or $\sigma_{\geq q}$ after the transformation count all possible combinations of partial solutions except those counted in any of the other states. This is exactly what we need since all combinations with less than p (or q) neighbours are present in the other states.

After traversing the whole decomposition tree T, one can find the number of $[\rho, \sigma]$-dominating sets of size κ in the table computed for the root node z of T in $A_z(\emptyset, \kappa)$.

We conclude with an analysis of the running time. The most time-consuming computations are again those involved in computing the table A_x for a join node x. Here, we need $\mathcal{O}(n(sk)^{s-1}s^{k+1}i_+(n))$ time for the transformations of Lemma 11.16 that introduce the index vectors since $\max\{|X_x| \mid x \in T\} = k + 1$. However, this is still dominated by the time required to compute the table A'_x: this table contains at most $s^{k+1}n(sk)^{s-2}$ entries $A'_x(c, \kappa, \vec{i})$, each of which is computed by an $n(sk)^{s-2}$-term sum. This gives a total time of $\mathcal{O}(n^2(sk)^{2(s-2)}s^{k+1}i_\times(n))$ since we use n-bit numbers. Because the nice tree decomposition has $\mathcal{O}(n)$ nodes, we conclude that the algorithm runs in $\mathcal{O}(n^3(sk)^{2(s-2)}s^{k+1}i_\times(n))$ time in total. □

This proof generalises ideas from the fast subset convolution algorithm [28]. While convolutions use ranked Möbius transforms [28], we use transformations with multiple states and multiple ranks in our index vectors.

The polynomial factors in the proof of Theorem 11.17 can be improved in several ways. Some improvements we give are for $[\rho, \sigma]$-domination problems in general, and others apply only to specific problems. Similar to $s = p + q + 2$, we define the value r associated with a $[\rho, \sigma]$-domination problems as follows:

$$r = \begin{cases} \max\{p-1, q-1\} & \text{if } \rho \text{ and } \sigma \text{ are cofinite} \\ \max\{p, q-1\} & \text{if } \rho \text{ is finite and } \sigma \text{ is cofinite} \\ \max\{p-1, q\} & \text{if } \rho \text{ is confinite and } \sigma \text{ is finite} \\ \max\{p, q\} & \text{if } \rho \text{ and } \sigma \text{ are finite} \end{cases}$$

Corollary 11.18 (General $[\rho, \sigma]$-Domination Problems). *Let $\rho, \sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q, r, and s be the values associated with the corresponding $[\rho, \sigma]$-domination problem. There is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of $[\rho, \sigma]$-dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(n^3(rk)^{2r}s^{k+1}i_\times(n))$ time. Moreover, there is an algorithm that decides whether there exist a $[\rho, \sigma]$-dominating set of size κ, for each individual value of κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(n^3(rk)^{2r}s^{k+1}i_\times(log(n) + k\log(r)))$ time.*

Proof. We improve the polynomial factor $(sk)^{2(s-2)}$ to $(rk)^{2r}$ by making the following observation. We never combine partial solutions corresponding to a ρ-state in one child node with a partial solution corresponding to a σ-state on the same vertex in the other child node. Therefore, we can combine the components of the index vector related to the states ρ_j and σ_j for each fixed j in a single index. For example consider the ρ_1-states and σ_1-states. For these states, this means the following: if we index the number of vertices used to create a ρ_1-state and σ_1-state in i_1 and we have i_1

vertices on which a partial solution is formed by considering the combinations (ρ_0, ρ_1), (ρ_1, ρ_0), (ρ_1, ρ_1), (σ_0, σ_1), (σ_1, σ_0), or (σ_1, σ_1), then non of the combinations (ρ_1, ρ_1) and (σ_1, σ_1) could have been used. Since the new components of the index vector range between 0 and rk, this proves the first running time in the statement of the corollary.

The second running time follows from reasoning similar to that in Corollary 11.10. In this case, we can stop counting the number of partial solutions of each size and instead keep track of the existence of a partial solution of each size. The state transformations then count the number of 1-entries in the initial tables instead of the number of solutions. After computing the table for a join node, we have to reset all entries e of A_x to $\min\{1, e\}$. For these computations, we can use $\mathcal{O}(\log(n) + k\log(r))$-bit numbers. This is because of the following reasoning. For a fixed colouring c using State Set II, each of the at most r^{k+1} colourings using State Set I that can be counted in c occur with at most one index vector in the tables A'_l and A'_r. Note that these are r^{k+1} colourings, not s^{k+1} colourings, since ρ-states are never counted in a colouring c where the vertex has a σ-state and vice versa. Therefore, the result of the large summation over all index vectors $\vec{g}$ and $\vec{h}$ with $\vec{i} = \vec{g} + \vec{h}$ can be bounded from above by $(r^k)^2$. Since we sum over n possible combinations of sizes, the maximum is nr^{2k} allowing us to use $\mathcal{O}(\log(n) + k\log(r))$-bit numbers. □

As a result, we can, for example, compute the size of a minimum-cardinality perfect code in $\mathcal{O}(n^3k^2 3^k i_\times(\log(n)))$ time. Note that the time bound follows because the problem is fixed and we use a computational model with $\mathcal{O}(k)$-bit word size.

Corollary 11.19 ($[\rho, \sigma]$-Optimisation Problems with the de Fluiter Property). *Let $\rho, \sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q, r, and s be the values associated with the corresponding $[\rho, \sigma]$-domination problem. If the standard representation using State Set I of the minimisation (or maximisation) variant of this $[\rho, \sigma]$-domination problem has the de Fluiter property for treewidth with function f, then there is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of minimum (or maximum) $[\rho, \sigma]$-dominating sets in G in $\mathcal{O}(n(f(k))^2(rk)^{2r}s^{k+1}i_\times(n))$ time. Moreover, there is an algorithm that computes the minimum (or maximum) size of such a $[\rho, \sigma]$-dominating set in $\mathcal{O}(n(f(k))^2(rk)^{2r}s^{k+1}i_\times(log(n) + k\log(r)))$ time.*

Proof. The difference with the proof of Corollary 11.18 is that, similar to the proof of Corollary 11.8, we can keep track of the minimum or maximum size of a partial solution in each node of the tree decomposition and consider only other partial solutions whose size differs at most $f(k)$ of this minimum or maximum size. As a result, both factors n (the factor n due to the size of the tables, and the factor n due to the summation over the sizes of partial solutions) are replaced by a factor $f(k)$. □

As an application of Corollary 11.19, it follows for example that 2-Dominating Set can be solved in $\mathcal{O}(nk^6 3^k i_\times(\log(n)))$ time.

Corollary 11.20 ($[\rho, \sigma]$-Decision Problems). *Let $\rho, \sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q, r, and s be the values associated with the corresponding $[\rho, \sigma]$-domination problem. There is an algorithm that, given a tree decomposition of a graph G of width k, counts the number of $[\rho, \sigma]$-dominating sets in G in $\mathcal{O}(n(rk)^{2r}s^{k+1}i_\times(n))$ time.*

Moreover, there is an algorithm that decides whether there exists a $[\rho,\sigma]$-dominating set in $\mathcal{O}(n(rk)^{2r}s^{k+1}i_\times(log(n)+k\log(r)))$ time.

Proof. This result follows similarly as Corollary 11.19. In this case, we can omit the size parameter from our tables, and we can remove the sum over the sizes in the computation of entries of A'_x completely. □

As an application of Corollary 11.20, it follows for example that we can compute the number of strong stable sets (distance-2 independent sets) in $\mathcal{O}(nk^2 3^k i_\times(n))$ time.

11.6. Clique Covering, Clique Packing, and Clique Partitioning Problems

We conclude this chapter by considering another type of problems for which we give the currently fastest algorithms on tree decompositions: a series of clique covering, packing, and partitioning problems. To give a general type of result applying to many different problems, we define three classes of problems that we call *γ-clique covering*, *γ-clique packing*, and *γ-clique partitioning problems*. For all these problems, we obtain $\mathcal{O}^*(2^k)$ algorithms.

We start by defining the γ-clique problems. Their definitions resemble the definition of $[\rho,\sigma]$-domination problems.

Definition 11.21 (γ-Clique Covering, Packing and Partitioning). Let $\gamma \subseteq \mathbb{N} \setminus \{0\}$, let G be a graph, and let $\mathcal{C}$ be a collection of cliques from G such that the size of every clique in $\mathcal{C}$ is contained in γ. We define the following notions:

- $\mathcal{C}$ is a γ-clique cover of G if $\mathcal{C}$ covers the vertices of G, i.e, $\bigcup_{C\in\mathcal{C}} C = V$.
- $\mathcal{C}$ is a γ-clique packing of G if the cliques are disjoint, i.e, for any two $C_1, C_2 \in \mathcal{C}$: $C_1 \cap C_2 = \emptyset$.
- $\mathcal{C}$ is a γ-clique partitioning of G if it is both a γ-clique cover and a γ-clique packing.

The corresponding computational problems are defined in the following way. The *γ-clique covering* problems ask for the cardinality of the smallest γ-clique cover. The *γ-clique packing* problems ask for the cardinality of the largest γ-clique packing. The *γ-clique partitioning* problems ask whether a γ-clique partitioning exists. For these problems, we also consider their minimisation, maximisation, and counting variants. See Table 11.2 for some concrete example problems. We note that clique covering problems in the literature often ask to cover all the edges of a graph: here we cover only the vertices.

Throughout this section, we assume that γ is decidable in polynomial-time, that is, for every $j \in \mathbb{N}$ we can decide in time polynomial in j whether $j \in \gamma$. This allows us to precompute $\gamma \cap \{1, 2, \ldots, k+1\}$ in time polynomial in k, after which we can decide in constant time whether a clique of size l is allowed to be used in an associated covering, packing, or partitioning.

We start by giving algorithms for the γ-clique packing and partitioning problems.

γ	problem type	Standard problem description
$\{1, 2, \ldots\}$	partitioning, minimisation	Minimum clique partition
$\{2\}$	partitioning, counting	Count perfect matchings
$\{3\}$	covering	Minimum triangle cover of vertices
$\{3\}$	packing	Maximum triangle packing
$\{3\}$	partitioning	Partition into triangles
$\{p\}$	partitioning	Partition into p-cliques
$\{1, 3, 5, 7, \ldots\}$	covering	Minimum cover by odd-cliques

Table 11.2. γ-clique covering, packing and partitioning problems.

Theorem 11.22. *Let $\gamma \subseteq \mathbb{N} \setminus \{0\}$ be polynomial-time decidable. There is an algorithm that, given a tree decomposition of a graph G of width k, computes the number of γ-clique packings or γ-clique partitionings of G using κ, $0 \leq \kappa \leq n$, cliques in $\mathcal{O}(n^3 k^2 2^k i_\times(nk + n\log(n)))$ time.*

Proof. Before we start dynamic programming on the tree decomposition T, we first compute the set $\gamma \cap \{1, 2, \ldots, k+1\}$.

We use states 0 and 1 for the colourings c, where 1 means that a vertex is already in a clique in the partial solution, and 0 means that the vertex is not in a clique in the partial solution. For each node $x \in T$, we compute a table A_x with entries $A_x(c, \kappa)$ containing the number of γ-clique packings or partitionings of G_x consisting of exactly κ cliques that satisfy the requirements defined by the colouring $c \in \{1, 0\}^{|X_x|}$, for all $0 \leq \kappa \leq n$.

The algorithm uses the well-known property of tree decompositions that for every clique C in the graph G, there exists a node $x \in T$ such that C is contained in the bag X_x (a nice proof of this property can be found in [47]). As every vertex in G is forgotten in exactly one forget node in T, we can implicitly assign a unique forget node x_C to every clique C, namely the first forget node that forgets a vertex from C. In this forget node x_C, we will update the dynamic programming tables such that they take the choice of whether to pick C in a solution into account.

Leaf node: Let x be a leaf node in T. We compute A_x in the following way:

$$A_x(\{0\}, \kappa) = \begin{cases} 1 & \text{if } \kappa = 0 \\ 0 & \text{otherwise} \end{cases} \qquad A_x(\{1\}, \kappa) = 0$$

Since we decide to take cliques in a partial solution only in the forget nodes, the only partial solution we count in A_x is the empty solution.

Introduce node: Let x be an introduce node in T with child node y introducing the vertex v. Deciding whether to take a clique in a solution in the corresponding forget nodes makes the introduce operation trivial since the introduced vertex must have state 0:

$$A_x(c \times \{1\}, \kappa) = 0 \qquad A_x(c \times \{0\}, \kappa) = A_y(c, \kappa)$$

Join node: In contrast to previous algorithms, we will first present the computations in the join nodes. We do so because we will use this operation as a subroutine in the forget nodes.

×	0	1
0	0	1
1	1	

×	0	1
0	0	1
1	1	1

Figure 11.3. Join tables for γ-clique problems: the left table corresponds to partitioning and packing problems and the right table corresponds to covering problems.

Let x be a join node in T and let l and r be its child nodes. For the γ-clique partitioning and packing problems, the join is very similar to the join in the algorithm for counting the number of perfect matchings (Theorem 11.13). This can be seen from the corresponding join table; see Figure 11.3. The only difference is that we now also have the size parameter κ. Hence, for $y \in \{l, r\}$, we first create the tables A'_y with entries $A'_y(c, \kappa, i)$, where i indexes the number of 1-states in c. Then, we transform the set of states used for these tables A'_y from $\{1, 0\}$ to $\{0, ?\}$ using Lemma 11.12, and compute the table A'_x, now with the extra size parameter κ, using the following formula:

$$A'_x(c, \kappa, i) = \sum_{\kappa_l + \kappa_r = \kappa} \sum_{i = i_l + i_r} A'_l(c, \kappa_l, i_l) \cdot A'_r(c, \kappa_r, i_r)$$

Finally, the states in A'_x are transformed back to the set $\{0, 1\}$, after which the entries of A_x can be extracted that correspond to the correct number of 1-states in c. Because the approach described above is a simple extension of the join operation in the proof of Theorem 11.13 which was also used in the proof of Theorem 11.17, we omit further details.

Forget node: Let x be a forget node in T with child node y forgetting the vertex v. Here, we first update the table A_y such that it takes into account the choice of taking any clique in X_y that contains y in a solution or not.

Let M be a table with all the (non-empty) cliques C in X_y that contain the vertex v and such that $|C| \in \gamma$, i.e., M contains all the cliques that we need to consider before forgetting the vertex v. We notice that the operation of updating A_y such that it takes into account all possible ways of choosing the cliques in M is identical to letting the new A_y be the result of the join operation on A_y and the following table A_M:

$$A_M(c \times \{1\}, \kappa) = \begin{cases} 1 & \text{if all the 1-states in } c \text{ form a clique with } v \text{ in } M \text{ and } \kappa = 1 \\ 1 & \text{if } c \text{ is the colouring with only 0-states and } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$A_M(c \times \{0\}, \kappa) = 0$$

It is not hard to see that this updates A_y as required since $A_M(c, \kappa)$ is non-zero only when a clique in M is used with size $\kappa = 1$, or if no clique is used and $\kappa = 0$.

If we consider a partitioning problem, then $A_x(c, \kappa) = A_y(c \times \{1\}, \kappa)$ since v must be contained in a clique. If we consider a packing problem, then $A_x(c, \kappa) = A_y(c \times \{1\}, \kappa) + A_y(c \times \{0\}, \kappa)$ since v can but does not need to be in a clique. Clearly, this correctly computes A_y.

After computing A_z for the root node z of T, the number of γ-clique packings or partitionings of each size κ can be found in $A_z(\emptyset, \kappa)$.

For the running time, we first observe that there are at most $\mathcal{O}(n2^k)$ cliques in G since T has $\mathcal{O}(n)$ nodes that each contain at most $k+1$ vertices. Hence, there are at most $\mathcal{O}((n2^k)^n)$ ways to pick at most n cliques, and we can work with $\mathcal{O}(nk + n\log(n))$-bit numbers. As a join and a forget operation require $\mathcal{O}(n^2k^22^k)$ arithmetical operations, the running time is $\mathcal{O}(n^3k^22^k i_\times(nk + n\log(n)))$. □

For the γ-clique covering problems, the situation is different. We cannot count the number of γ-clique covers of all possible sizes, as the size of such a cover can be arbitrarily large. Even if we restrict ourselves to counting covers that contain each clique at most once, then we need numbers with an exponential number of bits. To see this, notice that the number of cliques in a graph of treewidth k is at most $\mathcal{O}^*(2^k)$ since there are at most $\mathcal{O}(2^k)$ different cliques in each bag. Hence, there are at most $2^{\mathcal{O}^*(2^k)}$ different clique covers, and these can be counted using only $\mathcal{O}^*(2^k)$-bit numbers. Therefore, we will restrict ourselves to counting covers of size at most n because minimum covers will never be larger than n.

A second difference is that, in a forget node, we now need to consider covering the forgotten vertex multiple times. This requires a slightly different approach.

Theorem 11.23. *Let $\gamma \subseteq \mathbb{N} \setminus \{0\}$ be polynomial-time decidable. There is an algorithm that, given a tree decomposition of a graph G of width k, computes the size and number of minimum γ-clique covers of G in $\mathcal{O}(n^3\log(k)2^k i_\times(nk + n\log(n)))$ time.*

Proof. The dynamic programming algorithm for counting the number of minimum γ-clique covers is similar to the algorithm of Theorem 11.22. It uses the same tables A_x for every $x \in T$ with entries $A_x(c, \kappa)$ for all $c \in \{0,1\}^{|X_x|}$ and $0 \leq \kappa \leq n$. And, the computations of these tables in a leaf or introduce node of T are the same.

Join node: Let x be a join node in T and let l and r be its child nodes. The join operation is different from the join operation in the algorithm of Theorem 11.22 as can be seen from Figure 11.3. Here, the join operation is similar to our method of handling the 0_1-states and 0_0-states for the Dominating Set problem in the algorithm of Theorem 11.7. We simply transform the states in A_l and A_r to $\{0, ?\}$ and compute A_x using these same states by summing over identical entries with different size parameters:

$$A_x(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa} A_l(c, \kappa_l) \cdot A_r(c, \kappa_r)$$

Then, we obtain the required result by transforming A_x back to using the set of states $\{1, 0\}$. We omit further details because this works analogously to Theorem 11.7. The only difference with before is that we use the value zero for any $A_l(c, \kappa_l)$ or $A_r(c, \kappa_r)$ with $\kappa_l, \kappa_r < 0$ or $\kappa_l, \kappa_r > n$ as these never contribute to minimum clique covers.

Forget node: Let x be a forget node in T with child node y forgetting the vertex v. In contrast to Theorem 11.22, we now have to consider covering v with multiple cliques. In a minimum cover, v can be covered at most k times because there are no vertices in X_x left to cover after using k cliques from X_x.

Let A_M be as in Theorem 11.22. What we need is a table that contains more than just all cliques that can be used to cover v: it needs to count all combinations of cliques that we can pick to cover v at most k times indexed by the number of cliques used. To create this new table, we let $A_M^0 = A_M$, and let A_M^j be the result of the join operation applied to the table A_M^{j-1} with itself. Then, the table A_M^j counts all ways of picking a series of 2^j sets $C_1, C_2, \ldots, C_{2^j}$, where each set is either the empty set or a clique from M. To see that this holds, compare the definition of the join operation for this problem to the result of executing these operations repeatedly. The algorithm computes $A_M^{\lceil \log(k) \rceil}$. Because we want to know the number of clique covers that we can choose, and not the number of series of $2^{\lceil \log(k) \rceil}$ sets $C_1, C_2, \ldots, C_{2^{\lceil \log(k) \rceil}}$, we have to compensate for the fact that most covers are counted more than once. Clearly, each cover consisting of κ cliques corresponds to a series in which $2^{\lceil \log(k) \rceil} - \kappa$ empty sets are picked: there are $\binom{2^{\lceil \log(k) \rceil}}{\kappa}$ possibilities of picking the empty sets and $\kappa!$ permutations of picking each of the κ cliques in any order. Hence, we divide each entry $A_M(c, \kappa)$ by $\kappa! \binom{2^{\lceil \log(k) \rceil}}{\kappa}$. Now, $A_M^{\lceil \log(k) \rceil}$ contains the numbers we need for a join with A_y.

After performing the join operation with A_y and $A_M^{\lceil \log(k) \rceil}$ obtaining a new table A_y, we select the entries of A_y that cover v: $A_x(c, \kappa) = A_y(c \times \{1\}, \kappa)$.

If we have computed A_z for the root node z of T, the size of the minimum γ-clique cover equals the smallest κ for which $A_z(\emptyset, \kappa)$ is non-zero, and this entry contains the number of such sets.

For the running time, we find that in order to compute $A_M^{\lceil \log(k) \rceil}$, we need $\mathcal{O}(\log(k))$ join operations. The running time then follows from the same analysis as in Theorem 11.22. □

Similar to previous results, we can improve the polynomial factors involved.

Corollary 11.24. *Let $\gamma \subseteq \mathbb{N} \setminus \{0\}$ be polynomial-time decidable. There are algorithms that, given a tree decomposition of a graph G of width k:*

1. *decide whether there exists a γ-clique partition of G in $\mathcal{O}(nk^2 2^k)$ time.*
2. *count the number of γ-clique packings in G or the number of γ-clique partitionings in G in $\mathcal{O}(nk^2 2^k i_\times(nk + n\log(n)))$ time.*
3. *compute the size of a maximum γ-clique packing in G, maximum γ-clique partitioning in G, or minimum γ-clique partitioning in G of a problem with the de Fluiter property for treewidth in $\mathcal{O}(nk^4 2^k)$ time.*
4. *compute the size of a minimum γ-clique cover in G of a problem with the de Fluiter property for treewidth in $\mathcal{O}(nk^2 \log(k) 2^k)$ time, or in in $\mathcal{O}(nk^2 2^k)$ time if $|\gamma|$ is a constant.*
5. *compute the number of maximum γ-clique packings in G, maximum γ-clique partitionings in G, or minimum γ-clique partitionings in G of a problem with the de Fluiter property for treewidth in $\mathcal{O}(nk^4 2^k i_\times(nk + n\log(n)))$ time.*
6. *compute the number of minimum γ-clique covers in G of a problem with the de Fluiter property for treewidth in $\mathcal{O}(nk^2 \log(k) 2^k i_\times(nk + n\log(n)))$ time, or in $\mathcal{O}(nk^2 2^k i_\times(nk + n\log(n)))$ time if $|\gamma|$ is a constant.*

Proof (Sketch). Similar to before. Either use the de Fluiter property to replace a factor n^2 by k^2, or omit the size parameter to completely remove this factor n^2 if

possible. Moreover, we can use $\mathcal{O}(k)$-bit numbers instead of $\mathcal{O}(nk + n\log(n))$-bit numbers if we are not counting the number of solutions. In this case, we omit the time required for the arithmetic operations because of the computational model that we use with $\mathcal{O}(k)$-bit word size. For the γ-clique cover problems where $|\gamma|$ is a constant, we note that we can use A_M^p for some constant p because, in a forget node, we need only a constant number of repetitions of the join operation on A_M^0 instead of $\log(k)$ repetitions. □

By this result, Partition Into Triangles can be solved in $\mathcal{O}(nk^2 2^k)$ time. For this problem, Lokshtanov et al. proved that the given exponential factor in the running time is optimal, unless the Strong Exponential-Time Hypothesis fails [227].

We note that in Corollary 11.24 the problem of deciding whether there exists a γ-clique cover is omitted. This is because this problem can easily be solved without dynamic programming on the tree decomposition by considering each vertex and testing whether it is contained in a clique whose size is a member of γ. This requires $\mathcal{O}(nk2^k)$ time in general, and polynomial time if $|\gamma|$ is a constant.

11.7. Concluding Remarks

In this chapter, we have given exponentially faster algorithms on tree decompositions for a large variety of problems. This exponential speed-up comes at the cost of additional polynomial factors in the running time. Further improvement of the base of the exponent in the running time of many of these algorithms seems to be very hard as this would disprove the Strong Exponential-Time Hypothesis.

Some of our algorithms can be used in practical situations and improve on previous algorithms in these situations. For example, consider the problems #Perfect Matching or Partition Into Triangles. Our running time of $nk^2 2^k$ is already faster than the previously fastest $n3^k$ algorithm for $k \geq 13$. For other problems such as Dominating Set, our algorithm represents a smaller improvement since $nk^2 3^k$ is faster than $n4^k$ only for $k \geq 22$; for these values the algorithm does not appear practical as $22^2 3^{22} \approx 10^{13}$ (this in contrast to $13^2 2^{12} \approx 10^6$). For problems involving even more states, our running times have only theoretical implications.

It would be interesting to see how much the polynomial factors can be improved. For example, it is likely that one can replace the factor n^2 in the running time of our algorithm that counts the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, by a factor $n\log(n)$ by evaluating the formula for a join node in the proof of Theorem 11.7 using fast Fourier transforms. This could also replace the factor k^2 by $k\log(k)$ in Corollary 11.10 and Theorem 11.13, and possibly improve the polynomial factors in the running time in Corollaries 11.18-11.20 and Corollary 11.24.

As a final remark, we mention that, very recently, we have obtained $\mathcal{O}^*(c^k)$ algorithms for various problems on tree decompositions, for some constant $c \geq 2$, for which previously only $\mathcal{O}^*(k^k)$ algorithms existed [88]. This includes problems like Hamiltonian Cycle, Feedback Vertex Set, Steiner Tree, and Connected Dominating Set. In this paper, the techniques presented in this chapter are used to obtain algorithms for which the base of the exponent of the running time is small. For many problems, the base c is so small that we can give proofs that no exponentially faster algorithms exist unless the Strong Exponential-Time Hypothesis fails [88].

12

Fast Dynamic Programming on Branch Decompositions

The second type of graph decomposition we consider in this thesis are branch decompositions. Like tree decompositions, branch decompositions have shown to be an effective tool for solving many combinatorial problems with both theoretical and practical applications. They are used extensively in designing algorithms for planar graphs and for graphs excluding a fixed minor. In particular, most of the recent results aimed at obtaining faster exact or parameterised algorithms on these graphs rely on branch decompositions [110, 113, 153, 154]. Practical algorithms using branch decompositions include those for ring routing problems [81] and tour merging for the Travelling Salesman Problem [82].

Recall the general two-step approach for graph-decomposition-based algorithms from Chapter 11. Concerning the first step in this approach, we note that finding the branchwidth of a graph is $\mathcal{NP}$-hard in general [286]. For fixed k, one can find a branch decomposition of width k in linear time, if such a decomposition exists, by combining the results from [37] and [48]. This is similar to tree decompositions, and the constant factors involved in this algorithm are very large. However, in contrast to tree decompositions for which the complexity on planar graphs is unknown, there exists a polynomial-time algorithm that computes a branch decomposition of minimal width of a planar graph [286]. For general graphs several useful heuristics exist [81, 82, 182, 249].

Concerning the second step of the general two-step approach, Dorn has shown how to use fast matrix multiplication to speed up dynamic programming algorithms on branch decompositions [110]. Among others, he gave an $\mathcal{O}^*(4^k)$ time algorithm for the Dominating Set problem. On planar graphs, faster algorithms exist using so-called

[†]This chapter is joint work with Hans L. Bodlaender, Erik Jan van Leeuwen and Martin Vatshelle. The chapter contains results of which a preliminary version has been presented at the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010) [50].

sphere-cut branch decompositions [114]. On these graphs, DOMINATING SET can be solved in $\mathcal{O}^*(3^{\frac{\omega}{2}k})$ time, where ω is the smallest constant such that two $n \times n$ matrices can be multiplied in $\mathcal{O}(n^\omega)$ time. Some of these results can be generalised to graphs that avoid a minor [113]. We obtain the same results without requiring planarity.

Our Results. In this chapter, we show that one can count the number of dominating sets of each given size in a graph in $\mathcal{O}^*(3^{\frac{\omega}{2}k})$ time. We also show that one can count the number of perfect matchings in a graph in $\mathcal{O}^*(2^{\frac{\omega}{2}k})$ time, and we show that the $[\rho, \sigma]$-domination problems with finite or cofinite ρ and σ can be solved in $\mathcal{O}^*(s^{\frac{\omega}{2}k})$, where s is the natural number of states required to represent partial solutions.

These results are based on combining the approach on tree decompositions of Chapter 11 with *fast matrix multiplication* to speed up dynamic programming as introduced by Dorn [110]. To make this work efficiently, we introduce the use of asymmetric vertex states. We note that matrix multiplication has been used for quite some time as a basic tool for solving combinatorial problems; see, for example, the results mentioned in Section 2.4 or in [190, 285]. One of the results of this chapter is that (generalisations of) fast subset convolution and fast matrix multiplication can be combined to obtain faster algorithms for many optimization problems.

Different from the results on tree decompositions in Chapter 11, the base of the exponent of the running time of the algorithms in this chapter does not equal the base of the exponent of their space requirement. That is, if the space requirement is $\mathcal{O}^*(s^k)$, our algorithms use $\mathcal{O}^*(s^{\frac{\omega}{2}k})$ time. This difference between the running times in both chapters is due to the fact that the structure of a branch decomposition is different from the structure of a tree decomposition. A tree decomposition can be transformed into a nice tree decomposition, such that every join node x with children l, r has $X_x = X_r = X_l$. But a branch decomposition does not have such a property: here we need to consider combining partial solutions from both tables of the child edges while forgetting and introducing new vertices at the same time.

Optimality, Polynomial Factors and Model of Computation. We note that, under the hypothesis that $\omega = 2$, which could be the true value of ω, the running times of our algorithms do attain the space bound. In this case, the base of the exponent of some of the running times are optimal under the Strong Exponential-Time Hypothesis. This follows from the results of Lokshtanov et al. [227] and the fact that the branchwidth of any graph is at most its treewidth plus one (Proposition 12.2).

Similar to Chapter 11, we will spend some effort to make the polynomial factors in the running times in this chapter small. Specifically, we want to obtain algorithms that are linear in n whenever possible. The improvement of the polynomial factors may lead to seemingly strange situations when the matrix multiplication constant is involved. To see this, notice that ω is defined as the smallest constant such that two $n \times n$ matrices can be multiplied in $\mathcal{O}(n^\omega)$ time. Consequently, any polylogarithmic factor in the running time of the corresponding matrix-multiplication algorithm disappears in an infinitesimal increase of ω. These polylogarithmic factors are additional polynomial factor in the running times of our algorithms on branch decompositions. In our analyses, we pay no extra attention to this, and we only explicitly give the polynomial factors involved that are not related to the time required to multiply matrices.

Finally, we note that we again use the notation $i_+(n)$ and $i_\times(n)$ for the time required to multiply and add n-bit numbers, respectively. Also, we analyse our algorithms in the *Random Access Machine* (RAM) model with $\mathcal{O}(k)$-bit word size [157], where addition and multiplication are unit-time operations ($i_+(k) = i_\times(k) = \mathcal{O}(1)$). We do so because of reasons similar to those given in Chapter 11.

Fast Matrix Multiplication. We defined ω to be the matrix multiplication constant, that is, $\mathcal{O}(n^\omega)$ is the time in which we can multiply two $n \times n$ matrices. Currently, $\omega < 2.376$ due to the algorithm by Coppersmith and Winograd [83].

For multiplying a $(n \times p)$ matrix A and $(p \times n)$ matrix B, we differentiate between $p \leq n$ and $p > n$. Under the assumption that $\omega = 2.376$, an $\mathcal{O}(n^{1.85}p^{0.54})$ time algorithm is known if $p \leq n$ [83]. Otherwise, the matrices can be multiplied in $\mathcal{O}(\frac{p}{n}n^\omega) = \mathcal{O}(pn^{\omega-1})$ time by matrix splitting: split the matrices A and B into $\frac{p}{n}$ many $n \times n$ matrices $A_1, A_2, \ldots A_{\frac{p}{n}}$ and $B_1, B_2, \ldots B_{\frac{p}{n}}$, multiply each of the $\frac{p}{n}$ pairs $A_i \times B_i$, and sum up the results.

Organisation of the Chapter. This chapter is organised as follows. We begin with an introduction to dynamic programming on branch decompositions in Section 12.1. In this section, we give some definitions and an example algorithm for Dominating Set. In the following sections, we give a series of faster dynamic programming algorithms on branch decompositions. We give a faster algorithm for Dominating Set in Section 12.2, a faster algorithm for #Perfect Matching in Section 12.3, and faster algorithms for the $[\rho, \sigma]$-domination problems in Section 12.4. Finally, we give some concluding remarks in Section 12.5.

12.1. Introduction to Branchwidth-Based Algorithms

Branch decompositions are closely related to tree decompositions. Similar to tree decompositions, one can solve many $\mathcal{NP}$-hard problems in polynomial time on graphs for which the branchwidth is bounded by a constant. If one is given a graph G with a branch decomposition T of G of width k, then the running time of such an algorithm is typically polynomial in the size of the graph G but exponential in the branchwidth k. Because of the close relation between branch decompositions and tree decompositions, a relation that is also reflected in Proposition 12.2, similar problems can be solved in this way on both types on graph decompositions. For example, consider the algorithms for Independent Set, Dominating Set, and many other problems that can be found in [110], or the algorithm for Steiner Tree that can be found in [184].

In this section, we introduce some important ideas of branchwidth-based algorithms. First, we give some definitions in Section 12.1.1. Thereafter, we present an example of a dynamic programming algorithm on branch decompositions in Section 12.1.2. This example algorithm will be the basis for all other algorithms presented in this chapter. In Section 12.1.3, we conclude by defining the de Fluiter property for branchwidth: a property similar to the de Fluiter property for treewidth defined in Section 11.2.

12.1.1. Definitions

Branch decompositions are related to tree decompositions and also originate from the series of papers on graph minors by Robertson and Seymour [268].

Definition 12.1 (Branch Decomposition). A *branch decomposition* of a graph G is a tree T in which each internal node has degree three and in which each leaf x of T has an assigned edge $e_x \in E$ such that this assignment is a bijection between the leaves of T and the edges E of G.

For a decomposition tree T, we often identify T with the set of nodes in T, and we write $E(T)$ for the edges of T.

If we would remove any edge e from a branch decomposition T of G, then this cuts T into two subtrees T_1 and T_2. In this way, the edge $e \in E(T)$ partitions the edges of G into two sets E_1, E_2 where E_i contains exactly those edges in the leaves of subtree T_i. The *middle set* X_e associated with the edge $e \in E(T)$ is defined to be the set of vertices $X_e \subseteq V$ that are both an endpoint of an edge in the edge partition E_1 and an endpoint of an edge in the edge partition E_2, where E_1 and E_2 are associated with e. That is, if $V_i = \bigcup E_i$, then $X_e = V_1 \cap V_2$.

The *width* $bw(T)$ of a branch decomposition T is the size of the largest middle set associated with the edges of T. The branchwidth $bw(G)$ of a graph G is the minimum width over all possible branch decompositions of G. In this chapter, we always assume that a branch decomposition of the appropriate width is given.

Observe that vertices v of degree one in G are not in any middle set of a branch decomposition T of G. Let u be the neighbour of such a vertex v. We include the vertex v in the middle set of the edge e of T incident to the leaf of T that contains $\{u, v\}$. This raises the branchwidth to $\max\{2, bw(G)\}$. Throughout this chapter, we ignore this technicality.

The treewidth $tw(G)$ and branchwidth $bw(G)$ of any graph G are related in the following way:

Proposition 12.2 ([268]). *For any graph G with branchwidth $bw(G) \geq 2$:*

$$bw(G) \leq tw(G) + 1 \leq \left\lfloor \frac{3}{2} bw(G) \right\rfloor$$

To perform dynamic programming on a branch decomposition T, we need T to be rooted. To create a root, we choose any edge $e \in E(T)$ and subdivide it creating edges e_1 and e_2 and a new node y. Next, we create another new node z, which will be our root, and add it together with the new edge $\{y, z\}$ to T. The middle sets associated with the edges created by the subdivision are set to X_e, i.e., $X_{e_1} = X_{e_2} = X_e$. Furthermore, the middle set of the new edge $\{y, z\}$ is the empty set: $X_{\{y,z\}} = \emptyset$.

We use the following terminology for the edges in a branch decomposition T giving similar names to edges as we would usually do to vertices. We call any edge of T that is incident to a leaf but not the root a *leaf edge*. Any other edge is called a *internal edge*. Let x be the lower endpoint of an internal edge e of T and let l, r be the other two edges incident to x. We call the edges l and r the *child edges* of e.

Definition 12.3 (Partitioning of Middle Sets). For a branch decomposition T, let $e \in E(T)$ be an edge not incident to a leaf with left child $l \in E(T)$ and right child $r \in E(T)$. We define the following partitioning of $X_e \cup X_l \cup X_r$:

1. The *intersection vertices*: $I = X_e \cap X_l \cap X_r$.
2. The *forget vertices*: $F = (X_l \cap X_r) \setminus I$.
3. The *vertices passed from the left*: $L = (X_e \cap X_l) \setminus I$.
4. The *vertices passed from the right*: $R = (X_e \cap X_r) \setminus I$.

Notice that this is a partitioning because any vertex in at least one of the sets X_e, X_l, X_r must be in at least two of them by the definition of a middle set.

Because each bag has size at most k, the partitioning satisfies the properties:

$$|I| + |L| + |R| \leq k \qquad |I| + |L| + |F| \leq k \qquad |I| + |R| + |F| \leq k$$

We associate with each edge $e \in E(T)$ of a branch decomposition T the induced subgraph $G_e = G[V_e]$ of G. A vertex $v \in V$ belongs to V_e in this definition if and only if there is a middle set f with $f = e$ or f below e in T with $v \in X_f$. That is, v is in V_e if and only if v is an endpoint of an edge associated with a leaf of T that is below e in T, i.e.:

$$G_e = G\Big[\bigcup\{X_f \mid f = e \text{ or } f \text{ is below } e \text{ in } T\}\Big]$$

For an overview of branch-decomposition-based techniques, see [184].

12.1.2. An Example Algorithm for Dominating Set

Dynamic programming algorithms on branch decompositions work similar to those on tree decompositions. The tree is traversed in a bottom-up manner while computing tables A_e with partial solutions on G_e for every *edge* e of T. Again, the table A_e contains partial solutions of each possible *characteristic*, where two partial solutions P_1 and P_2 have the same characteristic if any extension of P_1 to a solution on G also is an extension of P_2 to a solution on G. After computing a table for every edge $e \in E(T)$, we find a solution for the problem on G in the single entry of the table $A_{\{y,z\}}$, where z is the root of T and y is its only child node. Because the size of the tables is often (at least) exponential in k, such an algorithm typically runs in $\mathcal{O}(f(k)poly(n))$ time, for some function f that grows at least exponentially.

As an introduction, we will first give a simple dynamic programming algorithm for Dominating Set whose running time will be improved later. We note that a faster algorithm exists; see [110].

Proposition 12.4. *There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(mn^2 6^k i_\times(n))$ time.*

Proof. Let T be a branch decomposition of G rooted at a vertex z. For each edge $e \in E(T)$, we will compute a table A_e with entries $A_e(c, \kappa)$ for all $c \in \{1, 0_1, 0_0\}^{X_e}$ and all $0 \leq \kappa \leq n$. Here, c is a colouring with states 1, 0_1, and 0_0 that have the same meaning as in the tree-decomposition-based algorithms: see Table 11.1 in Section 11.1.2. In the

table A_e, an entry $A_e(c, \kappa)$ equals the number of partial solutions of DOMINATING SET of size κ in G_e that satisfy the requirements defined by the colouring c on the vertices in X_e. That is, the number of vertex sets $D \subseteq V_e$ of size κ that dominate all vertices in V_e except for those with state 0_0 in colouring c of X_e, and that contain all vertices in X_e with state 1 in c.

The described tables A_e are computed by traversing the decomposition tree T in a bottom-up manner. A branch decompositions has only two kinds of edges for which we need to compute such a table: leaf edges, and internal edges which have two child edges.

Leaf edges: Let e be an edge of T incident to a leaf of T that is not the root. Then, $G_e = G[X_e]$ is a two-vertex graph with $X_e = \{u, v\}$. Note that $\{u, v\} \in E$.

We compute A_e in the following way:

$$A_e(c, \kappa) = \begin{cases} 1 & \text{if } \kappa = 2 \text{ and } c = (1, 1) \\ 1 & \text{if } \kappa = 1 \text{ and either } c = (1, 0_1) \text{ or } c = (0_1, 1) \\ 1 & \text{if } \kappa = 0 \text{ and } c = (0_0, 0_0) \\ 0 & \text{otherwise} \end{cases}$$

The entries in this table are zero unless the colouring c represents one of the four possible partial solutions of DOMINATING SET on G_e and the size of this solution is κ. In these non-zero entries, the single partial solution represented by c is counted.

Internal edges: Let e be an internal edge of T with child edges l and r. Recall the definition of the sets I, L, R, F induced by X_e, X_l, and X_r (Definition 12.3).

Given a colouring c, let $c(I)$ denote the colouring of the vertices of I induced by c. We define $c(L)$, $c(R)$, and $c(F)$ in the same way. Given a colouring c_e of X_e, a colouring c_l of X_l, and a colouring c_r of X_r, we say that these colourings *match* if they correspond to a correct combination of two partial solutions with the colourings c_l and c_r on X_l and X_r which result is a partial solution that corresponds to the colouring c_e on X_e. For a vertex in each of the four partitions I, L, R, and F of $X_e \cup X_l \cup X_r$, this means something different:

- For any $v \in I$: either $c_e(v) = c_l(v) = c_r(v) \in \{1, 0_0\}$, or $c_e(v) = 0_1$ while $c_l(v), c_r(v) \in \{0_0, 0_1\}$ and not $c_l(v) = c_r(v) = 0_0$. (5 possibilities)
- For any $v \in F$: either $c_l(v) = c_r(v) = 1$, or $c_l(v), c_r(v) \in \{0_0, 0_1\}$ while not $c_l(v) = c_r(v) = 0_0$. (4 possibilities)
- For any $v \in L$: $c_e(v) = c_l(v) \in \{1, 0_1, 0_0\}$. (3 possibilities)
- For any $v \in R$: $c_e(v) = c_r(v) \in \{1, 0_1, 0_0\}$. (3 possibilities)

That is, for vertices in L or R, the properties defined by the colourings are copied from A_l and A_r to A_e. For vertices in I, the properties defined by the colouring c_e is a combination of the properties defined by c_l and c_r in the same way as it is for tree decompositions (as in Proposition 11.3). For vertices in F, the properties defined by the colourings are such that they form correct combinations in which the vertices may be forgotten, i.e., such a vertex is in the vertex set of both partial solutions, or it is not in the vertex set of both partial solutions while it is dominated.

Let $\kappa_{\#_1} = \#_1(c_r(I \cup F))$ be the number of vertices that are assigned state 1 on $I \cup F$ in any matching triple c_e, c_l, c_r. We can count the number of partial solutions on G_e

satisfying the requirements defined by each colouring c_e on X_e using the following formula:

$$A_e(c_e, \kappa) = \sum_{c_e, c_l, c_r \text{ match}} \sum_{\kappa_l + \kappa_r = \kappa + \kappa_{\#_1}} A_l(c_l, \kappa_l) \cdot A_r(c_r, \kappa_r)$$

Notice that this formula correctly counts all possible partial solutions on G_e per corresponding colouring c_e on X_e by counting all valid combinations of partial solutions on G_l corresponding to a colouring c_l on X_l and partial solutions G_r corresponding to a colouring c_r on X_r.

Let $\{y, z\}$ be the edge incident to the root z of T. From the definition of $A_{\{y,z\}}$, $A_{\{y,z\}}(\emptyset, \kappa)$ contains the number of dominating sets of size κ in $G_{\{y,z\}} = G$.

For the running time, we observe that we can compute A_e in $\mathcal{O}(n)$ time for all leaf edges e of T. For the internal edges, we have to compute the $\mathcal{O}(n3^k)$ values of A_e, each of which requires $\mathcal{O}(n)$ terms of the above sum per set of matchings states. Since each vertex in I has 5 possible matching states, each vertex in F has 4 possible matching states, and each vertex in L or R has 3 possible matching states, we compute each A_e in $\mathcal{O}(n^2 5^{|I|} 4^{|F|} 3^{|L|+|R|} i_\times(n))$ time.

Under the constraint that $|I| + |L| + |R|, |I| + |L| + |F|, |I| + |R| + |F| \leq k$, the running time is maximal if $|I| = 0$, $|L| = |R| = |F| = \frac{1}{2}k$. As T has $\mathcal{O}(m)$ edges and we work with n-bit numbers, this leads to a running time of $\mathcal{O}(mn^2 4^{\frac{1}{2}k} 3^k i_\times(n)) = \mathcal{O}(mn^2 6^k i_\times(n))$. □

The above algorithm gives the framework that we use in all of our dynamic programming algorithms on branch decompositions. In later algorithms, we will specify only how to compute the tables A_e for both kinds of edges.

We notice that the above algorithm computes the size of a minimum dominating set in G, but does not give the dominating set itself. To construct a minimum dominating set D, the branch decomposition T can be traversed in top-down order in the same way as we described for tree decompositions, tracing the entries in the tables that lead to a minimum solution in $A_{\{y,z\}}$.

12.1.3. De Fluiter Property for Treewidth/Branchwidth

We conclude this introduction to branchwidth-based algorithms with a discussion on a de Fluiter property for branchwidth. We will see below that such a property for branchwidth is identical to the de Fluiter property for treewidth.

One could define a de Fluiter property for branchwidth by replacing the words treewidth and tree decomposition in Definition 11.4 by branchwidth and branch decomposition. However, the result would be a property equivalent to the de Fluiter property for treewidth. This is not hard to see, namely, consider any edge e of a branch decomposition with middle set X_e. A representation of the different characteristics on X_e of partial solutions on G_e used on branch decompositions can also be used as a representation of the different characteristics on X_x of partial solutions on G_x on tree decompositions, if $X_e = X_x$ and $G_e = G_x$. Clearly, an extension of a partial solution on G_e with some characteristic is equivalent to an extension of the same partial solution on $G_x = G_e$, and hence the representations can be used on both decompositions. This equivalence of both de Fluiter properties follows directly.

As a result, we will use the de Fluiter property for treewidth in this chapter. Recall that this property on tree decompositions and branch decompositions is closely related to the finite integer index property; see Section 11.2. We will later also define a similar property for cliquewidth in Chapter 13; this property is different from the other two.

12.2. Minimum Dominating Set

We now start with giving faster algorithms on branch decompositions for a variety of problems. In this section, we improve the example algorithm of Proposition 12.4. This improvement will be presented in two steps. First, we use state changes similar to what we did in in Section 11.3. Thereafter, we will further improve the result by using fast matrix multiplication in the same way as proposed in [110]. As a result, we will obtain an $\mathcal{O}^*(3^{\frac{\omega}{2}k})$ algorithm for Dominating Set for graphs given with *any* branch decomposition of branchwidth k.

Similar to tree decompositions, it is more efficient to transform the problem to one using states 1, 0_0, and $0_?$ if we want to combine partial solutions from different dynamic programming tables. However, there is a large difference between dynamic programming on tree and on branch decompositions. On tree decompositions, we can deal with forget vertices separately, while this is not possible on branch decompositions. This makes the situation more complicated. On branch decompositions, vertices in F must be dealt with simultaneously with the computation of A_e from the two tables A_l and A_r for the child edges l and r of e. We will overcome this problem by using different sets of states simultaneously. The set of states used depends on whether a vertex is in L, R, I or F. Moreover, we do this asymmetrically as different states can be used on the same vertices in a different table A_e, A_l, A_r. This use of *asymmetrical vertex states* will later allow us to easily combine the use of state changes with fast matrix multiplication and obtain significant improvements in the running time.

We state this use of different states on different vertices formally. We note that this construction has already been used in the proof of Theorem 11.17.

Lemma 12.5. *Let e be an edge of a branch decomposition T with corresponding middle set $|X_e|$, and let A_e be a table with entries $A_e(c, \kappa)$ representing the number of partial solutions of* Dominating Set *in G_e of each size κ, for some range of κ, corresponding to all colourings of the middle set X_e with states such that for every individual vertex in X_e one of the following fixed sets of states is used:*

$$\{1, 0_1, 0_0\} \qquad \{1, 0_1, 0_?\} \qquad \{1, 0_0, 0_?\} \qquad \textit{(see Table 11.1 in Section 11.1.2)}$$

The information represented in the table A_e does not depend on the choice of the set of states from the options given above. Moreover, there exist transformations between tables using representations with different sets of states on each vertex using $\mathcal{O}(|X_x||A_x|)$ arithmetic operations.

Proof. Use the same $|X_e|$-step transformation as in the proof of Lemma 11.9 with the difference that we can choose a different formula to change the states at each coordinate of the colouring c of X_e. At coordinate i of the colouring c, we use the formula that corresponds to the set of states that we want to use on the corresponding vertex in X_e. See also Remark 11.1. □

We are now ready to give the first improvement of Proposition 12.4.

Proposition 12.6. *There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(mn^2 3^{\frac{3}{2}k} i_\times(n))$ time.*

Proof. The algorithm is similar to the algorithm of Proposition 12.4, only we employ a different method to compute A_e for an internal edge e of T.

Internal edges: Let e be an internal edge of T with child edges l and r.

We start by applying Lemma 12.5 to A_l and A_r and change the sets of states used for each individual vertex in the following way. We let A_l use the set of states $\{1, 0_?, 0_0\}$ on vertices in I and the set of states $\{1, 0_1, 0_0\}$ on vertices in L and F. We let A_r use the set of states $\{1, 0_?, 0_0\}$ on vertices in I, the set of states $\{1, 0_1, 0_0\}$ on vertices in R, and the set of states $\{1, 0_1, 0_?\}$ on vertices in F. Finally, we let A_e use the set of states $\{1, 0_?, 0_0\}$ on vertices in I and the set of states $\{1, 0_1, 0_0\}$ on vertices in L and R. Notice that different colourings use the same sets of states on the same vertices with the exception of the set of states used for vertices in F; here, A_l and A_r use different sets of states.

Now, three colourings c_e, c_l and c_r *match* if:

- For any $v \in I$: $c_e(v) = c_l(v) = c_r(v) \in \{1, 0_?, 0_0\}$. (3 possibilities)
- For any $v \in F$: either $c_l(v) = c_r(v) = 1$, or $c_l(v) = 0_0$ and $c_r(v) = 0_1$, or $c_l(v) = 0_1$ and $c_r(v) = 0_?$. (3 possibilities)
- For any $v \in L$: $c_e(v) = c_l(v) \in \{1, 0_1, 0_0\}$. (3 possibilities)
- For any $v \in R$: $c_e(v) = c_r(v) \in \{1, 0_1, 0_0\}$. (3 possibilities)

For the vertices on I, these matching combinations are the same as used on tree decompositions in Theorem 11.7, namely the combinations with states from the set $\{1, 0_?, 0_0\}$ where all states are the same. For the vertices on L and R, we do exactly the same as in the proof of Proposition 12.4.

For the vertices in F, a more complicated method has to be used. Here, we can use only combinations that make sure that these vertices will be dominated: combinations with vertices that are in vertex set of the partial solution, or combinations in which the vertices are not in this vertex set, but in which they will be dominated. Moreover, by using different states for A_l and A_r, every combination of partial solutions is counted exactly once. To see this, consider each of the three combinations on F used in Proposition 12.4 with the set of states $\{1, 0_1, 0_0\}$. The combination with $c_l(v) = 0_0$ and $c_r(v) = 0_1$ is counted using the same combination, while the other two combinations ($c_l(v) = 0_1$ and $c_r(v) = 0_0$ or $c_r(v) = 0_1$) are counted when combining 0_1 with $0_?$.

In this way, we can compute the entries in the table A_e using the following formula:

$$A_e(c_e, \kappa) = \sum_{c_e, c_l, c_r \text{ match}} \sum_{\kappa_l + \kappa_r = \kappa + \kappa_{\#_1}} A_l(c_l, \kappa_l) \cdot A_r(c_r, \kappa_r)$$

Here, $\kappa_{\#_1} = \#_1(c_r(I \cup F))$ again is the number of vertices that are assigned state 1 on $I \cup F$ in any matching triple c_e, c_l, c_r.

After having obtained A_e in this way, we can transform the set of states used back to $\{1, 0_1, 0_0\}$ using Lemma 12.5.

For the running time, we observe that the combination of the different sets of states that we are using allows us to evaluate the above formula in $\mathcal{O}(n^2 3^{|I|+|L|+|R|+|F|} i_\times(n))$ time. As each state transformation requires $\mathcal{O}(n3^k i_+(n))$ time, the improved algorithm has a running time of $\mathcal{O}(mn^2 3^{|I|+|L|+|R|+|F|} i_\times(n))$. Under the constraint that $|I| + |L| + |R|, |I| + |L| + |F|, |I| + |R| + |F| \leq k$, the running time is maximal if $|I| = 0$, $|L| = |R| = |F| = \frac{1}{2}k$. This gives a total running time of $\mathcal{O}(mn^2 3^{\frac{3}{2}k} i_\times(n))$. □

We will now give our faster algorithm for counting the number of dominating sets of each size κ, $0 \leq \kappa \leq n$ on branch decompositions. This algorithm uses fast matrix multiplication to speed up the algorithm of Proposition 12.6. This use of fast matrix multiplication in dynamic programming algorithms on branch decompositions was first proposed by Dorn in [110].

Theorem 12.7. *There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(mn^2 3^{\frac{\omega}{2}k} i_\times(n))$ time.*

Proof. Consider the algorithm of Proposition 12.6. We will make one modification to this algorithm. Namely, when computing the table A_e for an internal edge $e \in E(T)$, we will show how to evaluate the formula for $A_e(c, \kappa)$ for a number of colourings c simultaneously using fast matrix multiplication. We give the details below. Here, we assume that the states in the tables A_l and A_r are transformed such that the given formula for $A_e(c, \kappa)$ in Proposition 12.6 applies.

We do the following. First, we fix the two numbers κ and κ_l, and we fix a colouring of I. Note that this is well-defined because all three tables use the same set of states for colours on I. Second, we construct a $3^{|L|} \times 3^{|F|}$ matrix M_l where each row corresponds to a colouring of L and each column corresponds to a colouring of F where both colourings use the states used by the corresponding vertices in A_l. We let the entries of M_l be the values of $A_l(c_l, \kappa_l)$ for the c_l corresponding to the colourings of L and F of the given row and column of M_l, and corresponding to the fixed colouring on I and the fixed number κ_l. We also construct a similar $3^{|F|} \times 3^{|R|}$ matrix M_r with entries from A_r such that its rows correspond to different colourings of F and its columns correspond to different colourings of R where both colourings use the states used by the corresponding vertices in A_r. The entries of M_r are the values of $A_r(c_r, \kappa - \kappa_l - \kappa_{\#_1})$ where c_r corresponds to the colouring of R and F of the given row and column of M_r, and corresponding to the fixed colouring on I and the fixed numbers κ and κ_l. Here, the value of $\kappa_{\#_1} = \#_1(c_r(I \cup F))$ depends on the colouring c_r in the same way as in Proposition 12.6. Third, we permute the rows of M_r such that column i of M_l and row i of M_r correspond to matching colourings on F.

Now, we can evaluate the formula for A_e for all entries corresponding to the fixed colouring on I and the fixed values of κ and κ_l simultaneously by computing $M_e = M_l \cdot M_r$. Clearly, M_e is a $3^{|L|} \times 3^{|R|}$ matrix where each row corresponds to a colouring of L and each column corresponds to a colouring of R. If one works out the matrix product $M_l \cdot M_r$, one can see that each entry of M_e contains the sum of the terms of the formula for $A_e(c_e, \kappa)$ such that the colouring c_e corresponds to the given row and column of M_e and the given fixed colouring on I and such that $\kappa_l + \kappa_r = \kappa + \kappa_{\#_1}$ corresponding to the fixed κ and κ_l. That is, each entry in M_e equals the sum over all possible allowed matching combinations of the colouring on F for the fixed values

of κ and κ_l, where the κ_r involved are adjusted such that the number of 1-states used on F is taken into account.

In this way, we can compute the function A_e by repeating the above matrix-multiplication-based process for every colouring on I and every value of κ and κ_l in the range from 0 to n. As a result, we can compute the function A_e by a series of $n^2\, 3^{|I|}$ matrix multiplications.

The time required to compute A_e in this way depends on $|I|$, $|L|$, $|R|$ and $|F|$. Under the constraint that $|I| + |L| + |F|, |I| + |R| + |F|, |I| + |L| + |R| \leq k$ and using the matrix-multiplication algorithms for square and non-square matrices as described in the introduction of this chapter, the worst case arises when $|I| = 0$ and $|L| = |R| = |F| = \frac{k}{2}$. In this case, we compute each table A_e in $\mathcal{O}(n^2(3^{\frac{k}{2}})^{\omega} i_{\times}(n))$ time. This gives a total running time of $\mathcal{O}(mn^2 3^{\frac{\omega}{2}k} i_{\times}(n))$. □

Using the fact that Dominating Set has the de Fluiter property for treewidth, and using the same tricks as in Corollaries 11.8 and 11.10, we also obtain the following results.

Corollary 12.8. *There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of minimum dominating sets in G in $\mathcal{O}(mk^2 3^{\frac{\omega}{2}k} i_{\times}(n))$ time.*

Corollary 12.9. *There is an algorithm that, given a branch decomposition of a graph G of width k, computes the size of a minimum dominating set in G in $\mathcal{O}(mk^2 3^{\frac{\omega}{2}k} i_{\times}(k))$ time.*

12.3. Counting the Number of Perfect Matchings

The next problem we consider is the problem of counting the number of perfect matchings in a graph. To give a fast algorithm for this problem, we use both the ideas introduced in Theorem 11.13 to count the number of perfect matchings on graphs given with a tree decomposition and the idea of using fast matrix multiplication of Dorn [110] found in Theorem 12.7. We note that [110, 111] did not consider counting perfect matchings. The result will be an $\mathcal{O}^*(2^{\frac{\omega}{2}k})$ algorithm.

From the algorithm to count the number of dominating sets of each given size in a graph of bounded branchwidth, it is clear that vertices in I and F need special attention when developing a dynamic programming algorithm over branch decompositions. This is no different when we consider counting the number of perfect matchings. For the vertices in I, we will use state changes and an index similar to Theorem 11.13, but for the vertices in F we will require only that all these vertices are matched. In contrast to the approach on tree decompositions, we will not take into account the fact that we can pick edges in the matching at the point in the algorithm where we forget the first endpoint of the edge. We represent this choice directly in the tables A_e of the leaf edges e: this is possible because every edge of G is uniquely assigned to a leaf of T.

Our algorithm will again be based on state changes, where we will again use different sets of states on vertices with different roles in the computation.

Lemma 12.10. *Let e be an edge of a branch decomposition T with corresponding middle set $|X_e|$, and let A_e be a table with entries $A_e(c, \kappa)$ representing the number of matchings in H_e matching all vertices in $V_e \setminus X_e$ and corresponding to all colourings of the middle set X_e with states such that for every individual vertex in X_e one of the following fixed sets of states is used:*

$\{1, 0\}$ $\{1, ?\}$ $\{0, ?\}$ *(meaning of the states as in Lemma 11.12)*

The information represented in the table A_e does not depend on the choice of the set of states from the options given above. Moreover, there exist transformations between tables using representations with different sets of states on each vertex using $\mathcal{O}(|X_x||A_x|)$ arithmetic operations.

Proof. The proof is identical to that of Lemma 12.5 while using the formulas from the proof of Lemma 11.12. □

Theorem 12.11. *There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of perfect matchings in G in $\mathcal{O}(mk^2 2^{\frac{\omega}{2}k} i_\times(k \log(n)))$ time.*

Proof. Let T be a branch decomposition of G of branchwidth k rooted at a vertex z.

For each edge $e \in E(T)$, we will compute a table A_e with entries $A_e(c)$ for all $c \in \{1, 0\}^{X_e}$ where the states have the same meaning as in Theorem 11.13. In this table, an entry $A_e(c)$ equals the number of matchings in the graph H_e matching all vertices in $V_e \setminus X_e$ and satisfying the requirements defined by the colouring c on the vertices in X_e. These entries do not count matchings in G_e but in its subgraph H_e that has the same vertices as G_e but contains only the edges of G_e that are in the leaves below e in T.

Leaf edges: Let e be an edge of T incident to a leaf of T that is not the root. Now, $H_e = G_e = G[X_e]$ is a two vertex graph with $X_e = \{u, v\}$ and with an edge between u and v.

We compute A_e in the following way:

$$A_e(c) = \begin{cases} 1 & \text{if } c = (1, 1) \text{ or } c = (0, 0) \\ 0 & \text{otherwise} \end{cases}$$

The only non-zero entries are the empty matching and the matching consisting of the unique edge in H_e. This is clearly correct.

Internal edges: Let e be an internal edge of T with child edges l and r.

Similar to the proof of Theorem 11.13, we start by indexing the tables A_l and A_r by the number of 1-states used for later use. However, we now count only the number of 1-states used on vertices in I in the index. We compute indexed tables A'_l and A'_r with entries $A'_l(c_l, i_l)$ and $A'_r(c_r, i_r)$ using the following formula with $y \in \{l, r\}$:

$$A'_y(c_y, i_y) = \begin{cases} A_y(c_y) & \text{if } \#_1(c_y(I))) = i_y \\ 0 & \text{otherwise} \end{cases}$$

Here, $\#_1(c_y(I))$ is the number of 1-entries in the colouring c_y on the vertices in I.

Next, we apply state changes by using Lemma 12.10. In this case, we change the states used for the colourings in A'_r and A'_l such that they use the set of states $\{0, 1\}$

on L, R, and F, and the set of states $\{0, ?\}$ on I. Notice that the number of 1-states used to create the ?-states is now stored in the index i_l of $A'_l(c, i_l)$ and i_r of $A'_r(c, i_r)$.

We say that three colourings c_e of X_e, c_l of X_l and c_r of X_r using these sets of states on the different partitions of $X_e \cup X_l \cup X_r$ *match* if:

- For any $v \in I$: $c_e(v) = c_l(v) = c_r(v) \in \{0, ?\}$. (2 possibilities)
- For any $v \in F$: either $c_l(v) = 0$ and $c_r(v) = 1$, or $c_l(v) = 1$ and $c_r(v) = 0$. (2 possibilities)
- For any $v \in L$: $c_e(v) = c_l(v) \in \{1, 0\}$. (2 possibilities)
- For any $v \in R$: $c_e(v) = c_r(v) \in \{1, 0\}$. (2 possibilities)

Now, we can compute the indexed table A'_e for the edge e of T using the following formula:

$$A'_e(c_e, i_e) = \sum_{c_e, c_l, c_r \text{ match}} \sum_{i_l + i_r = i_e} A'_l(c_l, i_l) \cdot A'_r(c_r, i_r)$$

Notice that we can compute A'_e efficiently by using a series of matrix multiplications in the same way as done in the proof of Theorem 12.7. However, the index i should be treated slightly differently from the parameter κ in the proof of Theorem 12.7. After fixing a colouring on I and the two values of i_e and i_l, we still create the two matrices M_l and M_r. In M_l each row corresponds to a colouring of L and each column corresponds to a colouring of F, and in M_r each rows again corresponds to a colourings of F and each column corresponds to a colouring of R. The difference is that we fill M_l with the corresponding entries $A'_l(c_l, i_l)$ and M_r with the corresponding entries $A'_r(c_r, i_r)$. That is, we do not adjust the value of i_r for the selected $A'_r(c_r, i_r)$ depending on the states used on F. This is not necessary here since the index counts the total number of 1-states hidden in the ?-states and no double counting can take place. This in contrast to the parameter κ in the proof of Theorem 12.7; this parameter counted the number of vertices in a solution, which we had to correct to avoid double counting of the vertices in F.

After computing A'_e in this way, we again change the states such that the set of states $\{1, 0\}$ is used on all vertices in the colourings used in A'_e. We then extract the values of A'_e in which no two 1-states hidden in a ?-state are combined to a new 1-state on a vertex in I. We do so using the indices in the same way as in Theorem 11.13 but with the counting restricted to I:

$$A_e(c) = A'_e(c, \#_1(c(I)))$$

After computing the A_e for all $e \in E(T)$, we can find the number of perfect matchings in $G = G_{\{y,z\}}$ in the single entry in $A_{\{y,z\}}$ where z is the root of T and y is its only child.

Because the treewidth and branchwidth of a graph differ by at most a factor $\frac{3}{2}$ (see Proposition 12.2), we can conclude that the computations can be done using $\mathcal{O}(k \log(n))$-bit numbers using the same reasoning as in the proof of Theorem 11.13. For the running time, we observe that we can compute each A_e using a series of $k^2 2^{|I|}$ matrix multiplications. The worst case arises when $|I| = 0$ and $|L| = |R| = |F| = \frac{k}{2}$. Then the matrix multiplications require $\mathcal{O}(k^2 2^{\frac{\omega}{2}k})$ time. Since T has $\mathcal{O}(m)$ edges, this gives a running time of $\mathcal{O}(mk^2 2^{\frac{\omega}{2}k} i_\times(k \log(n)))$ time. □

12.4. $[\rho, \sigma]$-Domination Problems

We have shown how to solve two fundamental graph problems in $\mathcal{O}^*(s^{\frac{\omega}{2}k})$ time on branch decompositions of width k, where s is the natural number of states involved in a dynamic programming algorithm on branch decompositions for these problem. Similar to the results on tree decompositions, we generalize this and show that one can solve all $[\rho, \sigma]$-domination problems with finite or cofinite ρ and σ in $\mathcal{O}^*(s^{\frac{\omega}{2}k})$ time.

For the $[\rho, \sigma]$-domination problems, we use states ρ_j and σ_j, where ρ_j and σ_j represent that a vertex is not in or in the vertex set D of the partial solution of the $[\sigma, \rho]$-domination problem, respectively, and has j neighbours in D. This is similar to Section 11.5. Note that the number of states used equals $s = p + q + 2$.

On branch decompositions, we have to use a different approach than on tree decompositions, since we have to deal with vertices in L, R, I, and F simultaneously. It is, however, possible to reuse part of the algorithm of Theorem 11.17. Observe that joining two children in a tree decomposition is similar to joining two children in a branch decomposition if $L = R = F = \emptyset$. Since we have demonstrated in the algorithms earlier in this chapter that one can have distinct states and perform different computations on I, L, R, and F, we can essentially use the approach of Theorem 11.17 for the vertices in I.

Theorem 12.12. *Let $\rho, \sigma \subseteq \mathbb{N}$ be finite or cofinite. There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of $[\rho, \sigma]$-dominating sets of G of each size κ, $0 \leq \kappa \leq n$, of a fixed $[\rho, \sigma]$-domination problem involving s states in $\mathcal{O}(mn^2(sk)^{2(s-2)}s^{\frac{\omega}{2}k}i_\times(n))$ time.*

Proof. Let T be the branch decomposition of G of width k rooted at the vertex z.

Recall the definitions of State Sets I and II defined in Definition 11.14. Similar to the proof of Theorem 11.17, we will use different sets of states to prove this theorem. In this proof, we mostly use State Set I while we let the subscripts of the states count only neighbours in D outside the current middle set. That is, we use states ρ_j and σ_j for our tables A_e, A_f, and A_g such that the subscripts j represent the number of neighbours in the vertex set D of each partial solution of the $[\sigma, \rho]$-domination problem outside of the vertex sets X_e, X_f and X_g, respectively. Using these states for colourings c, we compute the table A_e for each edge $e \in E(T)$ such that the entry $A_e(c, \kappa)$ contains the number of partial solutions of the $[\rho, \sigma]$-domination problem on G_e consisting of κ vertices that satisfy the requirements defined by c.

Leaf edges: Let e be an edge of T incident to a leaf of T that is not the root. Now, $G_e = G[X_e]$ is a two vertex graph.

We compute A_e in the following way:

$$A_e(c, \kappa) = \begin{cases} 1 & \text{if } c = (\rho_0, \rho_0) \text{ and } \kappa = 0 \\ 1 & \text{if } c = (\rho_0, \sigma_0) \text{ or } c = (\sigma_0, \rho_0), \text{ and } \kappa = 1 \\ 1 & \text{if } c = (\sigma_0, \sigma_0) \text{ and } \kappa = 2 \\ 0 & \text{otherwise} \end{cases}$$

Since the subscripts of the states count only vertices in the vertex set of a partial solutions of the $[\rho, \sigma]$-domination problem on G_e that are outside the middle set X_e, we only count partial solutions in which the subscripts are zero. Moreover, the size

parameter κ must equal the number of σ-states since these represent vertices in the vertex set of the partial solutions.

Internal edges: Let e be an internal edge of T with child edges l and r.

The process of computing the table A_e by combining the information in the two tables A_l and A_r is quite technical. This is mainly due to the fact that we need to do different things on the different vertex sets I, L, R, and F. We will give a three-step proof.

Step 1: As a form of preprocessing, we will update the entries in A_l and A_r such that the subscripts will not count only the vertices in vertex sets of the partial solutions outside of X_l and X_r, but also some specific vertices in the vertex sets of the partial solutions in the middle sets. Later, we will combine the information from A_l and A_r to create the table A_e according to the following general rule: combining ρ_i and ρ_j gives ρ_{i+j}, and σ_i and σ_j gives σ_{i+j}. In this context, the preprocessing makes sure that the subscripts of the states in the result in A_e correctly count the number of vertices in the vertex sets of the partial solutions of the $[\rho, \sigma]$-domination problem.

Recall that for an edge e of the branch decomposition T the vertex set V_e is defined to be the vertex set of the graph G_e, that is, the union of the middle set of e and all middle sets below e in T. We update the tables A_l and A_r such that the subscripts of the states ρ_j and σ_j count the number of neighbours in the vertex sets of the partial solutions with the following properties:

- States used in A_l on vertices in L or I count neighbours in $(V_l \setminus X_l) \cup F$.
- States used in A_l on vertices in F count neighbours in $(V_l \setminus X_l) \cup L \cup I \cup F$.
- States used in A_r on vertices in I count neighbours in $(V_r \setminus X_r)$ (nothing changes here).
- States used in A_r on vertices in R count neighbours in $(V_r \setminus X_r) \cup F$.
- States used in A_r on vertices in F count neighbours in $(V_r \setminus X_r) \cup R$.

If we now combine partial solutions with state ρ_i in A_l and state ρ_j in A_r for a vertex in I, then the state ρ_{i+j} corresponding to the combined solution in A_e correctly counts the number of neighbours in the partial solution in $V_e \setminus X_e$. Also, states for vertices in L and R in A_e count their neighbours in the partial solution in $V_e \setminus X_e$. And, if we combine solutions with a state ρ_i in A_l and a state ρ_j in A_r for a vertex in F, then this vertex will have exactly $i + j$ neighbours in the combined partial solution.

Although one must be careful which vertices to count and which not to count, the actual updating of the tables A_l and A_r is simple because one can see which of the counted vertices are in the vertex set of a partial solution (σ-state) and which are not (ρ-state).

Let A_y^* be the table before the updating process with $y \in \{l, r\}$. We compute the updated table A_y in the following way:

$$A_y(c, \kappa) = \begin{cases} 0 & \text{if } \phi(c) \text{ is not a correct colouring of } X_y \\ A_y^*(\phi(c), \kappa) & \text{otherwise} \end{cases}$$

Here, ϕ is the inverse of the function that updates the subscripts of the states, e.g., if $y = l$ and we consider a vertex in I with exactly one neighbour with a σ-state on a vertex in F in c, then it changes ρ_2 into ρ_1. The result of this updating is not a correct colouring of X_y if the inverse does not exist, i.e., if the strict application of subtracting

the right number of neighbours results in a negative number. For example, this happens if c contains a ρ_0- or σ_0-state while it has neighbours that should be counted in the subscripts.

Step 2: Next, we will change the states used for the tables A_l and A_r, and we will add index vectors to these tables that allow us to use the ideas of Theorem 11.17 on the vertices in I.

We will not change the states for vertices in L in the table A_l, nor for vertices in R in the table A_r. But, we will change the states for the vertices in I in both A_l and A_r and on the vertices in F in A_r. On F, simple state changes suffice, while, for vertices on I, we need to change the states and introduce index vectors at the same time.

We will start by changing the states for the vertices in F. On the vertices in F, we will not change the states in A_l, but introduce a new set of states to use for A_r. We define the states $\bar{\rho_j}$ and $\bar{\sigma_j}$. A table entry with state $\bar{\rho_j}$ on a vertex v requires that the vertex has an allowed number of neighbours in the vertex set of a partial solution when combined with a partial solution from A_l with state ρ_j. That is, a partial solution that corresponds to the state ρ_i on v is counted in the entry with state $\bar{\rho_j}$ on v if $i + j \in \rho$. The definition of $\bar{\sigma_j}$ is similar.

Let A_r^* be the result of the table for the right child r of e obtained by Step 1. We can obtain the table A_r with the states on F transformed as described by a coordinate-wise application of the following formula on the vertices in F. The details are identical to the state changes in the proofs of Lemmas 12.5 and 12.10.

$$\begin{aligned} A_r(c_1 \times \{\bar{\rho_j}\} \times c_2) &= \sum_{i+j\in\rho} A_r^*(c_1 \times \{\rho_i\} \times c_2) \\ A_r(c_1 \times \{\bar{\sigma_j}\} \times c_2) &= \sum_{i+j\in\sigma} A_r^*(c_1 \times \{\sigma_i\} \times c_2) \end{aligned}$$

Notice that if we combine an entry with state ρ_j in A_l with an entry with state $\bar{\rho_j}$ from A_r, then we can count all valid combinations in which this vertex is not in the vertex set of a partial solution of the $[\rho, \sigma]$-domination problem. The same is true for a combination with state σ_j in A_l with state $\bar{\sigma_j}$ in A_r for vertices in the vertex set of the partial solutions.

As a final part of Step 2, we now change the states in A_l and A_r on the vertices in I and introduce the index vectors $\vec{i} = (i_{\rho 1}, i_{\rho 2}, \ldots, i_{\rho p}, i_{\sigma 1}, i_{\sigma 2}, \ldots, i_{\sigma q})$, where $i_{\rho j}$ and $i_{\sigma j}$ index the sum of the number of neighbours in the vertex set of a partial solution of the $[\rho, \sigma]$-domination problem over the vertices with state $\rho_{\leq j}$ and $\sigma_{\leq j}$, respectively. That is, we change the states used in A_l and A_r on vertices in I to State Set II of Definition 11.14 and introduce index vectors in exactly the same way as in the proof of Lemma 11.16, but only on the coordinates of the vertices in I, similar to what we did in the proofs of Lemmas 12.5 and 12.10. Because the states $\rho_{\leq j}$ and $\sigma_{\leq j}$ are used only on I, we note that that the component i_{ρ_j} of the index vector $\vec{i}$ count the total number of neighbours in the vertex sets of the partial solutions of the $[\rho, \sigma]$-domination problem of vertices with state $\rho_{\leq j}$ on I. As a result, we obtain tables A_l' and A_r' with entries $A_l'(c_l, \kappa_l, \vec{g})$ and $A_r'(c_r, \kappa_r, \vec{h})$ with index vectors $\vec{g}$ and $\vec{h}$, where these entries have the same meaning as in Theorem 11.17. We note that the components $i_{\rho p}$ and $i_{\sigma q}$ of the index vector are omitted if ρ or σ is cofinite, respectively.

We have now performed all relevant preprocessing and are ready for the final step.

Step 3: Now, we construct the table A_e by computing the number of valid combinations from A'_l and A'_r using fast matrix multiplication.

We first define when three colourings c_e, c_l, and c_r match. They *match* if:

- For any $v \in I$: $c_e(v) = c_l(v) = c_r(v)$ with State Set II. (s possibilities)
- For any $v \in F$: either $c_l(v) = \rho_j$ and $c_r(v) = \bar{\rho_j}$, or $c_l(v) = \sigma_j$ and $c_r(v) = \bar{\sigma_j}$, with State Set I used for c_l and the new states used for c_r. (s possibilities)
- For any $v \in L$: $c_e(v) = c_l(v)$ with State Set I. (s possibilities)
- For any $v \in R$: $c_e(v) = c_r(v)$ with State Set I. (s possibilities)

State Set I and State Set II are as defined in Definition 11.14. That is, colourings match if they forget valid combinations on F, and have identical states on I, L, and R.

Using this definition, the following formula computes the table A'_e. The function of this table is identical to the same table in the proof of Theorem 11.17: the table gives all valid combinations of entries corresponding to the colouring c that lead to a partial solution of size κ with the given values of the index vector $\vec{i}$. The index vectors allow us to extract the values we need afterwards.

$$A'_e(c_e, \kappa, \vec{i}) = \sum_{\substack{c_e, c_l, c_r \\ \text{match}}} \sum_{\kappa_l + \kappa_r = \kappa + \#_\sigma(c)} \left(\sum_{i_{\rho 1} = g_{\rho 1} + h_{\rho 1}} \cdots \sum_{i_{\sigma q} = g_{\sigma q} + h_{\sigma q}} A'_l(c_l, \kappa_l, \vec{g}) \cdot A'_r(c_r, \kappa_r, \vec{h}) \right)$$

Here, $\#_\sigma = \#_\sigma(c_r(I \cup F))$ is the number of vertices that are assigned a σ-state on $I \cup F$ in any matching triple c_e, c_l, c_r.

We will now argue what kind of entries the table A'_e contains by giving a series of observations.

Observation 12.1. For a combination of a partial solutions on G_l counted in A'_l and a partial solution on G_r counted in A'_r to be counted in the summation for $A'_e(c, \kappa, \vec{i})$, it is required that both partial solutions contains the same vertices on $X_l \cap X_r$ ($= I \cap F$).

Proof. This holds because sets of matching colourings have a σ-state on a vertex if and only if the other colourings in which the same vertex is included also have a σ-state on this vertex. □

Observation 12.2. For a combination of a partial solutions on G_l counted in A'_l and a partial solution on G_r counted in A'_r to be counted in the summation for $A'_e(c, \kappa, \vec{i})$, it is required that the total number of vertices that are part of the combined partial solution is κ.

Proof. This holds because we demand that κ equals the sum of the sizes of the partial solutions on G_l and G_r used for the combination minus the number of vertices in these partial solutions that are counted in both sides, namely, the vertices with a σ-state on I or F. □

Observation 12.3. For a combination of a partial solutions on G_l counted in A'_l and a partial solution on G_r counted in A'_r to be counted in the summation for $A'_e(c, \kappa, \vec{i})$, it is required that the subscripts j of the states ρ_j and σ_j used in c on vertices in L and R correctly count the number of neighbours of this vertex in $V_e \setminus X_e$ in the combined partial solution.

Proof. This holds because of the preprocessing we performed in Step 1. □

Observation 12.4. For a combination of a partial solutions on G_l counted in A'_l and a partial solution on G_r counted in A'_r to be counted in the summation for $A'_e(c, \kappa, \vec{i})$, it is required that the forgotten vertices in a combined partial solution satisfy the requirements imposed by the specific $[\rho, \sigma]$-domination problem. I.e., if such a vertex is not in the vertex set D of the combined partial solutions, then it has a number of neighbours in D that is a member of ρ, and if such a vertex is in the vertex set D of the combined partial solution, then it has a number of neighbours in D that is a member of σ. Moreover, all such combinations are considered.

Proof. This holds because we combine only entries with the states ρ_j and $\bar{\rho_j}$ or with the states σ_j and $\bar{\sigma_j}$ for vertices in F. These are all required combinations by definition of the states $\bar{\rho_j}$ and $\bar{\sigma_j}$. □

Observation 12.5. For a combination of a partial solutions on G_l counted in A'_l and a partial solution on G_r counted in A'_r to be counted in the summation for $A'_e(c, \kappa, \vec{i})$, it is required that the total sum of the number of neighbours outside X_e of the vertices with state $\rho_{\leq j}$ or $\sigma_{\leq j}$ in a combined partial solution equals $i_{\rho j}$ or $i_{\sigma j}$, respectively.

Proof. This holds because of the following. First the subscripts of the states are updated such that every relevant vertex is counted exactly once in Step 1. Then, these numbers are stored in the index vectors at Step 2. Finally, the entries of A'_e corresponding to a given index vector combine only partial solutions which index vectors sum to the given index vector $\vec{i}$. □

Observation 12.6. Let D_l and D_r be the vertex set of a partial solution counted in A_l and A_r that are used to create a combined partial solution with vertex set D, respectively. After the preprocessing of Step 1, the vertices with state $\rho_{\leq j}$ or $\sigma_{\leq j}$ have at most j neighbours that we count in the vertex sets D_l and D_r, respectively. And, if a vertex in the partial solution from A_l has i such counted neighbours in D_l, and the same vertex in the partial solution from A_r has j such counted neighbours in D_r, then the combined partial solution has a total of $i + j$ neighbours in D outside of X_e.

Proof. The last statement holds because we count each relevant neighbour of a vertex either in the states used in A_l or in the states used in A_r by the preprocessing of Step 1. The first part of the statement follows from the definition of the states $\rho_{\leq j}$ or $\sigma_{\leq j}$: here, only partial solutions that previously had a state ρ_i and σ_i with $i \leq j$ are counted. □

We will now use Observations 12.1-12.6 to show that we can compute the required values for A_e in the following way. This works very similar to Theorem 11.17. First, we change the states in the table A'_e back to State Set I (as defined in Definition 11.14). We can do so similar as in Lemma 11.15 and Lemmas 12.5 and 12.10. Then, we extract the entries required for the table A_e using the following formula:

$$A_e(c, \kappa) = A'_e\left(c,\ \kappa,\ (\Sigma^1_\rho(c), \Sigma^2_\rho(c), \ldots, \Sigma^p_\rho(c), \Sigma^1_\sigma(c), \Sigma^2_\sigma(c), \ldots, \Sigma^q_\sigma(c))\ \right)$$

Here, $\Sigma^l_\rho(c)$ and $\Sigma^l_\sigma(c)$ are defined as in the proof of Theorem 11.17: the weighted sums of the number of ρ_j- and σ_j-states with $0 \leq j \leq l$, respectively.

If ρ or σ is cofinite, we use the same formula but omit the components $\Sigma_\rho^p(c)$ or $\Sigma_\sigma^q(c)$ from the index vector of the extracted entries, respectively.

That the values of these entries equal the values we want to compute follows from the following reasoning. First of all, any combination considered leads to a new partial solution since it uses the same vertices (Observation 12.1) and forgets vertices that satisfy the constraints of the fixed $[\rho, \sigma]$-domination problem (Observation 12.4). Secondly, the combinations lead to combined partial solutions of the correct size (Observation 12.2). Thirdly, the subscripts of the states used in A_e correctly count the number of neighbours of these vertices in the vertex set of the partial solution in $V_e \setminus X_e$. For vertices in L and R, this directly follows from Observation 12.3 and the fact that for any three matching colourings the states used on each vertex in L and R are the same. For vertices in I, this follows from exactly the same arguments as in the last part of the proof of Theorem 11.17 using Observation 12.5 and Observation 12.6. This is the argument where we first argue that any entry which colouring uses only the states ρ_0 and σ_0 is correct, and thereafter inductively proceed to ρ_j and σ_j for $j > 0$ by using correctness for ρ_{j-1} and σ_{j-1} and fact that we use the entries corresponding to the chosen values of the index vectors.

All in all, we see that this procedure correctly computes the required table A_e.

After computing A_e in the above way for all $e \in E(T)$, we can find the number of $[\rho, \sigma]$-dominating sets of each size in the table $A_{\{y,z\}}$, where z is the root of T and y its only child because $G = G_{\{y,z\}}$ and $X_{\{y,z\}} = \emptyset$.

For the running time, we note that we have to compute the tables A_e for the $\mathcal{O}(m)$ edges $e \in E(T)$. For each table A_e, the running time is dominated by evaluating the formula for the intermediate table A'_e with entries $A'_e(c, \kappa, \vec{i})$. We can evaluate each summand of the formula for A'_e for all combinations of matching states by $s^{|I|}$ matrix multiplications as in Theorem 12.7. This requires $\mathcal{O}(n^2(sk)^{2(s-2)}s^{|I|})$ multiplications of an $s^{|L|} \times s^{|F|}$ matrix and an $s^{|F|} \times s^{|R|}$ matrix. The running time is maximal if $|I| = 0$ and $|L| = |R| = |F| = \frac{k}{2}$. In this case, the total running time equals $\mathcal{O}(mn^2(sk)^{2(s-2)}s^{\frac{\omega}{2}k}i_\times(n))$ since we can do the computations using n-bit numbers. □

Similar to our results on the $[\rho, \sigma]$-domination problem on tree decompositions, we can improve the polynomial factors of the above algorithm in several ways. The techniques involved are identical to those of Corollaries 11.18, 11.19, and 11.20. Similar to Section 11.5, we define the value r associated with a $[\rho, \sigma]$-domination problems as follows:

$$r = \begin{cases} \max\{p-1, q-1\} & \text{if } \rho \text{ and } \sigma \text{ are cofinite} \\ \max\{p, q-1\} & \text{if } \rho \text{ is finite and } \sigma \text{ is cofinite} \\ \max\{p-1, q\} & \text{if } \rho \text{ is confinite and } \sigma \text{ is finite} \\ \max\{p, q\} & \text{if } \rho \text{ and } \sigma \text{ are finite} \end{cases}$$

Corollary 12.13 (General $[\rho, \sigma]$-Domination Problems). *Let $\rho, \sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q, r and s be the values associated with the corresponding $[\rho, \sigma]$-domination problem. There is an algorithm that, given a branch decomposition of a graph G of width k, computes the number of $[\rho, \sigma]$-dominating sets in G of each size κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(mn^2(rk)^{2r}s^{\frac{\omega}{2}k}i_\times(n))$ time. Moreover, there is an algorithm that decides whether there exist a $[\rho, \sigma]$-dominating set of size κ, for each individual value of κ, $0 \leq \kappa \leq n$, in $\mathcal{O}(mn^2(rk)^{2r}s^{\frac{\omega}{2}k}i_\times(log(n) + k\log(r)))$ time.*

Proof. Apply the modifications to the algorithm of Theorem 11.17 that we have used in the proof of Corollary 11.18 for $[\rho,\sigma]$-domination problems on tree decompositions to the algorithm of Theorem 12.12 for the same problems on branch decompositions. □

Corollary 12.14 ($[\rho,\sigma]$-Optimisation Problems with the de Fluiter Property). *Let $\rho,\sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q, r and s be the values associated with the corresponding $[\rho,\sigma]$-domination problem. If the standard representation using State Set I of the minimisation (or maximisation) variant of this $[\rho,\sigma]$-domination problem has the de Fluiter property for treewidth with function f, then there is an algorithm that, given a branch decomposition of a graph G of width k, computes the number of minimum (or maximum) $[\rho,\sigma]$-dominating sets in G in $\mathcal{O}(m[f(k)]^2(rk)^{2r}s^{\frac{\omega}{2}k}i_\times(n))$ time. Moreover, there is an algorithm that computes the minimum (or maximum) size of such a $[\rho,\sigma]$-dominating set in $\mathcal{O}(m[f(k)]^2(rk)^{2r}s^{\frac{\omega}{2}k}i_\times(log(n)+k\log(r)))$ time.*

Proof. Improve the result of Corollary 12.13 in the same way as Corollary 11.19 improves Corollary 11.18 on tree decompositions. □

Corollary 12.15 ($[\rho,\sigma]$-Decision Problems). *Let $\rho,\sigma \subseteq \mathbb{N}$ be finite or cofinite, and let p, q, r and s be the values associated with the corresponding $[\rho,\sigma]$-domination problem. There is an algorithm that, given a branch decomposition of a graph G of width k, counts the number of $[\rho,\sigma]$-dominating sets in G of a fixed $[\rho,\sigma]$-domination problem in $\mathcal{O}(m(rk)^{2r}s^{\frac{\omega}{2}k}i_\times(n))$ time. Moreover, there is an algorithm that decides whether there exists a $[\rho,\sigma]$-dominating set in $\mathcal{O}(m(rk)^{2r}s^{\frac{\omega}{2}k}i_\times(log(n)+k\log(r)))$ time.*

Proof. Improve the result of Corollary 12.14 in the same way as Corollary 11.20 improves upon Corollary 11.19 on tree decompositions. □

12.5. Concluding Remarks

In this chapter, we have given the currently fastest algorithms on branch decompositions of width k, when the running time is expressed as a function of k, for #Perfect Matching and the $[\rho,\sigma]$-domination problems. These algorithms attain the same time bounds as earlier algorithms on branch decompositions of planar graphs by Dorn [110, 111]. Our results were obtained by combining the techniques introduced in Chapter 11 with fast matrix multiplication. For this combined approach to work, we introduced the use of asymmetric vertex states on vertices that we forget while computing the dynamic programming table for the next edge of the branch decomposition.

We observe that our branchwidth-based algorithms are efficient algorithms for problems such as #Perfect Matching, both in theory and in practice. Namely, any graph G has branchwidth at most $\lceil\frac{2}{3}n\rceil$ because the branchwidth of a clique of size n is $\lceil\frac{2}{3}n\rceil$; see for example [183]. As a result, we directly obtain an $\mathcal{O}(2^{\frac{\omega}{2}\cdot\frac{2}{3}n}) = \mathcal{O}(1.7315^n)$-time algorithm for this problem. This running time is identical to the fast-matrix-multiplication based algorithm for this problem by Björklund et al. [27]. We note that this result has recently been improved to $\mathcal{O}^*(1.6181^n)$-time by Koivisto [208]. Our algorithm improves this result on graphs for which we can compute a branch decomposition of width at most $0.5844n$ in polynomial time; this is a very large family of graphs since this bound is not much smaller than the given upper bound of $\frac{2}{3}n$.

13 Fast Dynamic Programming on Clique Decompositions

After considering treewidth-based algorithms in Chapter 11 and branchwidth-based algorithms in Chapter 12, the final results that we present in this thesis concern cliquewidth-based algorithms. In this chapter, we will give the currently fastest cliquewidth-based algorithms when the running time is expressed as a function of the cliquewidth k. We obtain these results by extending the techniques introduced in Chapter 11 to this graph-width parameter, which is quite different from the other two.

The notion of cliquewidth was first studied by Courcelle et al. [85]. The graph decomposition associated with cliquewidth is a k-expression, which is sometimes also called a clique decomposition. Whereas the treewidth and branchwidth of any graph are closely related, its cliquewidth can be very different from both of them. For example, the treewidth of the complete graph on n vertices is equal to $n-1$, while its cliquewidth is equal to 2. However, the cliquewidth of a graph is always bounded by a function of its treewidth [87]. This makes cliquewidth an interesting graph parameter to consider on graphs where the treewidth or branchwidth is too high for obtaining efficient algorithms.

In this chapter, we will give a very fast algorithm for Dominating Set on graphs given with a clique decomposition of cliquewidth k. The first algorithm for this problem on clique decompositions is an $\mathcal{O}^*(16^k)$-time algorithm by Kobler and Rotics [206]. The previously fastest algorithm for this problem has a running time of $\mathcal{O}^*(8^k)$, obtained by transforming the problem to a problem on boolean decompositions [62]. We present a direct algorithm that runs in $\mathcal{O}^*(4^k)$ time. We also show that one can also count the number of dominating sets of each size κ, $0 \leq \kappa \leq n$, at the cost of an extra polynomial

[†]This chapter is joint work with Hans L. Bodlaender, Erik Jan van Leeuwen and Martin Vatshelle. The chapter contains results of which a preliminary version has been presented at the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010) [50].

factor. Furthermore, we show that one can solve Independent Dominating Set and Total Dominating Set in $\mathcal{O}^*(4^k)$ time.

We note that there are other width parameters that potentially have lower values than cliquewidth, for example rankwidth (see [248]) or booleanwidth (see [62]). Also, a problem is in $\mathcal{FPT}$ when parameterised by cliquewidth if and only if it is in $\mathcal{FPT}$ when parameterised by rankwidth or booleanwidth [62, 248]. However, for many problems the best known running times for these problems are often much better as a function of the cliquewidth than as function of the rankwidth or booleanwidth.

Similar to Chapters 11 and 12, we use the notation $i_+(n)$ and $i_\times(n)$ for the time required to multiply and add n-bit numbers, respectively. Also, we analyse our algorithms in the *Random Access Machine* (RAM) model with $\mathcal{O}(k)$-bit word size [157], where addition and multiplication are unit-time operations ($i_+(k) = i_\times(k) = \mathcal{O}(1)$). We do so because of reasons similar to those given in Chapter 11.

This chapter is organised as follows. First, we introduce the reader to cliquewidth and the related graph decompositions in Section 13.1. Then, we give our results on cliquewidth-based algorithms in Section 13.2. Finally, we give some concluding remarks in Section 13.3

13.1. *k*-Expressions and Cliquewidth

In this section, we introduce cliquewidth and k-expressions or clique decompositions. Cliquewidth is, like treewidth and branchwidth, a notion related to the decomposition of graphs. This notion was introduced by Courcelle et al. [85].

A labelled graph with set of labels L is a graph $G = (V, E)$ where each vertex $v \in V$ is assigned a label from L.

Definition 13.1 (*k*-Expression). A *k-expression* is an expression combining any number of the following four operations on labelled graphs with labels $\{1, 2, \ldots, k\}$:

1. *create a new graph*: create a new graph with one vertex having any label.
2. *relabel*: relabel all vertices with label i to j ($i \neq j$).
3. *add edges*: connect all vertices with label i to all vertices with label j ($i \neq j$).
4. *join graphs*: take the disjoint union of two labelled graphs.

The *cliquewidth* $cw(G)$ of a graph G is defined to be the minimum k for which there exists a k-expression that evaluates to a graph isomorphic to G.

The definition of a k-expression can also be turned into a rooted decomposition tree. In this decomposition tree T, leaves of the tree T correspond to the operations that create new graphs, effectively creating the vertices of G, and internal vertices of T correspond to one of the other three operations described above. We call this tree a *clique decomposition* of width k.

Like the other two width parameters, computing the cliquewidth of general graphs is $\mathcal{NP}$-hard [130]. However, graphs of cliquewidth 1, 2, or 3 can be recognised in polynomial time [84]. Also, for $k \geq 4$, there is an $\mathcal{FPT}$-algorithm that, given a graph of cliquewidth k, outputs a 2^{k+1}-expression [248]. In this chapter, we always assume that a given decomposition of the appropriate width is given.

In this chapter, we also assume that any k-expression does not contain superfluous operations, e.g., a k-expression does apply the operation to add edges between vertices with labels i and j twice in a row without first changing the sets of vertices with the labels i and j, and it does not relabel vertices with a given label or add edges between vertices with a given label if the set of vertices with such a label is empty. Under these conditions, it is not hard to see that any k-expressions consists of at most $\mathcal{O}(n)$ join operations and at most $\mathcal{O}(nk^2)$ other operations.

For more information on algorithms on graphs of bounded cliquewidth, see [86].

13.2. Domination Problems on Clique Decompositions

On graphs of bounded cliquewidth, we mainly consider the Dominating Set problem. We will show how to improve the complex $\mathcal{O}^*(8^k)$-time algorithm, which computes a boolean decomposition of a graph of cliquewidth at most k to solve Dominating Set [62], to an $\mathcal{O}^*(4^k)$-time algorithm that involves only clique decompositions. Similar $\mathcal{O}^*(4^k)$-time algorithm for Independent Dominating Set and Total Dominating Set follow from the same approach.

Theorem 13.2. *There is an algorithm that, given a k-expression for a graph G, computes the number of dominating sets in G of each size $0 \leq \kappa \leq n$ in $\mathcal{O}(n^3(k^2+i_\times(n))\,4^k)$ time.*

Proof. An operation in a k-expression applies a procedure on zero, one, or two labelled graphs with labels $\{1, 2, \ldots, k\}$ that transforms these labelled graphs into a new labelled graph with the same set of labels. If H is such a labelled graph with vertex set V, then we use $V(i)$ to denote the vertices of H with label i.

For each labelled graph H obtained by applying an operation in a k-expression, we will compute a table A with entries $A(c, \kappa)$ that store the number of partial solutions of Dominating Set of size κ that satisfy the constraints defined by the colouring c. In contrast to the algorithms on tree and branch decompositions, we do not use colourings that assign a state to each individual vertex, but colourings that assign states to the sets $V(1), V(2), \ldots, V(k)$.

The states that we use are similar to the ones used for Dominating Set on tree decompositions and branch decomposition. The states that we use are tuples representing two attributes: inclusion and domination. The first attribute determines whether at least one vertex in $V(i)$ is included in a partial solution. We use states 1, 0, and ? to indicate whether this is true, false, or any of both, respectively. The second attribute determines whether all vertices of $V(i)$ are dominated in a partial solution. Here, we also use states 1, 0, and ? to indicate whether this is true, false, or any of both, respectively. Thus, we get tuples of the form (s, t), where the first components is related to inclusion and the second to domination, e.g., $(1, ?)$ for vertex set $V(i)$ represents that the vertex set contains a vertex in the dominating set while we are indifferent about whether all vertices in $V(i)$ are dominated.

We will now show how to compute the table A for a k-expression obtained by using any of the four operations on smaller k-expressions that are given with similar tables for these smaller k-expressions. This table A contains an entry for every colouring c of the series of vertex sets $\{V(1), V(2), \ldots, V(k)\}$ using the four states $(1, 1)$, $(1, 0)$,

$(0,1)$, and $(0,0)$. We note that the other states will be used to perform intermediate computations. By a recursive evaluation, we can compute A for the k-expression that evaluates to G.

Create a new graph: In this operation, we create a new graph H with one vertex v with any label $j \in \{1,2,\ldots,k\}$. We assume, without loss of generality by permuting the labels, that $j = k$. We compute A by using the following formula where c is a colouring of the first $k-1$ vertex sets $V(i)$ and c_k is the state of $V(k)$:

$$A(c \times \{c_k\}, \kappa) = \begin{cases} 1 & \text{if } c_k = (1,1),\ \kappa = 1, \text{ and } c = \{(0,1)\}^{k-1} \\ 1 & \text{if } c_k = (0,0),\ \kappa = 0, \text{ and } c = \{(0,1)\}^{k-1} \\ 0 & \text{otherwise} \end{cases}$$

Since H has only one vertex and this vertex has label k, the vertex sets for the other labels cannot have a dominating vertex, therefore the first attribute of their state must be 0. Also, all vertices in these sets are dominated, hence the second attribute of their state must be 1. The above formula counts the only two possibilities: either taking the vertex in the partial solution or not.

Relabel: In this operation, all vertices with some label $i \in \{1,2,\ldots,k\}$ are relabelled such that they obtain the label $j \in \{1,2,\ldots,k\}$, $j \neq i$. We assume, without loss of generality by permuting the labels, that $i = k$ and $j = k-1$.

Let A' be the table belonging to the k-expression before the relabelling and let A be the table we need to compute. We compute A using the following formulas:

$$A(c \times \{(0,1)\} \times \{(i,d)\}, \kappa) = \sum_{\substack{i_1,i_2 \in \{0,1\} \\ \max\{i_1,i_2\}=i}} \sum_{\substack{d_1,d_2 \in \{0,1\} \\ \min\{d_1,d_2\}=d}} A'(c \times \{(i_1,d_1)\} \times \{(i_2,d_2)\}, \kappa)$$

$$A(c \times \{(i^*,d^*)\} \times \{(i,d)\}, \kappa) = 0 \qquad \text{if } (i^*,d^*) \neq (0,1)$$

These formula correctly compute the table A because of the following observations. For $V(i)$, the first attribute must be 0 and the second attribute must be 1 in any valid partial solution because $V(i) = \emptyset$ after the operations; this is similar to this requirement in the 'create new graph' operation. If $V(j)$ must have a vertex in the dominating set, then this vertex must be in $V(i)$ or $V(j)$ originally. And, if all vertices in $V(j)$ must be dominated, then all vertices in $V(i)$ and $V(j)$ must be dominated. Note that the minimum and maximum under the summations correspond to 'and' and 'or' operations, respectively.

Add edges: In this operation, all vertices with some label $i \in \{1,2,\ldots,k\}$ are connected to all vertices with another label $j \in \{1,2,\ldots,k\}$, $j \neq i$. We again assume, without loss of generality by permuting the labels, that $i = k-1$ and $j = k$.

Let A' be the table belonging to the k-expression before adding the edges and let A be the table we need to compute. We compute A using the following formula:

$$A(c \times \{(i_1,d_1)\} \times \{(i_2,d_2)\}, \kappa) = \sum_{\substack{d'_1 \in \{0,1\} \\ \max\{d'_1,i_2\}=d_1}} \sum_{\substack{d'_2 \in \{0,1\} \\ \max\{d'_2,i_1\}=d_2}} A(c \times (i_1,d'_1) \times (i_2,d'_2), \kappa)$$

This formula is correct as a vertex sets $V(i)$ and $V(j)$ contain a dominating vertex if and only if they contained such a vertex before adding the edges. For the property of

domination, correctness follows because the vertex sets $V(i)$ and $V(j)$ are dominated if and only if they were either dominated before adding the edges, or if they become dominated by a vertex from the other vertex set because of the adding of the edges.

Join graphs: This operation joins two labelled graphs H_1 and H_2 with tables A_1 and A_2 to a labelled graph H with table A. To do this efficiently, we first apply state changes similar to those used in Chapters 11 and Chapter 12. We use states 0 and ? for the first attribute (inclusion) and states 1 and ? for the second attribute (domination).

Changing A_1 and A_2 to tables A_1^* and A_2^* that use this set of states can be done in a similar manner as in Chapters 11 and Chapter 12, i.e., by changing the states coordinate-wise as in for example Lemmas 11.9 and 12.5. We first copy A_y into A_y^*, for $y \in \{1, 2\}$ and then iteratively use the following formulas in a coordinate-wise manner:

$$\begin{aligned}
A_y^*(c_1 \times (0,1) \times c_2, \kappa) &= A_y^*(c_1 \times (0,1) \times c_2, \kappa) \\
A_y^*(c_1 \times (?,1) \times c_2, \kappa) &= A_y^*(c_1 \times (1,1) \times c_2, \kappa) + A_y^*(c_1 \times (0,1) \times c_2, \kappa) \\
A_y^*(c_1 \times (0,?) \times c_2, \kappa) &= A_y^*(c_1 \times (0,1) \times c_2, \kappa) + A_y^*(c_1 \times (0,0) \times c_2, \kappa) \\
A_y^*(c_1 \times (?,?) \times c_2, \kappa) &= A_y^*(c_1 \times (1,1) \times c_2, \kappa) + A_y^*(c_1 \times (1,0) \times c_2, \kappa) + \\
&\quad A_y^*(c_1 \times (0,1) \times c_2, \kappa) + A_y^*(c_1 \times (0,0) \times c_2, \kappa)
\end{aligned}$$

We have already seen many state changes similar to these in Chapters 11 and 12. Therefore, it is not surprising that we can now compute the table A^* in the following way, where the table A^* is the equivalent of the table A we want to compute only using the different set of states:

$$A^*(c, \kappa) = \sum_{\kappa_1 + \kappa_2 = \kappa} A_1^*(c, \kappa_1) \cdot A_2^*(c, \kappa_2)$$

Next, we apply state changes to obtain A from A^*. These state changes are the inverse of those given above. Again, first copy A^* into A and then iteratively transform the states in a coordinate-wise manner using the following formulas:

$$\begin{aligned}
A(c_1 \times (0,1) \times c_2, \kappa) &= A(c_1 \times (0,1) \times c_2, \kappa) \\
A(c_1 \times (1,1) \times c_2, \kappa) &= A(c_1 \times (?,1) \times c_2, \kappa) - A(c_1 \times (0,1) \times c_2, \kappa) \\
A(c_1 \times (0,0) \times c_2, \kappa) &= A(c_1 \times (0,?) \times c_2, \kappa) - A(c_1 \times (0,1) \times c_2, \kappa) \\
A(c_1 \times (1,0) \times c_2, \kappa) &= A(c_1 \times (?,?) \times c_2, \kappa) - A(c_1 \times (0,?) \times c_2, \kappa) \\
&\quad - A(c_1 \times (?,1) \times c_2, \kappa) + A(c_1 \times (0,1) \times c_2, \kappa)
\end{aligned}$$

Correctness of the computed table A follows by exactly the same reasoning as used in Theorem 11.7 and in Proposition 12.6. We note that the last of the above formulas is a nice example of an application of the principle of inclusion/exclusion: to find the number of sets corresponding to the $(1,0)$ -state, we take the number of sets corresponding to the $(?,?)$-state; then, we subtract what we counted to much, but because we subtract some sets twice, we need to add some number of sets again to obtain the required value.

The number of dominating sets in G of size κ can be computed from the table A related to the final operation of the k-expression for G. In this table, we consider only the entries in which the second property is 1, i.e., the entries corresponding to partial

solutions in which all vertices in G are dominated. Now, the number of dominating sets in G of size κ equals the sum over all entries $A(c, \kappa)$ with $c \in \{(0,1),(1,1)\}$.

For the running time, we observe that each of the $\mathcal{O}(n)$ join operations take $\mathcal{O}(n^2 4^k i_\times(n))$ time because we are multiplying n-bit numbers. Each of the $\mathcal{O}(nk^2)$ other operations take $\mathcal{O}(n^2 4^k)$ time since we need $\mathcal{O}(n4^k)$ series of a constant number of additions using n-bit numbers, and $i_+(n) = \mathcal{O}(n)$. The running time of $\mathcal{O}(n^3(k^2 + i_\times(n))\, 4^k)$ follows. □

Similar to the algorithms for Dominating Set on tree decompositions and branch decompositions in Chapters 11 and 12, we can improve the polynomial factors in the running time if we are interested only in the size of a minimum dominating set, or the number of these sets. To this end, we will introduce a notion of a de Fluiter property for cliquewidth. This notion is defined similarly to the de Fluiter property for treewidth; see Section 11.2.

Definition 13.3 (De Fluiter Property for Cliquewidth). Consider a method to represent the different partial solutions used in an algorithm that performs dynamic programming on clique decompositions (k-expressions) for an optimisation problem Π. Such a representation has the *de Fluiter property for cliquewidth* if the difference between the objective values of any two partial solutions of Π that are stored for a partially evaluated k-expression and that can both still lead to an optimal solution is at most $f(k)$, for some function f. Here, the function f depends only on the cliquewidth k.

The definition of the de Fluiter property for cliquewidth is very similar to the same notion for treewidth. However, the structure of a k-expression is different from tree decompositions and branch decompositions in such a way that the de Fluiter property for cliquewidth does not appear to be equivalent to the other two. This in contrast to the same notion for branchwidth that is equivalent to this notion for treewidth; see Section 12.1.3. The main difference is that k-expressions deal with sets of equivalent vertices instead of the vertices themselves.

The representation used in the algorithm for the Dominating Set problem above also has the de Fluiter property for cliquewidth.

Lemma 13.4. *The representation of partial solutions for the* Dominating Set *problem used in Theorem 13.2 has the de Fluiter property for cliquewidth with* $f(k) = 2k$.

Proof. Consider any partially constructed graph H from a partial bottom-up evaluation of the k-expression for a graph G, and let S be the set of vertices of the smallest remaining partial solution stored in the table for the subgraph H. We prove the lemma by showing that by adding at most $2k$ vertices to S, we can dominate all future neighbours of the vertices in H and all vertices in H that will receive future neighbours. We can restrict ourselves to adding vertices to S that dominate these vertices and not vertices in H that do not receive future neighbours, because Definition 13.3 considers only partial solutions on H that can still lead to an optimal solution on G. Namely, a vertex set $V(i)$ that contains undominated vertices that will not receive future neighbours when continuing the evaluation of the k-expression will not lead to an optimal

solution on G. This is because the selection of the vertices that will be in a dominating set happens only in the 'create a new graph' operations.

We now show that by adding at most k vertices to S, we can dominate all vertices in H, and by adding another set of at most k vertices to S, we can dominated all future neighbours of the vertices in H. To dominate all future neighbours of the vertices in H, we can pick one vertex from each set $V(i)$. Next, consider dominating the vertices in each of the vertex sets $V(i)$ and are not yet dominated and that will receive future neighbours. Since the 'add edges' operations of a k-expression can only add edges between future neighbours and all vertices with the label i, and since the 'relabel' operation can only merges the sets $V(i)$ and not split them, we can add a single vertex to S that is a future neighbour of a vertex in $V(i)$ to dominate all vertices in $V(i)$. □

Using this property, we can easily improve the result of Theorem 13.2 for the case where we want to count only the number of minimum dominating sets. This goes in a way similar to Corollaries 11.8, 11.10, 12.8, and 12.9.

Corollary 13.5. *There is an algorithm that, given a k-expression for a graph G, computes the number of minimum dominating sets in G in $\mathcal{O}(nk^2 4^k i_\times(n))$ time.*

Proof. For each colouring c, we maintain the size $B(c)$ of any minimum partial dominating set inducing c, and the number $A(c)$ of such sets. This can also be seen as a table $D(c)$ of tuples. Define a new function $\oplus$ such that

$$(A(c),B(c)) \oplus (A(c'),B(c')) \;=\; \begin{cases} (A(c)+A(c'),B(c)) & \text{if } B(c)=B(c') \\ (A(c^*),B(c^*)) & \text{otherwise} \end{cases}$$

where $c^* = \arg\min\{B(c),B(c')\}$. We will use this function to ensure that we count only dominating sets of minimum size.

We now modify the algorithm of Theorem 13.2 to use the tables D. For the first three operations, simply omit the size parameter κ from the formula and replace any $+$ by $\oplus$. For instance, the computation for the third operation that adds new edges connecting all vertices with label $V(i)$ to all vertices with label $V(j)$, becomes:

$$D(c\times\{(i_1,d_1)\}\times\{(i_2,d_2)\}) = \bigoplus_{\substack{d_1'\in\{0,1\}\\ \max\{d_1',i_2\}=d_1}} \bigoplus_{\substack{d_2'\in\{0,1\}\\ \min\{d_2',i_1\}=d_2}} D(c\times(i_1,d_1')\times(i_2,d_2'))$$

For the fourth operation, where we take the union of two labelled graphs, we need to be more careful. Here, we use that the given representation of partial solutions has the de Fluiter property for cliquewidth. We first discard solutions that contain vertices that are undominated and will not receive new neighbours in the future, that is, we set the corresponding table entries to $D(c)=(\infty,0)$. Then, we also discard any remaining solutions that are at least $2k$ larger than the minimum remaining solution.

Let $D_1(c)=(A_1(c),B_1(c))$ and $D_2(c)=(A_2(c),B_2(c))$ be the two resulting tables for the two labelled graphs H_1 and H_2 we need to join. To perform the join operation, we construct tables $A_1(c,\kappa)$ and $A_2(c,\kappa)$ as follows, with $y\in\{1,2\}$:

$$A_y(c,\kappa) = \begin{cases} A_y(c) & \text{if } B_y(c)=\kappa \\ 0 & \text{otherwise} \end{cases}$$

In these tables, κ has a range of size $2k$ and thus this table has size $\mathcal{O}(k\,4^k)$.

Now, we can apply the same algorithm for the join operations as described in Theorem 13.2. Afterwards, we retrieve the value of $D(c)$ by setting $A(c) = A(c, \kappa')$ and $B(c) = \kappa'$, where κ' is the smallest value of κ for which $A(c, \kappa)$ is non-zero.

For the running time, we observe that each of the $\mathcal{O}(k^2 n)$ operations that create a new graph, relabel vertex sets, or add edges to the graph compute $\mathcal{O}(4^k)$ tuples that cost $\mathcal{O}(i_+(n))$ time each since we use a constant number of additions and comparisons of an $\log(n)$-bits number and an n-bits number. Each of the $\mathcal{O}(n)$ join operations cost $\mathcal{O}(k^2 4^k i_\times(n))$ time because of the reduced table size. In total, this gives a running time of $\mathcal{O}(nk^2 4^k i_\times(n))$. □

Finally, we show that one can use $\mathcal{O}(k)$-bit numbers when considering the decision version of this minimisation problem instead of the counting variant.

Corollary 13.6. *There is an algorithm that, given a k-expression for a graph G, computes the size of a minimum dominating sets in G in $\mathcal{O}(nk^2 4^k)$ time.*

Proof. Maintain only the size $B(c)$ of any partial solution satisfying the requirements of the colouring c in the computations involved in any of the first three operations. Store this table by maintaining the size ξ of the smallest solution in B that has no undominated vertices that will not get future neighbours, and let B contain $\mathcal{O}(\log(k))$-bit numbers giving the difference in size between the size of the partial solutions and the number ξ; this is similar to, for example, Corollary 11.10.

For the fourth operation, follow the same algorithm as in Corollary 13.5, using $A(c, \kappa) = 1$ if $B(c) = \kappa$ and $A(c, \kappa) = 0$ otherwise. Since the total sum of all entries in this table is 4^k, the computations for the join operation can now be implemented using $\mathcal{O}(k)$-bit numbers. See also, Corollaries 11.10 and 12.9. In the computational model with $\mathcal{O}(k)$-bit word size that we use, the term in the running time for the arithmetic operations disappears since $i_\times(k) = \mathcal{O}(1)$. □

We conclude by noticing that $\mathcal{O}^*(4^k)$ algorithms for Independent Dominating Set and Total Dominating Set follow from the same approach. For Total Dominating Set, we have to change only the fact that a vertex does not dominate itself at the 'create new graph' operations. For Independent Dominating Set, we have to incorporate a check such that no two vertices in the solution set become neighbours in the 'add edges' operation.

13.3. Concluding Remarks

In this chapter, we have given $\mathcal{O}(4^k)$ algorithms for Dominating Set, Independent Dominating Set, and Total Dominating Set on graphs of cliquewidth at most k given with a k-expression. This improves previous results for these problems.

We conclude by noting that Dominating Set can be solved on rank decompositions of width k in $\mathcal{O}^*(2^{\frac{3}{4}k^2+\mathcal{O}(k)})$ time [63, 161] and on boolean decompositions of width k in $\mathcal{O}^*(8^k)$ time [62]. Both of these algorithms can be faster than our algorithm since the rankwidth and booleanwidth of a graph can be much smaller than its cliquewidth; see [62] and [248].

V

Conclusion

14 Conclusion

In this PhD thesis, we studied exact exponential-time algorithms for domination problems in graphs. This study led to faster exact exponential-time algorithms for many basic graph domination problems including Dominating Set, Independent Set, Edge Dominating Set, Total Dominating Set, Red-Blue Dominating Set, Partial Dominating Set, #Dominating Set, the parameterised problem k-Nonblocker, and many others. We also obtained faster algorithms for these and many other graph domination problems on some prominent types of graph decompositions: tree decompositions, branch decompositions, and, for some problems, clique decompositions (also called k-expressions).

A series of interesting new insights and techniques arose from this study. We mention the techniques of *inclusion/exclusion-based branching* and *extended inclusion/exclusion-based branching*. We also mention our generalisation of the *fast subset convolution* algorithm, which we translated to the setting of state-based dynamic programming algorithms on graph decompositions.

New insights often lead to new questions and new research directions. This is also the case for this study. Below, we give two open problems related to exact exponential-time algorithms for domination problems in graphs that we find very interesting. Furthermore, it is interesting to see to what extent the techniques presented in this thesis can be used to obtain faster algorithms for problems outside the context of domination problems in graphs. Two examples of this are our results for 2-Disjoint Connected Subgraphs in Chapter 10, and the $\mathcal{O}^*(c^k)$ algorithms for a large series of connectivity problems (including Hamiltonian Cycle, Steiner Tree, Feedback Vertex Set, Connected Dominating Set, and many others) on tree decompositions of width k that we obtained very recently [88].

We note that many suggestions for further research are also given in the concluding remarks sections of Chapters 4-13.

Beating the $\mathcal{O}^*(2^n)$-Barrier for Capacitated Vertex Cover. For every graph domination problem in $\mathcal{NP}$ for which vertex subsets are certificates (i.e., for which we can decide, for a given subset $D \subseteq V$, whether D is a solution to this problem in polynomial time) there exists a trivial $\mathcal{O}^*(2^n)$-time algorithm. Over the last years, algorithms have been developed beating these trivial algorithms for more and more complex problems.

For example, the first algorithms for Dominating Set beating this trivial time bound date from 2004 [151, 172, 279], while such algorithms for more complex problems such as Connected Dominating Set [143] (see also [1, 133]), Capacitated Dominating Set [90] (see also [225]), and the problems of computing the lower and upper irredundance numbers [22, 91] followed later. We note that we posted finding such algorithms for Capacitated Dominating Set and for the irredundance numbers of a graph as open problems in [40] and [148], respectively.

It is interesting to search for algorithms beating the trivial time bound for increasingly hard problems. Naturally, we can always impose more restrictions to a solution of a problem, as long as these restrictions are polynomial-time verifiable. In this context, we find the Capacitated Vertex Cover problem an interesting next problem to consider.

Capacitated Vertex Cover
Input: A graph $G = (V, E)$, a capacity function $c : V \to \mathbb{N}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a capacitated vertex cover $C \subseteq V$ in G of size at most k with the capacities c?

Given a graph G and a capacity function $c : V \to \mathbb{N}$, a *capacitated vertex cover* is a vertex subset $C \subseteq V$ such that there exist a function $f : E \to C$ that assigns to each edge $e \in E$ the vertex $v \in e$ that covers it; this in such a way that there exist at most $c(v)$ vertices e with $f(e) = v$.

Open Problem 14.1. *Give an algorithm for* Capacitated Vertex Cover *that runs in $\mathcal{O}((2-\epsilon)^n)$ time, for some $\epsilon > 0$.*

Of course, it is possible that such an algorithm does not exists. However, proving such a result (under any reasonable complexity-theoretic hypothesis) seems to be very hard.

Bounds on the Treewidth of Sparse Graphs. Many of the results in Chapters 8-10 depend on the upper bound on the pathwidth of cubic graphs given by Fomin and Høie [147] (Theorem 2.14) and the extension of this result to other low-degree graphs [138] (Proposition 2.16). Any (significantly enough) improvement of this upper bound would directly improve many of these results, possibly even leading to faster algorithms for Dominating Set and some closely related problems than those given in Chapter 5.

This upper bound of $\frac{1}{6}n$ (plus polylogarithmic factors that we will ignore here) on the pathwidth of a cubic graph also is the best known upper bound on the treewidth of a cubic graph. This is so even while a tree decomposition is a more general structure than a path decomposition. Due to our results in Chapter 11, the exponential parts of the running times of dynamic programming algorithms on path decompositions and tree decompositions are equal, for many problems. As a result, any improvement of the upper bound on the treewidth of cubic graphs would probably have almost

the same consequences for exact exponential-time algorithms. We note that such an improvement would also have effects on a series of results in parameterised algorithms: see Theorem 6.18 or, for example, [97, 138].

Open Problem 14.2. *For some constant $c_t < \frac{1}{6}$, prove that the treewidth of a cubic graph is at most $c_t \cdot n + o(n)$, and that tree decompositions of this width can be computed in polynomial time.*

We note that the upper bound on the pathwidth of a cubic graph given by Fomin and Høie [147] is not known to be tight. The best known lower bound on the pathwidth of cubic graphs is $0.082n$ [149, notes on chapter 5]; this bound can be derived from a similar lower bound on the bisection width of cubic graphs [21]. Also, no similar lower bounds on the treewidth of cubic graphs are known.

Appendix

A Omitted Case Analyses

A.1. The Faster Algorithm for Partition Into Triangles on Graphs of Maximum Degree Four

We have given an $\mathcal{O}(1.02445^n)$-time algorithm for Partition Into Triangles on graphs of maximum degree four in Chapter 4. We have also claimed a faster exponential-time algorithm that solves this problem in $\mathcal{O}(1.02220^n)$ time. In this section of the appendix, we will give the details of this faster algorithm.

In Chapter 4, we have used known algorithms for Exact Satisfiability and Exact 3-Satisfiability to solve Partition Into Triangles on graphs of maximum degree four. In this section, we present an algorithm for Exact Satisfiability that is specifically tailored to the fact that the input is obtained from an instance of Partition Into Triangles. This algorithm will be analysed using the number of vertices in a Partition Into Triangles instance used to build the different structures involved in an Exact 3-Satisfiability instance as a measure of instance size.

Our algorithm is a branch-and-reduce algorithm for Exact Satisfiability instances (not only Exact 3-Satisfiability instances). We analyse the resulting algorithm by bounding the number of subproblems generated. To do so, we will use the following measure of progress k on instances with variable set X and clause set $\mathcal{C}$.

$$k := 5|X| + \sum_{C \in \mathcal{C}, |C| \geq 3} 2\frac{1}{3}\left(|C| - 3\right)$$

Before justifying this measure, we introduce a series of standard reduction rules used in many algorithms for Exact Satisfiability. Besides using these reduction rules, we always decide that we have a No-instance if two or more variables in a clause are set to $True$. Also, we set any literal to $False$ that occurs in a clause with a literal that has been set to $True$, and thereafter we remove the clause. After doing so, we

remove all literals that have been set to $False$ from the remaining clauses. If this results in an empty clause, we decide that we have a No-instance.

Below, we let x and y be arbitrary literals, we let C and C' be arbitrary (sub)clauses, and we let Φ be the rest of the current Exact Satisfiability formula. By $\Phi : a \to b$, we denote the formula Φ with all occurrences of the literal a replaced by b and all occurrences of the literal $\neg a$ by $\neg b$. This notation is extended to sets of variables, for example in $\Phi : C \to False$. The numbers behind the reduction rules represent the minimum decrease of the measure as a result of the reduction.

1.	$C \wedge C \wedge \Phi$	$\Rightarrow$	$C \wedge \Phi$	(0)
2.	$(x) \wedge \Phi$	$\Rightarrow$	$\Phi : x \to True$	(-5)
3.	$(x, y) \wedge \Phi$	$\Rightarrow$	$\Phi : y \to \neg x$	(-5)
4.	$(x, x, C) \wedge \Phi$	$\Rightarrow$	$C \wedge \Phi : x \to False$	(-5)
5.	$(x, \neg x, C) \wedge \Phi$	$\Rightarrow$	$\Phi : C \to False$	(-5)
6.	$(x, y, C) \wedge (x, \neg y, C') \wedge \Phi$	$\Rightarrow$	$(y, C) \wedge (\neg y, C) \wedge \Phi : x \to False$	(-5)
7.	$(x, y, C) \wedge (\neg x, \neg y, C') \wedge \Phi$	$\Rightarrow$	$\Phi : y \to \neg x;\ C, C' \to False$	(-5)
8.	$C \wedge C' \wedge \Phi$ with $C \subset C'$	$\Rightarrow$	$C \wedge \Phi : (C' \setminus C) \to False$	(-5)
9.	$(x, C) \wedge (y, C) \wedge \Phi$	$\Rightarrow$	$(x, C) \wedge \Phi : y \to x$	(-5)
10.	$(x, C) \wedge (C, C') \wedge \Phi$ with $\lvert C\rvert, \lvert C'\rvert \geq 2$	$\Rightarrow$	$(x, C) \wedge (\neg x, C') \wedge \Phi$	$(-2\frac{1}{3})$
11.	$(x, C) \wedge (\neg x, C') \wedge \Phi$ with $x, \neg x \notin \bigcup \Phi$	$\Rightarrow$	$(C, C') \wedge \Phi$	$(-2\frac{2}{3})$
12.	If, after application of Reduction Rules 1-11, Φ contains a variable x and a series of variables $y_1, \ldots, y_l$ that occur only in clauses with x in a such way that every clause that contains x contains exactly one of the variables y_i, then set x to $False$.			(-5)

Lemma A.1. *Reduction Rules 1-12 are correct and result in the given minimum reductions in the measure k.*

Proof. Reduction Rules 1-11 are used in many papers on Exact Satisfiability, e.g. see [67]; their correctness is evident. For the correctness of Reduction Rule 12, consider a variable x and a series of variables $y_1, \ldots, y_l$ as in the statement of the reduction rule. Since Reduction Rules 6 and 7 do not apply, the signs of all literals of x must be equal, and the same goes for the signs of the literals of each of the individual variables y_i. Without loss of generality, we assume all these literals to be positive. Consider any solution with x set to $True$. Since the y_i occur in clauses with x, they must all be set to $False$. Because none of the y_i occur in clauses together or in a clause without x, we can replace this assignment by an equivalent one by setting x to $False$ and all the y_i to $True$. Correctness of the reduction rule follows.

Now, consider the decrease of the measure. When any of the above reduction rules except Reduction Rules 1, 10, and 11 are applied, at least one variable is assigned a value or replaced by another, and no clauses are increased in size. Hence, the measure decreases by at least 5 in these cases. Clearly, Reduction Rule 1 does not increase the measure as it removes a clause. Reduction Rule 10 reduces the size of one clause of size at least four by one, hence the measure decreases by $2\frac{1}{3}$. Finally, Reduction Rule 11 removes one variable and one possibly large clause of size s decreasing the measure by $5 + 2\frac{1}{3}(s - 3)$. However, this reduction rule also increases the size of another clause by $s - 2$ increasing the measure by $2\frac{1}{3}(s - 2)$. Together, this leads to a decrease of $5 - 2\frac{1}{3} = 2\frac{2}{3}$. □

We now justify our measure. Recall that $f(x)$ denotes the frequency of the varia-

ble x, that $f_+(x)$ and $f_-(x)$ denote the frequencies of the positive and negative literals of x, respectively, and that $F(x) = (f_+(x), f_-(x))$.

Lemma A.2. *Let G be an n-vertex graph of maximum degree four. In polynomial time, we can either decide that G is a* No*-instance of* Partition Into Triangles*, or transform G into an equivalent* Exact Satisfiability *instance of measure k that satisfies $k \leq n$.*

Proof. We first apply the procedure used to prove Theorem 4.9 to G, see Section 4.3. If this procedure does not decide that we have a No-instance, then it results in an equivalent Exact 3-Satisfiability instance satisfying $2|\mathcal{C}| + \sum_{x \in X} (2f(x) - 3) \leq n$, where X is the set of variables and $\mathcal{C}$ is the set of clauses.

We distinguish between two types of variables $x \in X$: variables with $f(x) = 2$ and $F(x) = (1, 1)$, and all other variables, which by Property 4.6 satisfy $f(x) \geq 3$. Let n_2 be the number of variables with $f(x) = 2$, and let $n_{\geq 3}$ be the number of other variables. Then:

$$n \geq 2|C| + \sum_{x \in X} (2f(x) - 3) \geq 2|C| + n_2 + 3n_{\geq 3} = 5n_{\geq 3} + 2\frac{1}{2}n_2$$

The last equality follows from distributing the two vertices used by the clauses of size three to the variables: these variables are given $\frac{2}{3}$ vertex for each occurrence in $\mathcal{C}$.

To the obtained instance, we exhaustively apply Reduction Rules 1-12. We note that this will not result in an instance of Exact 3-Satisfiability, but in an instance of Exact Satisfiability instead. This is because the reduction rules (specifically Reduction Rule 11) can create clauses of size at least four when removing variables x with $f(x) = 2$. Since a clause can increase by at most one in size per removed variable, we obtain the following inequality:

$$n \geq 5n_{\geq 3} + 2\frac{1}{2}n_2 \geq 5n_{\geq 3} + \sum_{C \in \mathcal{C}, |C| \geq 3} 2\frac{1}{3}(|C| - 3) = k$$

This proves that $k \leq n$. □

We note that the amounts proven in Lemma A.1 by which the measure decreases as a result of applying a reduction rule apply only after first applying Lemma A.2: Lemma A.2 uses any such decrease due to Reduction Rule 11 for its correctness. We also note that the new Exact Satisfiability instance no longer satisfies Property 4.6. Before, this property held only if we counted identical clauses multiple times; now, we remove these double clauses.

Reduction Rules 1-12 enforce some new constraints on the resulting Exact Satisfiability instances. These will be proven in the next lemma. Recall that a *unique variable* is a variable x with $f(x) = 1$.

Lemma A.3. *After exhaustively applying Reduction Rules 1-12 to an* Exact Satisfiability *instance, it satisfies the following properties:*

1. *All clauses have size at least three.*
2. *All variables occur at most once in each clause.*

3. *If variables occur together in multiple clauses, their literals have identical signs in all clauses in which they occur together.*
4. *For any pair of clauses, each clause contains at least two variables not occurring in the other.*
5. *There are no variables x with $F(x) = (1, 1)$.*
6. *Every clause contains at most one unique variable.*

Proof. (1.) Smaller clauses are removed by Reduction Rules 2 and 3. (2.) Reduction Rule 4 or 5 applies if a clause contains a variable two or more times. (3.) If their literals do not have the same signs, Reduction Rule 6 or 7 applies. (4.) No clauses are identical by Reduction Rule 1. No clause is a subclause of another clause by Reduction Rule 8. And, if a clause contains only one variable that does not occur in the other clause, Reduction Rule 9 or 10 applies. (5.) By Reduction Rule 11. (6.) If a clause has more than one unique variable, Reduction Rule 12 applies. □

Next, we give a series of lemmas that describe the branching rules of our algorithm. Since we first exhaustively apply the reduction rules before branching, each lemma will assume that no reduction rule applies without mentioning this. Also, we implicitly assume that directly after the branching all reduction rules are exhaustively applied again. In each lemma, we prove that the described branching has associated branching number at most 1.02220 when analysed using the measure k (see Section 2.1).

Lemma A.4. *If an* Exact Satisfiability *instance contains a variable x that occurs both as a positive and as a negative literal, then we can either reduce the instance to an equivalent smaller instance, or we can branch on the instance such that the associated branching number is at most* 1.02220.

Proof. Let us first consider branching on a variable x with $f_+(x) \geq 2$ and $f_-(x) \geq 2$, i.e., we have the following situation:

$$(x, C_1) \wedge (x, C_2) \wedge (\neg x, C_1') \wedge (\neg x, C_2') \wedge \Phi$$

where x can also occur in Φ.

We branch by considering two subproblems: one where we set x to $True$ and one where we set x to $False$. Below, we consider a series of cases where we distinguish between whether the C_i or C_i' have size two or size at least three.

If we set x to $True$, then the measure decreases by at least the following quantities. We give these quantities by a series of bullets. Below, we will compute the corresponding sums of the decrease of the measure for each of the cases considered.

- 5 for removing x.
- 5 for each literal in C_1 or C_2 because these are set to $False$. Note that by Lemma A.3(4), at least two variables occur in C_1 that do not occur in C_2 and vice versa. Therefore, then the measure decreases by 20 if $|C_1| = |C_2| = 2$, and by at least 25 otherwise.
- 5 per C_i' with $|C_i'| = 2$ because $\neg x$ is removed from the corresponding clauses: this results in the removal of at least one more variable by Reduction Rule 3. Notice that by Lemma A.3(3): $(C_1 \cup C_2) \cap (C_1' \cup C_2') = \emptyset$.
- A number of times $2\frac{1}{3}$ for reducing the sizes of the clauses.

The situation is symmetric, hence setting x to $False$ decreases the measure by the same quantities after replacing C_i by C'_i and vice versa.

The table below gives all considered cases together with the minimum decrease of the measure obtained by each of the above reasons. In the first two columns, we give the number of C_i and C'_i with $|C_i| \geq 3$ and $|C'_i| \geq 3$, respectively. We assume that all other C_i and C'_i have size two. In the third and fourth column, we give the decrease of the measure as a sum of four terms: the first one corresponds to the first bullet given above, the second corresponds to the second bullet, etc. In the last column, we give the branching number τ associated with the branching. By symmetry reasons, we can restrict ourselves to the given cases.

$\#C_i$:	$\#C'_i$:	decrease of the measure k when we set		
$\lvert C_i\rvert \geq 3$	$\lvert C'_i\rvert \geq 3$	$x \to True$	$x \to False$	τ
0	0	$5+20+10+0=35$	$5+20+10+0=35$	1.02001
1	0	$5+25+10+2\frac{1}{3}=42\frac{1}{3}$	$5+20+5+2\frac{1}{3}=32\frac{1}{3}$	1.01886
2	0	$5+25+10+4\frac{2}{3}=44\frac{2}{3}$	$5+20+0+4\frac{2}{3}=29\frac{2}{3}$	1.01910
1	1	$5+25+5+4\frac{2}{3}=39\frac{2}{3}$	$5+25+5+4\frac{2}{3}=39\frac{2}{3}$	1.01763
2	1	$5+25+5+7=42$	$5+25+0+7=37$	1.01773
2	2	$5+25+0+9\frac{1}{3}=39\frac{1}{3}$	$5+25+0+9\frac{1}{3}=39\frac{1}{3}$	1.01778

This proves the branching numbers for branching on variables x with $f_+(x) \geq 2$ and $f_-(x) \geq 2$. Hence, we can assume without loss of generality by negating variables that, for each variable x, we have have $f_-(x) \in \{0,1\}$ and $f_+(x) \geq 1$.

If $f_+(x) \geq 3$, we can make a similar table associated with the following situation:

$$(x, C_1) \wedge (x, C_2) \wedge (x, C_3) \wedge (\neg x, C) \wedge \Phi$$

Again, the first two columns give the size of $|C|$ and the $|C_i|$; the third and the fourth column contain the decrease of the measure in both branches as a sum of the quantities based on each of the four bullets given above; and the fifth column gives the associated branching number. In the sum corresponding to the branch where we set $x \to True$, we bound the decrease of the measure due to assigning values to variables with literals in C_1, C_2, and C_3 by 30 if $|C_1| = |C_2| = |C_3| = 2$, and by 35 otherwise.

	$\#C_i$:	decrease of the measure k when we set		
$\lvert C\rvert$	$\lvert C_i\rvert \geq 3$	$x \to True$	$x \to False$	τ
2	0	$5+30+5+0=40$	$5+10+15+0=30$	1.02015
2	1	$5+35+5+2\frac{1}{3}=47\frac{1}{3}$	$5+10+10+2\frac{1}{3}=27\frac{1}{3}$	1.01924
2	2	$5+35+5+4\frac{2}{3}=49\frac{2}{3}$	$5+10+5+4\frac{2}{3}=24\frac{2}{3}$	1.01963
2	3	$5+35+5+7=52$	$5+10+0+7=22$	1.02013
≥ 3	0	$5+30+0+2\frac{1}{3}=37\frac{1}{3}$	$5+15+15+2\frac{1}{3}=37\frac{1}{3}$	1.01874
≥ 3	1	$5+35+0+4\frac{2}{3}=44\frac{2}{3}$	$5+15+10+4\frac{2}{3}=34\frac{2}{3}$	1.01773
≥ 3	2	$5+35+0+7=47$	$5+15+5+7=32$	1.01794
≥ 3	3	$5+35+0+9\frac{1}{3}=49\frac{1}{3}$	$5+15+0+9\frac{1}{3}=29\frac{1}{3}$	1.01820

This proves the branching numbers for branching on variables x with $f_+(x) \geq 3$ and $f_-(x) \geq 1$. Because variables x with $F(x) = (1,1)$ are removed by the reductions rules (Lemma A.3(5)), the only remaining variables x for which we have to prove the lemma are those with $F(x) = (2,1)$. Let x be such a variable with $F(x) = (2,1)$.

If the negated literal of x occurs in a clause of size three, we apply the following transformation:

$$(x, C_1) \wedge (x, C_2) \wedge (\neg x, v_1, v_2) \wedge \Phi \quad \Longrightarrow \quad (v_1, v_2, C_1) \wedge (v_1, v_2, C_2) \wedge \Phi$$

This transformation is well-known as resolution; see for example [67]. In this transformation, we remove one variable, but increase two clauses in size by one. Therefore, this transformation does not increase the measure: it decreases by $5 - 2 \times 2\frac{1}{3} = \frac{1}{3}$.

If the negated literal of x occurs in a clause of size at least four, this corresponds to the following situation with $|C| \geq 3$.

$$(x, C_1) \wedge (x, C_2) \wedge (\neg x, C) \wedge \Phi$$

In this case, we again branch by considering either setting $x \to True$ in one branch and setting $x \to False$ in the other branch. We again consider a number of subcases corresponding to the C_i having size two or at least three. The associated branching numbers are again computed in a table similar to the two tables given above.

		decrease of the measure k when we set		
$\|C_1\|$	$\|C_2\|$	$x \to True$	$x \to False$	τ
2	2	$5 + 20 + 0 + 2\frac{1}{3} = 27\frac{1}{3}$	$5 + 15 + 10 + 2\frac{1}{3} = 32\frac{1}{3}$	1.02357
2	≥ 3	$5 + 25 + 0 + 4\frac{2}{3} = 34\frac{2}{3}$	$5 + 15 + 5 + 4\frac{2}{3} = 29\frac{2}{3}$	1.02183
≥ 3	≥ 3	$5 + 25 + 0 + 7 = 37$	$5 + 15 + 0 + 7 = 27$	1.02209

At this point, each case except one gives a branching number that is smaller than the claimed 1.02220. So, to obtain our result, we must analyse this case in more detail: this is the case where the variable x has $F(x) = (2, 1)$ and where $|C_1| = |C_2| = 2$ in the above situation. A refined analysis of the decrease of the measure give the result.

Let us inspect this one case a little more thoroughly. This case corresponds to the following situation:

$$(x, v_1, v_2) \wedge (x, v_3, v_4) \wedge (\neg x, w_1, w_2, w_3) \wedge \Phi$$

To obtain a branching number that improves upon the one given in the above table, we look at the effect of the branching on Φ. Consider setting x to $True$ and hence the v_i to $False$. Notice that at least two of the variables v_i must also occur somewhere in Φ by Lemma A.3(6).

Let us first assume that a literal $\neg v_i$ occurs in Φ, and without loss of generality let this be $\neg v_1$. Consider the clause with $\neg v_1$. By Lemma A.3(4), this clause cannot contain a literal of v_2, and it must contain at least two literals that are not literals of the variables v_3 and v_4. Hence, this clause must contain at least one variable that we have not considered this far. The literal of this variable will be set to $False$ decreasing the measure by at least an additional 5.

If no literals of the form $\neg v_i$ occur in Φ, at least two positive literals of the v_i must occur in Φ; these literals are set to $False$. We now consider several cases with a clause containing these literals depending on the number of literals in the clause that are not among the v_i. By Lemma A.3(4), each clause in Φ can contain at most two occurrences of the v_is and thus must contain at least one literal of a different variable. If these literals fill a clause except for one spot, as in (v_1, v_3, y), then y is set to $True$

decreasing the measure by at least an additional 5. If these literals fill a clause except for two spots, as in (v_1, v_3, y_1, y_2), then Reduction Rule 3 will replace y_2 by $\neg y_1$ also decreasing the measure by an additional 5. And, if these literals fill a clause except for at least three spots, then each such literal will be removed decreasing the measure an additional by $2\frac{1}{3}$ each.

Altogether, we conclude that with at least two v_i in Φ, this decreases the measure by at least an additional $4\frac{2}{3}$. Therefore, we obtain a branching number of $\tau(27\frac{1}{3} + 4\frac{2}{3}, 32\frac{1}{3}) = \tau(32, 32\frac{1}{3}) < 1.02179$. □

We notice that we will use systematic case analyses as in the proof of the above lemma throughout the rest of this appendix. In these analyses, we will often start with a series of bullets corresponding to the different effects that decrease the measure. Then, for each case considered, we will give the associated decrease of the measure associated with each bullet systematically. Thereafter, we will perform the case analysis by giving a table with a row for each case giving the relevant properties of this case, the total decrease of the measure in each branch as a sum of the effects of each bullet in the enumeration given before, and the associated branching number.

From now on, we assume without loss of generality that all variables occur only as positive literals. Based on this assumption, we can give a simple lower bound on the decrease of the measure as a result of setting a number of literals in Φ to $False$. This is formalised in the following proposition. Its proof has similarities to the last few paragraphs of the proof of Lemma A.4.

Proposition A.5. *Let Φ be an* Exact Satisfiability *formula containing only positive literals. Consider setting some variables with a total of l literals in Φ to $False$, while at least one variable in Φ remains without a truth assignment. Then, setting the literals to $False$ decreases the measure of Φ by at least $2\frac{1}{3} \times l$ if $l \leq 2$ and at least 5 if $l \geq 3$ besides the decrease due to removing the corresponding variables.*

Proof. If Φ contains a clause in which all literals are set to $False$, then Φ is not satisfiable resulting in the removal of the whole formula. If Φ contains a clause in which all literals except for one are set to $False$, then the last literals is set to $True$ removing a variable and thus decreasing the measure by at least 5. If Φ contains a clause in which all literals except for two are set to $False$, then the variables corresponding to the last two literals will be replaced by one variable by Reduction Rule 3 decreasing the measure by at least 5. Finally, if Φ contains a clause in which a literal is set to $False$ and in which at least three literals are not assigned a value, then this reduces the size of the clause decreasing the measure by at least $2\frac{1}{3}$ each.

We conclude that the minimum decrease of the measure is $\min\{2\frac{1}{3} \times l,\ 5\}$. This proves the proposition. □

Lemma A.6. *If an* Exact Satisfiability *instance contains two clauses that have two or more variables in common, then we can either reduce the instance to an equivalent smaller instance, or we can branch on the instance such that the associated branching number is at most* 1.02220.

Proof. In the proof below, we can assume that all literals are positive literals since we can otherwise apply Lemma A.4.

Let C be the set of literals contained in both clauses, and let C_1 and C_2 be the literals in each clause not contained in the other. We have the following situation:

$$(C, C_1) \wedge (C, C_2) \wedge \Phi$$

with $|C| \geq 2$ as in the statement of the lemma, and $|C_1|, |C_2| \geq 2$ by Lemma A.3(4).

In most cases, we will branch in the following way. In one subproblem, we assume that a literal in C will be $True$; consequently, we set all variables in C_1 and C_2 to $False$. In the other subproblem, we assume that none of the literals in C will be $True$; we set the corresponding variables to $False$. We will distinguish different cases where C, C_1 and C_2 have size two or size at least three.

In the first subproblem where the literals in C_1 and C_2 are set to $False$, this leads to the following decrease of the measure k:

- 10 per C_i with $|C_i| = 2$ and 15 per C_i with $|C_i| \geq 3$ for removing the variables that are set to $False$. This is correct since all variables occur at most once per clause by Lemma A.3(2).
- 5 if $|C| = 2$ because then Reduction Rule 3 will remove one additional variable.
- a number of times $2\frac{1}{3}$ depending on the size of C, C_1 and C_2 for reducing the size of the two given clauses.
- $4\frac{2}{3}$ if $|C_1| = |C_2| = 2$ and 5 otherwise for the additional decrease of the measure of Φ. By Lemma A.3(6), at least two literals in Φ are set to $False$ if $|C_1| = |C_2| = 2$ and at least three literals otherwise. The given quantities by which the measure decreases correspond to the ones proven in Proposition A.5.

In the second subproblem where the literals in C are set to $False$, this leads to the following decrease of the measure k:

- 10 if $|C| = 2$ and 15 if $|C| \geq 3$ for removing the variables that are set to $False$.
- 5 for each C_i with $|C_i| = 2$ because Reduction Rule 3 will remove additional variables in these cases.
- a number of times $2\frac{1}{3}$ depending on the size of C, C_1 and C_2 for reducing the size of the two clauses.
- a quantity bounding the additional decrease of the measure of Φ from below. If $|C| = 2$, we use $2\frac{1}{3}$ because by Lemma A.3(6) one of the variables in C must occur in Φ; this leads to the given decrease by Proposition A.5. If $|C| \geq 3$, we use $4\frac{2}{3}$ by the same reasoning.

Next, we compute the branching numbers associated with the given branching by giving a table that is similar to the tables used in the proof of Lemma A.4.

	$\#C_i$:	decrease of the measure k when we set		
$\|C\|$	$\|C_i\| \geq 3$	$C_1, C_2 \to False$	$C \to False$	τ
2	0	$20 + 5 + 4\frac{2}{3} + 4\frac{2}{3} = 34\frac{1}{3}$	$10 + 10 + 4\frac{2}{3} + 2\frac{1}{3} = 27$	1.02298
2	1	$25 + 5 + 7 + 5 = 42$	$10 + 5 + 7 + 2\frac{1}{3} = 24\frac{1}{3}$	1.02167
2	2	$30 + 5 + 9\frac{1}{3} + 5 = 49\frac{1}{3}$	$10 + 0 + 9\frac{1}{3} + 2\frac{1}{3} = 21\frac{2}{3}$	1.02088
≥ 3	0	$20 + 0 + 9\frac{1}{3} + 4\frac{2}{3} = 34$	$15 + 10 + 9\frac{1}{3} + 4\frac{2}{3} = 39$	1.01921
≥ 3	1	$25 + 0 + 11\frac{2}{3} + 5 = 41\frac{2}{3}$	$15 + 5 + 11\frac{2}{3} + 4\frac{2}{3} = 36\frac{1}{3}$	1.01797
≥ 3	2	$30 + 0 + 14 + 5 = 49$	$15 + 0 + 14 + 4\frac{2}{3} = 33\frac{2}{3}$	1.01712

Hence, we have proven the lemma for all cases except when $|C| = |C_1| = |C_2| = 2$. In this case, we have the following situation:

$$(x, y, v_1, v_2) \wedge (x, y, v_3, v_4) \wedge \Phi$$

If, in the branch where we set $v_1, v_2, v_3, v_4 \to False$, the additional decrease of the measure of Φ is at least 7, then we obtain the required branching number since $\tau(20 + 5 + 4\frac{2}{3} + 7, 27) = \tau(36\frac{2}{3}, 27) < 1.02220$.

By Lemma A.3(6), at least two occurrences of the literals of v_1, v_2, v_3, and v_4 must occur in Φ. If these are exactly two occurrences, then both x and y must occur at least once in Φ also, as Reduction Rule 12 would otherwise be applicable. In this case, the additional decrease of the measure of Φ in the branch where we set $x, y \to False$ can be bounded from below by $4\frac{2}{3}$ in stead of $2\frac{1}{3}$ by Proposition A.5. Thus, we obtain a branching number of $\tau(34\frac{1}{3}, 10 + 10 + 4\frac{2}{3} + 4\frac{2}{3}) = \tau(34\frac{1}{3}, 29\frac{1}{3}) < 1.02207$ for this case.

If there are at least three occurrences of the literals of v_1, v_2, v_3, and v_4 in Φ while setting them to $False$ decreases the measure by less than 7, then all of these occurrences of the v_i must be in clauses of size three with exactly one other variable z, as for example in: $(v_1, v_3, z) \wedge (v_2, v_4, z)$. This holds because all literals occur only as positive literals, and all other configurations that do not directly give a No-instance lead to an additional decrease of the measure of Φ of at least 7: in these cases, at least three clauses of size at least four will be reduced in size ($3 \times 2\frac{1}{3} = 7$); at least two variables will be removed ($2 \times 5 > 7$); or exactly one variable will be removed and at least one clause of size at least four is reduced in size ($5 + 2\frac{1}{3} > 7$).

In fact, the only remaining situation is the following:

$$(x, y, v_1, v_2) \wedge (x, y, v_3, v_4) \wedge (v_1, v_3, z) \wedge (v_2, v_4, z) \wedge \Phi$$

This holds because if z exist in a clause of size three with any of the v_i, then z will not occur in a clause of size three with the same v_i again due to Lemma A.3(4). Hence, in order to put at least three of the literals of the variables v_i in clauses of size three with no other variables than z, exactly one occurrence of each of the four v_is is necessary.

In this special case, we branch z instead. If we set $z \to True$, this results in v_1, v_2, v_3, and v_4 being set to $False$, and in the replacement of y by $\neg z$ by Reduction Rule 3. Thus, this removes a total of 6 variables and 2 clauses of size four decreasing the measure by at least $6 \times 5 + 2 \times 2\frac{1}{3} = 34\frac{2}{3}$. If we set $z \to False$, this directly results in the following replacements: $v_3 \to \neg v_1$ and $v_4 \to \neg v_2$. In the two clauses with x and y this leads to the following situation $(x, y, v_1, v_2) \wedge (x, y, \neg v_1, \neg v_2)$. This situation is reduced by Reduction Rule 7 by setting x and y to $False$. In this branch, a total of 5 variables and 2 clauses of size four are removed decreasing the measure by at least $5 \times 5 + 2 \times 2\frac{1}{3} = 29\frac{2}{3}$. The associated branching number equals $\tau(34\frac{2}{3}, 29\frac{2}{3}) < 1.02183$, completing the proof of the lemma. □

We deal with variables of relatively high frequency next: variables x with $f(x) \geq 4$.

Lemma A.7. *If an* Exact Satisfiability *instance contains a variable x with $f(x) \geq 4$, then we can either reduce the instance to an equivalent smaller instance, or we can branch on the instance such that the associated branching number is at most* 1.02220.

Proof. We can assume that Lemmas A.4 and A.6 do not apply, otherwise we are done. Therefore, each variable has only positive literals and no two variables occur together in a clause more than once. This means that we have the following situation:

$$(x, C_1) \wedge (x, C_2) \wedge (x, C_3) \wedge (x, C_4) \wedge \Phi$$

where x can also occur in Φ.

We branch on x and again distinguish several cases based on the sizes of the C_i. If we set $x \to True$, we obtain the following quantities for the decrease in the measure:

- 5 for removing x.
- $5 \times \sum_{i=1}^{4} |C_i|$ for removing the variables in the C_i; these are set to $False$.
- a number of times $2\frac{1}{3}$ for reducing the clauses.
- 5 extra since by Lemma A.3(6) at least 4 variables must also occur in Φ and this leads to an additional decrease of the measure of at least 5 by Proposition A.5.

If we set $y \to False$, we obtain the following quantities for the decrease in the measure:

- 5 for removing x.
- 5 for each C_i with $|C_i| = 2$ because Reduction Rule 3 will remove an additional variable in these cases.
- a number of times $2\frac{1}{3}$ for reducing the sizes of the clauses.

Identical to the proofs of the previous lemmas, we calculate the branching numbers for each considered case in a table. In this table, we compute the decrease of the measure in each branch as a sum of the above bullets.

$\#C_i$:	decrease of the measure k when we set		
$\lvert C_i\rvert \geq 3$	$x \to True$	$x \to False$	τ
0	$5+40+0+5=50$	$5+20+0=25$	1.01944
1	$5+45+2\frac{1}{3}+5=57\frac{1}{3}$	$5+15+2\frac{1}{3}=22\frac{1}{3}$	1.01891
2	$5+50+4\frac{2}{3}+5=64\frac{2}{3}$	$5+10+4\frac{2}{3}=19\frac{2}{3}$	1.01859
3	$5+55+7+5=72$	$5+5+7=17$	1.01849
4	$5+60+9\frac{1}{3}+5=79\frac{1}{3}$	$5+0+9\frac{1}{3}=14\frac{1}{3}$	1.01859

This completes the proof. □

What remains is to deal with variables x with $f(x) = 3$. Hereafter, only variables x with $F(x) = (1, 0)$ and $F(x) = (2, 0)$ remain. In this case, the problem solvable in polynomial time as noted in many earlier papers on Exact Satisfiability; see for example [67].

Before giving the last lemmas that deal with the branching of the algorithm, we first introduce a new proposition dealing with the additional decrease of the measure due to setting a number of literals in Φ to $False$ under some extra conditions: this will improve upon Proposition A.5 when these conditions apply. Hereafter, we introduce a new reduction rule that will make sure that these extra conditions apply when needed.

Proposition A.8. *Let Φ be an* Exact Satisfiability *formula containing only positive literals. Consider setting some variables with a total of l literals in Φ to $False$, while at least three variables in Φ remain without a truth assignment. Then, setting the literals to $False$ decreases the measure of Φ by at least the following quantities besides the decrease due to removing the corresponding variables.*

1. $\min\{2\frac{2}{3} \times l,\ 15\}$ *if no variables exist in* Φ *that, in at least two clauses, occur only with literals that have been set to* $False$.
2. $\min\{5\lfloor l/4 \rfloor + 2\frac{1}{3}(l \bmod 4),\ 15\}$ *if no variables exist in* Φ *that, in at least three clauses, occur only with literals that have been set to* $False$.

Proof. We start with the first situation where no variables in Φ exist that, in two or more clauses, occur only with literals that have been set to $False$. If any of the l literals that are set to $False$ occur in a clause of size at least four in Φ, then this removes one literal decreasing the measure by $2\frac{1}{3}$. This shows that the minimum decrease of the measure is at most $2\frac{1}{3} \times l$. We will show that this minimum decrease can be bounded from below by $\min\{2\frac{1}{3} \times l,\ 15\}$. To do so, we consider clauses in Φ with literals that have been set to $False$ and show that every other configuration decreases the measure by at least the same quantity, or removes at least three variables.

We can assume that there are no clauses containing only literals that are set to $False$ since this results in a No-instance in which the whole formula Φ will be removed. First, consider a clause in Φ containing only one literal z that is not set to $False$. In this case, the variable z will be set to $True$. We note that, in the current situation, there can be only one clause that only contains z and literals that have been set to $False$. If the clause has size three, two occurrences of the v_i lead to the removal of one extra variable, which has more measure than $2 \times 2\frac{2}{3}$. With larger clauses, the measure decreases by an additional $2\frac{1}{3}$ for each extra literal: this remains more than $2\frac{1}{3} \times l$.

Second, consider clauses in Φ containing two or more literals that do not belong to the l literals that have not been set to $False$. It is possible that literals in such a clause have been set to $True$ due to the previous step where we first considered clauses with one literals that was not among the l literals set to $False$ in advance. If more than one of these literals is set to $True$, then we have a No-instance and Φ is removed completely. Hence, at most one literal in the clause has been set to $True$. If one literal has been set to $True$ in a clause of size three, the remaining variable will be set to $False$ decreasing the measure by an additional 5 while using only one occurrence of the l literals: this is more than given by $2\frac{1}{3} \times l$ and will remain more if we consider larger clauses also. Finally, if one literal has been set to $True$ and all remaining literals have been set to $False$ by new assignments as described in the previous sentence, then, because no two literals may occur in a clause together more than once, at least three different variables that are not among the variables initially set to $False$ are given a value: this gives the term 15 in $\min\{2\frac{1}{3} \times l,\ 15\}$.

What remains is to considering clauses in Φ containing two or more literals that are not among the l literals that have been set to $False$ in advance and in which no literals are set to $True$ by new assignments as described in the previous paragraph. Here, we distinguish between literals that are set to $False$ in advance, literals that are set to $False$ due to the effects described in the previous paragraph, and literals of variables that have no assigned value yet. Again, if all literals in a clause have been set to $False$, then we again have a No-instance. If all literals except for one have been set to $False$ due to the effects described in the previous paragraph, then the last literal will be set to $True$; if the clause has size three, this removes one variable while using one occurrences of the l literals; if the clause is larger, each extra literal increases the decrease of the measure by $2\frac{1}{3}$ (this is always as least as much as $2\frac{1}{3} \times l$). If all literals except for two have been set to $False$ due to the effects of the previous

paragraph, then Reduction Rule 3 will also remove one additional variable leading to the same decrease of the measure as in the previous case. Finally, if some literals have been set to $False$ due to the effects described in the previous paragraph, but at least three others remain, then each of the l literals only reduces the size of the clause giving exactly a decrease in the measure of $2\frac{1}{3} \times l$.

This proves the bound on the additional decrease of the measure of Φ under the first condition in the proposition.

For the decreases of the measure under the second condition, we can give a similar proof. The only difference is that Φ can contain one structure that decreases the measure by less than given under the first condition. This is the case if a variable in Φ exists which occurs in only two clauses and only with some of the l literals that are set to $False$: the situation excluded by the first condition and not by the second condition. If both clauses have size three, four of the l literals that have been set to $False$ are used while removing only one additional variable. This decreases the measure by 5 per four literals set to $False$. Using larger clauses, this again increases the decrease of the measure by $2\frac{1}{3}$ each. We conclude that the measure decreases by at least $\min\{5\lfloor l/4 \rfloor + 2\frac{1}{3}(l \bmod 4),\ 15\}$. □

If Reduction Rules 1-12 and Lemmas A.4, A.6, and A.7 do not apply, then we try to apply the following new reduction rule. This reduction rule considers a variable x of frequency three as in the following situation:

$$(x, v_1, v_2, \ldots) \wedge (x, v_3, v_4, \ldots) \wedge (x, v_5, v_6, \ldots) \wedge \Phi$$

Reduction Rule 13. If, in the above situation, there exists a variable z in Φ that occurs in a clause with only literals of the variables v_i, and all v_i from one of the clauses with x occur in some clause with z, then we apply the replacement: $z \to x$.

Proof of correctness. Setting $x \to True$ implies $z \to True$ because z occurs in a clause in which it occurs only with variables that are among the v_i. Also, setting $z \to True$ implies $x \to True$ because the v_i that are set to $False$ set all literals in a clause with x to $False$ except for x itself. We conclude that $x = z$ in any solution. □

Notice that this reduction rule removes a variable and thus decreases the measure by at least five.

We continue by giving the remaining lemmas describing the branching rules of our algorithm.

Lemma A.9. *If an* Exact Satisfiability *instance contains a variable x with $f(x) \geq 3$ that occurs only in clauses of size three, and such that the clauses containing x do not have the following form: $(x, v_1, w_1) \wedge (x, w_2, u_1) \wedge (x, w_2, u_2)$ with $f(v_1) = 3$, $f(w_i) = 2$, and $f(u_i) = 1$, for all i. Then, we can either reduce the instance to an equivalent smaller instance, or we can branch on the instance such that the associated branching number is at most* 1.02220.

Proof. We can assume that Lemmas A.4, A.6, and A.7 do not apply, otherwise we are done.

We consider the following situation:

$$(x, v_1, v_2) \wedge (x, v_3, v_4) \wedge (x, v_5, v_6) \wedge \Phi$$

Branching on x removes four variables if we set $x \to False$ by Reduction Rule 3. If we set $x \to True$ this results in the removal of seven variables and an additional decrease of the measure of the instance due to the effect that setting the v_is to $False$ has on Φ. Let l be the number of occurrences of the literals of v_i in Φ, and let $k(l)$ be the minimum additional decrease of the measure of Φ as a result of setting these literals to $False$. We can conclude that if $k(l) \geq 14$, we obtain a branching number of $\tau(20, 35 + k(l)) \leq \tau(20, 49) < 1.02171$.

Notice that Φ cannot contain a variable z that occurs in two clauses with only variables that are among the v_i as this must be with at least four different variables v_i and then Reduction Rule 13 applies. Hence, we can apply Proposition A.8(1) and use that $k(l) \geq \min\{2\frac{2}{3} \times l,\ 15\}$.

We will consider branching numbers for three different values of l. Lemma A.3(6) shows that at least 3 of the v_i must also occur in Φ. Actually, we can make this argument a little stronger by noticing that x may occur only in clauses with at most two unique variables as Reduction Rule 12 otherwise applies. Consequently, the v_i must occur at least 4 times in Φ: $l \geq 4$.

If $l \geq 6$, then $k(l) \geq 14$ as required.

If $l = 4$, then exactly four of the v_i occur exactly once in Φ and the other two do not occur in Φ. This means that at least one of the clauses with x must contain two literals also occurring in Φ; without loss of generality, let these variables be v_1 and v_2. Since $F(v_1) = F(v_2) = (2, 0)$, these two variables are combined to one variable y with $F(y) = (1, 1)$ by Reduction Rule 3 in the branch where $x \to False$. This fires Reduction Rule 11 decreasing the measure of the instance in this branch by an additional $2\frac{2}{3}$ by Lemma A.1. Hence, we obtain a decrease of the measure of at least $20 + 2\frac{2}{3} = 22\frac{2}{3}$ in total in the branch where $x \to False$. Since $k(l) \geq 9\frac{1}{3}$, the associated branching number is at most $\tau(35 + 9\frac{1}{3}, 22\frac{2}{3}) < 1.02173$.

Finally, let $l = 5$. If any of the three clauses with x contain two v_is with $F(v_i) = (2, 0)$, then we can repeat the argument of $l = 4$ as Reduction Rule 11 causes the measure to decrease by an additional $2\frac{2}{3}$ in the branch where we set $x \to False$.

The case that remains is when $l = 5$ and no clause containing two v_i with $F(v_i) = (2, 0)$ exists. In this case, two v_is must be unique variables and one v_i must have $F(v_i) = (3, 0)$: this is the one special case excluded in the statement of the lemma. □

Lemma A.10. *If an* Exact Satisfiability *instance contains a variable x with $f(x) \geq 3$ occurring in two clauses of size three and one clause of size four, then we can either reduce the instance to an equivalent smaller instance, or we can branch on the instance such that the associated branching number is at most* 1.02220.

Proof. We can assume that Lemmas A.4, A.6, A.7, and A.9 do not apply, otherwise we are done.

We have the following situation:

$$(x, v_1, v_2) \wedge (x, v_3, v_4) \wedge (x, v_5, v_6, v_7) \wedge \Phi$$

If we set $x \to True$, the measure decreases by 40 for removing eight variables, by $2\frac{1}{3}$ for removing one clause of size four, and by an additional quantity that we call k_Φ for the additional effect on Φ. If we set $x \to False$, the measure decreases by 5 for removing x, by 10 for the two replacements due to Reduction Rule 3, and by $2\frac{1}{3}$ for

reducing one clause of size four in size. To obtain a bound on k_Φ, we first observe that at least five occurrences of the variables v_i exist in Φ because otherwise Reduction Rule 12 can be applied. If there exists no variable z that occurs only with literals of the variables v_i in at least two clauses in Φ, then we apply Proposition A.8 to conclude that $k_\Phi \geq 11\frac{2}{3}$. In this case, we obtain a branching number of $\tau(54, 17\frac{1}{3}) < 1.02181$.

The only case that remains is when there exists a variable z that occurs only with literals of the variables v_i in at least two clauses in Φ. If these are at least three clauses or one of them has size at least four, then the literals of at least five variables v_i are in these clauses as no literal may occur in a clause with z twice: in this case, Reduction Rule 13 applies. Hence, exactly four literals of the v_i occur in clauses with z. More precisely, the situation is isomorphic to the following:

$$(x, v_1, v_2) \wedge (x, v_3, v_4) \wedge (x, v_5, v_6, v_7) \wedge (v_1, v_5, z) \wedge (v_3, v_6, z) \wedge \Phi$$

In this specific case, we branch on z. Setting $z \to True$ directly results in the removal of z, v_1, v_3, v_5, and v_6 and indirectly removes three more variables as Reduction Rule 3 sets $v_2 \to \neg x$, $v_4 \to \neg x$, and $v_7 \to \neg x$, i.e., eight variables are removed decreasing the measure by 40. Setting $z \to False$ results in the removal of z and the setting of $v_5 \to \neg v_1$ and $v_6 \to \neg v_3$. This results in the following clauses with x: $(x, v_1, v_2) \wedge (x, v_3, v_4) \wedge (x, \neg v_1, \neg v_3, v_7)$. To this instance, Reduction Rule 6 is applied setting $x \to False$ resulting in $v_2 \to \neg v_1$ and $v_4 \to \neg v_3$. In total, six variables are removed and a clauses of size four is reduced: the measure decreases by $32\frac{1}{3}$. This gives a branching number of $\tau(32\frac{1}{3}, 40) < 1.01943$. □

Lemma A.11. *If an* Exact Satisfiability *instance contains a variable x with $f(x) \geq 3$, then we can either reduce the instance to an equivalent smaller instance, or we can branch on the instance such that the associated branching number is at most* 1.02220.

Proof. We can assume that Lemmas A.4, A.6, A.7, A.9, and A.10 do not apply, otherwise we are done. This means that we have to consider only the following four remaining cases:

Two clauses of size three and one clause of size at least five. In this case, we have the following situation:

$$(x, v_1, v_2) \wedge (x, v_3, v_4) \wedge (x, v_5, v_6, v_7, v_8, \ldots) \wedge \Phi$$

Setting $x \to False$ removes three variables since Reduction Rule 3 applies, and it reduces the larger clauses in size: this gives a decrease of the measure of $15 + 2\frac{1}{3} = 17\frac{1}{3}$. Setting $x \to True$ removes at least nine variables, removes a clause of size at least five, and sets at least six literals in Φ to $False$ since Reduction Rule 12 would otherwise apply. If the condition in the second case of Proposition A.8 applies, then the measure decreases by at least $45 + 4\frac{2}{3} + 9\frac{2}{3} = 59\frac{1}{3}$ since the effect of the six $False$ literals in Φ is an additional decrease of the measure of at least $9\frac{2}{3}$ by Proposition A.8(2). The resulting branching number equals $\tau(59\frac{1}{3}, 17\frac{1}{3}) < 1.02062$.

We will now show that the condition in the second case of Proposition A.8 applies. We can restrict ourselves to the case where the large clause with x is of size at most six, otherwise at least two extra variables are removed when $x \to True$: these have more measure than the $9\frac{2}{3}$ we need to prove. If Φ contains a variable z that, in three

clauses, occurs only with literals of the variables v_i, then this must be with at most 5 of the variables v_i if the third clause has size five, and with at most 6 of the variables v_i if the third clause has size six, as otherwise Reduction Rule 13 applies. However, no such configuration with only five of the variables v_i exist since there are at least six slots to full in the three clauses with z. Furthermore, it is not so hard to check that Lemma A.9 applies to variables z in that occur in three clauses with six if the variables v_i.

One clause of size three and two larger clauses. We have the following situation:

$$(x, v_1, v_2) \wedge (x, v_3, v_4, v_5, \ldots) \wedge (x, v_6, v_7, v_8, \ldots) \wedge \Phi$$

Setting $x \to False$ removes two variable as Reduction Rule 3 sets $v_2 \to \neg v_1$ and reduces the two larger clauses in size: this decreases the measure by at least $10 + 2 \times 2\frac{1}{3} = 14\frac{2}{3}$. Setting $x \to True$ removes at least nine variables, removes two clauses of size at least four, and sets at least six literals in Φ to $False$ as Reduction Rule 12 would otherwise apply. This decreases the measure by at least $45 + 4\frac{2}{3} + 9\frac{2}{3} = 59\frac{1}{3}$ if the effect on the measure of the six $False$ literals in Φ is at least $9\frac{2}{3}$, which is the case if the condition in the second case of Proposition A.8 applies. The resulting branching number equals $\tau(59\frac{1}{3}, 14\frac{2}{3}) < 1.02207$.

Now, the second case of Proposition A.8 applies for the same reasons as in the previous case where we considered two clauses of size three and one clause of size at least five. That is, either two additional variables are removed when $x \to True$, or no variable in three clauses with literals of the variables v_i exists because either Reduction Rule 13 is applicable, or we have already branched on such variables.

Three clauses of size at least four. If all clauses have size at least four, then we have the following situation:

$$(x, v_1, v_2, v_3, \ldots) \wedge (x, v_4, v_5, v_6, \ldots) \wedge (x, v_7, v_8, v_9, \ldots) \wedge \Phi$$

Setting $x \to False$ removes one variable and reduces all three clauses in size: this decreases the measure by at least $5 + 3 \times 2\frac{1}{3} = 12$. Setting $x \to True$ removes at least ten variables, removes at least three clauses of size at least four, and sets at least seven literals in Φ to $False$ as Reduction Rule 12 would otherwise fire. This decreases the measure by at least $50 + 7 + 10 = 67$ since again, by the same reasoning as in the above two cases, either two additional variables are removed, or the second case of Proposition A.8 applies to the at least seven literals that are set to $False$ in Φ. The resulting branching number equals $\tau(67, 12) < 1.02212$.

The special case of three clauses of size three. At this point, the only variables x with $f(x) = 3$ that remain correspond to the following situation that was explicitly excluded in the statement of Lemma A.9:

$$(x, v_1, v_2) \wedge (x, v_3, u_1) \wedge (x, v_4, u_2) \wedge \Phi$$

with $f(v_1) = 3$, $f(v_2) = f(v_3) = f(v_4) = 2$ and $f(u_1) = f(u_2) = 1$.

Since this case represents the only remaining variables of frequency three, v_1 must be a variable similar to x. Therefore, a more specific view of the current case is:

$$(x, v_1, v_2) \wedge (x, v_3, u_1) \wedge (x, v_4, u_2) \wedge (v_1, v_5, u_3) \wedge (v_1, v_6, u_4) \wedge \Phi$$

with $f(v_i) = 2$ and $f(u_i) = 1$.

Branching on x results in the required branching number of $\tau(45 + 4\frac{2}{3}, 20) = \tau(49\frac{2}{3}, 20) < 1.02154$. Namely, setting $x \to True$ removes seven variables in the clauses with x, and two variables in the other two clauses due to Reduction Rule 3. Moreover, the measure of Φ is reduced by at least $4\frac{2}{3}$ by Proposition A.5 since v_2 and v_3 also occur in Φ. And, in the other branch, setting $x \to False$ removes x and three other variables due to Reduction Rule 3. □

We now take all the above lemmas together to obtain our result on Partition Into Triangles on graphs of maximum degree four.

Theorem A.12. *There is an $\mathcal{O}(1.02220^n)$-time and linear-space algorithm for* Partition Into Triangles *on graphs of maximum degree four.*

Proof. We first use Theorem 4.9 from Chapter 4 together with Lemma A.2 to obtain an Exact Satisfiability instance of measure at most n that is equivalent to the Partition Into Triangles instance on graphs of maximum degree four.

To this instance, we exhaustively apply Reduction Rules 1-12 and Lemmas A.4-A.11. As a result, we generate a branching tree with at most 1.02220^n leaves, each containing an instance of Exact Satisfiability in which all variables x satisfy $F(x) = (1, 0)$ or $F(x) = (2, 0)$. It is known that these instances can be solved in polynomial time and linear space, see for example [67]. This is true because such an instance is equivalent to the question whether the following graph $H = (V', E')$ has a perfect matching. Let X be the set of variables, and $\mathcal{C}$ be the set of clauses of a remaining Exact Satisfiability instance. We construct H by letting $V' = \mathcal{C}$ and introducing an edge for each variable $x \in X$ of frequency two between the corresponding clauses. We also add self-loops to all clauses containing a variable x of frequency one. It is not hard to see that every solution of the Exact Satisfiability instance corresponds to a perfect matching in H and vice versa. □

We notice that the polynomial part of the running time of this algorithm consists of only two components. One, the time required to test which reduction rules and which lemmas should be applied to the current instance. Two, the time required to test whether there exists a perfect matching in the graphs we build in the leaves of the search tree. Both can be implemented quite efficiently, and thus no large polynomial factors are hidden in the running time of the algorithm. This makes it a complicated but practical and very fast exponential-time algorithm.

A.2. Case Analysis for the Final Algorithm for Dominating Set

In Chapter 5, we claimed an $\mathcal{O}(1.4969^n)$-time and polynomial-space algorithm for Dominating Set. We will give a proof of this claim below.

Consider Algorithm 5.4 in Section 5.3.8 and the measure-and-conquer analysis of the previous algorithm in the step-by-step improvement series in Section 5.3.7. For a proper analysis of Algorithm 5.4, we further subdivide the case analysis of the algorithm in Section 5.3.7 by differentiating between different kinds of elements of frequency two depending on the kind of set their second occurrence is in. Let r_{f3} and $r_{f\geq 4}$ be the number of elements of frequency two whose second occurrence is in a set of size two with a *frequency three* element and with a *higher frequency* element, respectively. And, let r_{s3} and $r_{s\geq 4}$ be the number of elements of frequency two whose second occurrence is in a set of *cardinality three* and a set of *greater cardinality*, respectively.

We consider branching on a set S with r_i elements of frequency i, for all $|S| \geq 3$, all $|S| = \sum_{i=2}^{\infty} r_i$, and all $r_2 = r_{f3} + r_{f\geq 4} + r_{s3} + r_{s\geq 4}$. Because S is of maximum cardinality, we consider only subcases with $r_{s4} > 0$ if $|S| \geq 4$. For these cases, we will initially derive lower bounds on the decrease of the measure. In order to keep the bounding cases of the associated numerical problem corresponding to real instances on which the algorithm can branch, we perform a subcase analysis dealing with the more subtle details later in the analysis. Let $\mathcal{R}$ be the collection of sets containing a frequency-two element from S, excluding S itself. Notice that the algorithm takes the sets in $\mathcal{R}$ in the set cover if we discard S.

In the branch where S is taken in the solution, we again start with a decrease in measure of $w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i(i-1)$ due to the removal of S, its elements, and the reduction in size of the sets containing elements from S. Additionally, for each element of frequency two occurring in a set of size two, the measure is not decreased by $\Delta w(|S|)$ for reducing this set in size, but by $w(2)$ since the set will be removed by Reduction Rule 5.3. Similarly, for each element of frequency two occurring in a set of size three, the decrease is $\Delta w(3)$ instead of $\Delta w(|S|)$ if $|S| \geq 4$. Together, this gives an extra decrease of $(r_{f3}+r_{f\geq 4})(w(2)-\Delta w(|S|))+r_{s3}(\Delta w(3)-\Delta w(|S|))$. Finally, if S contains elements of frequency two whose second occurrence is in a set of size two containing an element of frequency three, then these frequency three elements are reduced in frequency because these sets have become singleton subsets; these elements can also be removed completely if they occur in multiple such sets. This leads to an additional decrease of the measure of at least $[r_{f3} > 0] \min\{v(3),\ r_{f3}\Delta v(3)\}$.

In the branch where S is discarded, the measure decreases by $w(|S|)+\sum_{i=2}^{\infty} r_i \Delta v(i)$ because of removing S. The sets in $\mathcal{R}$ are taken in the set cover; this decreases the measure by at least $r_2(v(2) + w(2))$ because we have at least r_2 sets, and they together contain at least r_2 other elements by Reduction Rule 5.6. Additionally, the r_{s3} sets of size three and the $r_{s\geq 4}$ sets of size at least four reduce the measure by at least an additional $r_{s3}\Delta w(3) + r_{s\geq 4}(w(4) - w(2))$. And, if these sets are of size two while containing elements of frequency at least three, we can add $[r_{f3} > 0]\Delta v(3) + [r_{f\geq 4} > 0](v(4) - v(2))$; notice that we cannot add $r_{f3}\Delta v(3)$ as one element may be in multiple sets in $\mathcal{R}$.

Furthermore, other sets are reduced in size because of the removal of all elements

in $\bigcup_{S' \in \mathcal{R}} S'$. Let $q_{\vec{r}}$ be the number of element occurrences outside S and $\mathcal{R}$ that are removed after taking all sets in $\mathcal{R}$ in the set cover in the subcase corresponding to $\vec{r}$, i.e., for this subcase, this is the number of times a set is reduced in cardinality by one. By using the steepness inequalities in the same way as in the previous section, we decrease the measure by at least an additional: $\min\{q_{\vec{r}}\Delta w(|S|), \lfloor \frac{q_{\vec{r}}}{2} \rfloor w(2) + (q_{\vec{r}} \bmod 2)\Delta w(|S|)\}$.

We now give a lower bound on $q_{\vec{r}}$. There are at least r_2 additional elements that are removed; these are all of frequency at least two, hence at least r_2 additional element occurrences are removed. These occurrences do not all need to be outside of $\mathcal{R}$: for every set in $\mathcal{R}$ of size three, there is one empty slot that could be filled by an occurrence of these elements. Similarly, for every set in $\mathcal{R}$ of size at least four, there are $|S| - 2$ empty slots that can be filled with these elements. Furthermore, if the sets in $\mathcal{R}$ contain elements of frequency three or at least four, then the number of removed element occurrences increases by 1 or 2, respectively. Altogether, we find that $q_{\vec{r}} \geq \max\{0,\ r_2 + [r_{f3} > 0] + 2[r_{f\geq 4} > 0] - r_{s3} - (|S| - 2)r_{s\geq 4}\}$.

Finally, we split some recurrences based on Reduction Rule 5.7. If $r_{s3} > 0$, and a corresponding set of size three contains only elements of frequency two, then this set is removed when taking S in the set cover: this can either be done by Reduction Rule 5.7, or by the old Reduction Rules 5.1 and 5.3. We split the recurrence relations with $r_{s3} > 0$ into two separate cases. We identify these case by introducing yet another identifier: $r_{\text{rule5.7}}$. One subcase has $r_{\text{rule5.7}} = True$, and one has $r_{\text{rule5.7}} = False$. If $r_{\text{rule5.7}} = True$, we add an additional $2v(2)+w(2)$ to the formula of Δk_{take} representing the additional set and elements that are removed. Notice that we can do this because we did not take these two frequency-two elements and this cardinality two set into account in the new restrictions on the weights we imposed before starting the above analysis.

This leads to the following set of recurrence relations:

$$\forall\ |S| \geq 3, \forall\ r_i : \sum_{i=2}^{\infty} r_i = |S|, \forall\ r_2 = r_{f3} + r_{f\geq 4} + r_{s3} + r_{s\geq 4}, \forall\ r_{\text{rule5.7}} \in \{True, False\}$$

$$\text{with } r_{s\geq 4} = 0 \text{ if } |S| = 3, \text{ and } r_{\text{rule5.7}} = False \text{ if } r_{s3} = 0 \qquad :$$

$$N(k) \leq N(k - \Delta k_{\text{take}}) + N(k - \Delta k_{\text{discard}})$$

$$\begin{aligned}
\Delta k_{\text{take}} \quad \geq \quad & w(|S|) + \sum_{i=2}^{\infty} r_i v(i) + \Delta w(|S|) \sum_{i=2}^{\infty} r_i(i-1) + (r_{f3} \\
& + r_{f\geq 4})(w(2) - \Delta w(|S|)) + r_{s3}(\Delta w(|3|) - \Delta w(|S|)) \\
& + [r_{f3} > 0] \min\{v(3), r_{f3}\Delta v(3)\} + [r_{\text{rule5.7}}](2v(2) + w(2)) \\
\Delta k_{\text{discard}} \quad \geq \quad & w(|S|) + \sum_{i=2}^{\infty} r_i \Delta v(i) + r_2(v(2) + w(2)) + r_{s3}\Delta w(3) + r_{s\geq 4}(w(4) - w(2)) \\
& + [r_{f3} > 0]\Delta v(3) + [r_{f\geq 4} > 0](v(4) - v(2)) \\
& + \min\left\{q_{\vec{r}}\Delta w(|S|),\ \left\lfloor \frac{q_{\vec{r}}}{2} \right\rfloor w(2) + (q_{\vec{r}} \bmod 2)\Delta w(|S|)\right\}
\end{aligned}$$

Here, we let $q_{\vec{r}}$ be tight the above lower bound.

Unfortunately, this does not yet complete the description of the new numerical optimisation problem. For some specific cases, we will prove that we can increase the

corresponding Δk_{take} and $\Delta k_{\text{discard}}$. We do this not only to improve the proven running time of the algorithm, but also to make sure that the recurrences representing bounding cases of this numerical problem represent actual instances that can be branched on. In other words, the bounding cases should not be based on non-tight lower bounds on the reductions of the measure. For one subcase, we not only increase Δk_{take} and $\Delta k_{\text{discard}}$, but also split the corresponding recurrence into two separate recurrence relations.

Consider the following cases where $S = \{e_1, e_2, e_3\}$ or $S = \{e_1, e_2, e_3, e_4\}$. For each case considered, we write $\Delta k_{\text{take}} += x$ and $\Delta k_{\text{discard}} += x$ if we add the additional decrease of the measure x to the given quantities for the specific subcase.

1. $|S| = 3, r_2 = r_{s3} = 3$ $(q_{\vec{r}} = 0)$.

   (a) $r_{\text{rule5.7}} = False$.
   Remind that this means that none of the sets in $\mathcal{R}$ can consist solely of frequency-two elements, i.e., $\mathcal{R}$ contains one element of frequency at least three existing in all three sets in $\mathcal{R}$, or $\mathcal{R}$ contains at least two elements of frequency at least three. In both cases, we consider what happens in the branch where the algorithm discards S and takes the sets in $\mathcal{R}$ in the set cover.

   In the first case, no frequency-two element may occur in two sets in $\mathcal{R}$ by Reduction Rule 5.5. Thus, we need at least one more element than the three counted in the above analyses: we decrease the measure by at least an additional $\Delta v(3) + v(2)$. Furthermore, by taking the sets in $\mathcal{R}$ in the set cover, there are at least three element occurrences outside S and $\mathcal{R}$ removed. No two of these elements may exist together in a set of size two by Reduction Rule 5.7. Hence, this gives an at least additional $3\Delta w(3)$.

   In the second case, the measure is decreased by at least an additional $2\Delta v(3)$. Moreover, the sum of the frequencies of the elements in $(\bigcup \mathcal{R}) \setminus S$ is at least eight, while there are only six open slots in $\mathcal{R}$, i.e., there are at least two elements outside of S and $\mathcal{R}$ removed. If there are exactly two extra element occurrences removed which occur together in a size two set, then this set is a subset of a set in $\mathcal{R}$ because the two frequency three elements must be in some set together: this is not possible due to Reduction Rule 5.3. In any other case, the decrease in measure due to the removal of these element occurrences is at least $2\Delta w(3)$.

   Altogether, we add the minimum of both quantities to $\Delta k_{\text{discard}}$ by setting $\Delta k_{\text{discard}} += \min\{\Delta v(3) + v(2) + 3\Delta w(3),\ 2\Delta v(3) + 2\Delta w(3)\}$.

   (b) $r_{\text{rule5.7}} = True$.
   Notice that subcase (a) dominates every case where there exist one element of frequency at least three and one extra element in $\mathcal{R}$, or where there exist at least two elements of frequency at least three in $\mathcal{R}$. Furthermore, we can disregard the case with one element of frequency at least three and two frequency-two elements because of Reduction Rule 5.5. Hence, we can restrict ourselves to the case where $(\bigcup \mathcal{R}) \setminus S$ consists of at least three frequency-two elements.

   Consider the branch where the algorithm takes S in the set cover. If there are only three frequency-two elements, that is, $\mathcal{R} = \{\{e_1, e_4, e_5\}, \{e_2, e_4, e_6\},$

$\{e_3, e_5, e_6\}\}$, then Reduction Rules 5.7 and 5.1 remove all sets and elements from $\mathcal{R}$. This gives an additional decrease of $v(2) + 2w(2)$ to Δk_{take} besides the $2v(2) + w(2)$ we counted already because $r_{\text{rule5.7}} = True$. If there are four or more frequency-two elements in $(\bigcup \mathcal{R}) \setminus S$, then Reduction Rule 5.7 can be applied at least twice reducing the measure by the same amount. We set $\Delta k_{\text{take}} \mathrel{+}= v(2) + 2w(2)$.

2. $|S| = 3, r_2 = r_{s3} = 2, r_3 = 1$ $(q_{\vec{r}} = 0)$.

   (a) $r_{\text{rule5.7}} = False$.

   $\mathcal{R}$ contains one element of frequency at least three existing in both sets in $\mathcal{R}$, or $\mathcal{R}$ contains at least two elements of frequency at least three. We consider the branch where S is discarded and the sets in $\mathcal{R}$ are taken in the set cover.

   In the first case, $\mathcal{R} = \{\{e_1, e_4, e_5\}, \{e_2, e_4, e_6\}\}$ with $f(e_4) \geq 3$ and all other elements have frequency two. We have an extra decrease in measure of $\Delta v(3) + v(2)$ because e_4 has higher frequency and we have an extra element. Additionally, we look at the number of element occurrences outside of S and $\mathcal{R}$ that are removed: there are at least three of these. Hence, we get an additional decrease of $\min\{3\Delta w(3),\ w(2) + \Delta w(3)\}$ equivalent to setting $q_{\vec{r}} = 3$.

   In the second case, we decrease the measure by at least an additional $2\Delta v(3)$ because of the higher frequency elements. Moreover, there are at least two element occurrences outside S and $\mathcal{R}$ removed: this gives an additional decrease of $\min\{2\Delta w(3),\ w(2)\}$ equivalent to setting $q_{\vec{r}} = 2$.

   We add the minimum of both quantities to $\Delta k_{\text{discard}}$ by setting $\Delta k_{\text{discard}} \mathrel{+}= \min\{\Delta v(3) + v(2) + \min\{3\Delta w(3),\ w(2) + \Delta w(3)\},\ 2\Delta v(3) + \min\{2\Delta w(3), w(2)\}\}$.

   (b) $r_{\text{rule5.7}} = True$.

   Subcase (a) dominates every case with elements of frequency at least three in $\mathcal{R}$. Thus, we may assume that $\mathcal{R}$ contains only elements of frequency two, and there are at least three of them because of Reduction Rule 5.5. In the branch where S is discarded and all sets in $\mathcal{R}$ are taken in the set cover, one extra element of frequency two and at least two element occurrences outside of S and $\mathcal{R}$ are removed. Altogether, we add their contribution to the measure to $\Delta k_{\text{discard}}$ by setting $\Delta k_{\text{discard}} \mathrel{+}= v(2) + w(2)$.

3. $|S| = 3, r_2 = r_{s3} = 1, r_3 = 2, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 0)$.

   In the branch where S is discarded, there must be two elements in the unique set in $\mathcal{R}$ and one must be of frequency at least three. Hence, we set $\Delta k_{\text{discard}} \mathrel{+}= \Delta v(3) + v(2)$.

4. $|S| = 3, r_2 = r_{s3} = 2, r_4 = 1, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 0)$.

   Analogous to the case where $|S| = 3, r_2 = r_{s3} = 2, r_3 = 1, r_{\text{rule5.7}} = False$ (Case 2a), we set $\Delta k_{\text{discard}} \mathrel{+}= \min\{\Delta v(3) + v(2) + \min\{3\Delta w(3), w(2) + \Delta w(3)\}, 2\Delta v(3) + \min\{2\Delta w(3), w(2)\}\}$.

5. $|S| = 3, r_2 = 3, r_{f3} = 1, r_{s3} = 2, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 2)$.

   Since $r_{f3} = 1$, we have a set R_1 of size two in $\mathcal{R}$ with an element of frequency three. If this element is also in the other two sets in $\mathcal{R}$, then technically we are in this case because none of the r_{s3} sets of size three consist solely of frequency-two elements. However, after taking S in the solution, R_1 disappears because it has become a singleton set and Reduction Rule 5.7 fires on the remaining sets in $\mathcal{R}$. The recurrence relation for the corresponding case with $r_{\text{rule5.7}} = True$ correctly represents this case. Therefore, we have to consider only the case where there is another higher frequency element that prevents the r_{s3} sets of size three from having only frequency-two elements.

   For this case, consider the branch where S is discarded and the sets in $\mathcal{R}$ are taken in the set cover. Since there is another element of frequency at least three, we decrease the measure by at least an additional $\Delta v(3)$. Furthermore, there are at least eight element occurrences of the elements in $(\bigcup \mathcal{R}) \setminus S$, while $\mathcal{R}$ has only five available slots. Thus, $q_{\vec{r}}$ should be 3 instead of 2. This allows us to set $\Delta k_{\text{discard}} \mathrel{+}= \Delta v(3) + \Delta w(3)$.

6. $|S| = 3, r_2 = 3, r_{f\geq 4} = 1, r_{s3} = 2, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 3)$.

   In contrast to the previous case, the element of frequency at least four in the size two set in $\mathcal{R}$ can cause all other sets in $\mathcal{R}$ not to consist of only frequency-two elements. We split this case into two separate recurrences: one where this element has frequency four, and one where it has higher frequency.

   In the first case, we can set $\Delta k_{\text{take}} \mathrel{+}= \Delta v(4)$, because after taking S in the set cover, this element exists in a singleton set and hence its frequency is reduced by one. Notice that we could not bound this decrease before since the frequency of the element was unbounded.

   In the second case, we remove at least one extra element occurrence outside $\mathcal{R}$ and S. If all other elements in $\mathcal{R}$ have frequency two, this extra element occurrence cannot be in a set with another removed element occurrence by Reduction Rule 5.5, and we can add $\Delta w(3)$ to the decrease (not needing to increasing $q_{\vec{r}}$ by one). If some other element in $\mathcal{R}$ has frequency at least three, then we remove at least two extra element occurrence outside $\mathcal{R}$ and S. Taking the worst case of both, we can set $\Delta k_{\text{discard}} \mathrel{+}= \Delta w(3)$.

7. $|S| = 3, r_{f3} = 2, r_{s3} = 1, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 3)$.

   There are two possible situations: either there are two different elements of frequency three in the r_{f3} sets of size two, or both sets contain the same element of frequency three. In the latter case, this element cannot also be in the third set in $\mathcal{R}$ because this would trigger Reduction Rule 5.6 on S. Since $r_{\text{rule5.7}} = False$, there must be another element of frequency at least three in this third set in $\mathcal{R}$. In both cases, we have an extra element of frequency at least three; hence, we can set $\Delta k_{\text{discard}} \mathrel{+}= \Delta v(3)$.

8. $|S| = 3, r_{f\geq 4} = 2, r_{s3} = 1$ $(q_{\vec{r}} = 4)$.

(a) $r_{\text{rule5.7}} = False$.

Similar to the previous case, there are two possible situations: either there are two different elements of frequency at least four in the $r_{f\geq 4}$ sets of size two, or both sets contain the same element of frequency at least four.

In the first case, we have an additional reduction of at least $v(4) - v(2)$ due to this element. Moreover, at least ten element occurrences of the elements in $(\bigcup \mathcal{R}) \setminus S$ are removed while there are only 4 slots available in $\mathcal{R}$ and currently $q_{\vec{r}} = 4$. This means we decrease the measure by an additional $\min\{2\Delta w(3),\ w(2)\}$.

In the second case, the element of frequency at least four cannot be in the third set in $\mathcal{R}$ because then Reduction Rule 5.6 applies to S; hence, there must be an element of frequency at least three in the third set in $\mathcal{R}$. This gives an additional $\Delta v(3)$, while counting the number of removed element occurrences of the elements in $(\bigcup \mathcal{R}) \setminus S$ gives us an additional $\Delta w(3)$.

Altogether, we add the minimum of both cases to $\Delta k_{\text{discard}}$ by setting $\Delta k_{\text{discard}} \mathrel{+}= \min\{\Delta v(3) + \Delta w(3),\ v(4) - v(2) + \min\{2\Delta w(3),\ w(2)\}\}$.

(b) $r_{\text{rule5.7}} = True$.

Consider the branch where we take S in the solution. After removing the elements e_1, e_2, and e_3, no sets outside of $\mathcal{R}$ are reduced in size. This is true because the two size two sets in $\mathcal{R}$ that are removed contain an element of frequency at least four: after removing their occurrences in $\mathcal{R}$, they still have frequency at least two. Moreover, $r_i = 0$ for all $i \geq 3$, so no other element occurrences outside of $\mathcal{R}$ are removed. As a result, two sets of size two or three are merged by Reduction Rule 5.7. Notice that we already counted the decrease due to removing a set of size two and its elements, but not the decrease due to the fact that two other sets are replaced by one larger one. This gives an additional decrease of the measure; we set $\Delta k_{\text{take}} \mathrel{+}= \min\{2w(3) - w(4),\ w(3) + w(2) - w(3),\ 2w(2) - w(2)\}$.

9. $|S| = 3, r_{f3} = 3, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 4)$.

Consider the branch where S is discarded and the sets in $\mathcal{R}$ are taken in the set cover. Since all sets in $\mathcal{R}$ are of size two, there are three different frequency three elements in $\mathcal{R}$ by Reduction Rule 5.6 instead of the one we count now. These elements decrease the measure by an additional $2\Delta v(3)$. Moreover, we removed at least nine element occurrences of the elements in $(\bigcup \mathcal{R}) \setminus S$ from which only three can be in $\mathcal{R}$ and $q_{\vec{r}}$ counts only four: at least two more are removed decrease the measure by an additional $\min\{2\Delta w(3),\ w(2)\}$. Altogether, we set $\Delta k_{\text{discard}} \mathrel{+}= 2\Delta v(3) + \min\{2\Delta w(3),\ w(2)\}$.

10. $|S| = 3, r_{f\geq 4} = 3, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 5)$.

This case is similar to the above: there must be at least two more elements of frequency at least four decrease the measure by $2(v(4) - v(2))$ in the branch where S is discarded. And, by a counting removed element occurrences of the elements in $(\bigcup \mathcal{R}) \setminus S$, we should increase $q_{\vec{r}}$ by four. Hence, we set $\Delta k_{\text{discard}} \mathrel{+}= 2(v(4) - v(2)) + \min\{4\Delta w(3),\ 2w(2)\}$.

11. $|S| = 4, r_{s\geq 4} = 4, r_{\text{rule5.7}} = False$ $(q_{\vec{r}} = 0)$.

    In the branch where S is discarded and the sets in $\mathcal{R}$ are taken in the solution, we count only a measure of $4v(2)$ for the removed elements in $(\bigcup \mathcal{R}) \setminus S$. There must be at least four elements in $(\bigcup \mathcal{R}) \setminus S$ by Reduction Rule 5.6 and there are twelve slots to fill. This can either be done by six elements of frequency two, or by using higher frequency elements. Hence, we set $\Delta k_{\text{discard}} \mathrel{+}= \min\{2v(2),\ \Delta v(3)\}$.

We solve the numerical problem associated with the described set of recurrence relations and obtain a solution of $N(k) \leq 1.28935^k$ using the following set of weights:

i	1	2	3	4	5	6	7	> 7
$v(i)$	0.000000	0.011179	0.379475	0.526084	0.573797	0.591112	0.595723	0.595723
$w(i)$	0.000000	0.353012	0.706023	0.866888	0.943951	0.981278	0.997062	1.000000

This leads to an upper bound of $\mathcal{O}\big(1.28759^{(0.595723+1)n}\big) = \mathcal{O}(1.49684^n)$ on the running time of the algorithm. The bounding cases of the numerical problem are:

$|S| = r_3 = 3$ $|S| = r_4 = 3$ $|S| = 4, r_2 = r_{e\geq 4} = 4$ $|S| = r_5 = 4$ $|S| = r_5 = 5$
$|S| = r_6 = 5$ $|S| = r_6 = 6$ $|S| = r_7 = 6$ $|S| = r_7 = 7$ $|S| = r_8 = 7$ $|S| = r_8 = 8$

This proves Theorem 5.1.

A.3. Case Analysis for the Final Algorithm for Edge Dominating Set

In Chapter 6, we claimed an $\mathcal{O}(1.3226^n)$-time and polynomial-space algorithm for Edge Dominating Set. We will give a proof of this claim below.

Consider the improvement of Algorithm 6.3 described in Section 6.5. Here, we have improved the worst case of the numerical optimisation problem that corresponded to $d = 3$, $r_2 = 2$, $r_3 = 1$; the worst case that equals Graph 1 in Figure A.1. We have done so by splitting the case into two subcases: one where this case corresponds to a separate connected component and in which we now branch on another vertex (Graph 1 in Figure A.1), and one where this case does not equal a separate connected component and thus more edges are removed. As both cases lead to recurrence relations with smaller solutions, this improves the upper bound on the running time of the algorithm.

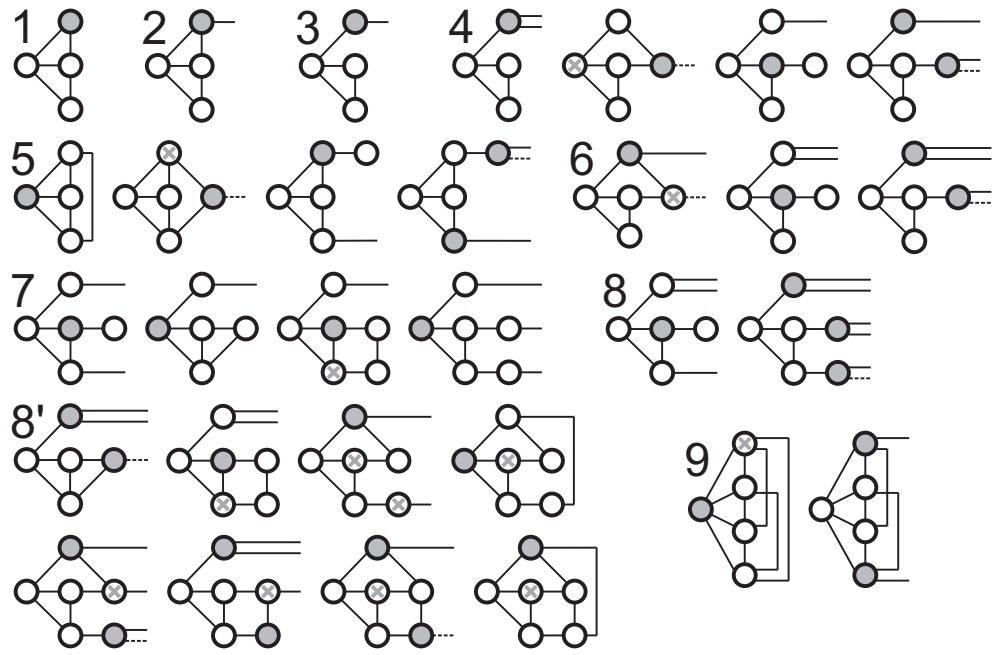
We give the other improvements on the algorithms in a more schematic manner. Listed in the order in which they appear as worst cases in the improvement series, we introduce the following set of alternative branching strategies to Algorithm 6.3. The numbering corresponds to the subgraphs of $G[U]$ drawn in Figure A.1. We refer to the vertex on which the previous algorithm could branch as v (leftmost vertex in the Figure A.1), and we denote the decrease in the measure of the subproblems generated by the alternative branching strategy by $\Delta_1, \Delta_2, \ldots$. At each subcase, we either increase the current lower bound d_2 on the number of edges between $N_U[v]$ and the rest of $G[U]$, or find other means of increasing the lower bounds on the total reduction in measure when branching (Δ_{indep} and Δ_{vc}). See Table A.1, for the upper bounds on the running times of the individual algorithms in the series.

1. $d = 3, r_2 = 2, r_3 = 1$. See introductory example. The subcase tight to $d_2 = 0$ is handled more efficiently by branching on a U-degree two vertex. This results in $\Delta_1 = \Delta_2 = 2w(3) + 2w(2)$. All other subcases have at least two edges with only one endpoint in $N_U[v]$, thus: $d_2 = 2$.

2. $d = 3, r_2 = 1, r_3 = 2$. If there is a unique edge in $G[U]$ with only one endpoint u in $N_U[v]$, then u has U-degree three. Branch on u and apply Reduction Rule 6.1 to any 3-clique remaining in $G[U]$. Because the other vertex incident to this unique edge has weight at least $w(1)$, and when its U-degree is reduced by one this reduces its weight by at least $\Delta w(3)$; we derive $\Delta_1 = 3w(3) + w(2) + w(1), \Delta_2 = 3w(3) + w(2) + \Delta w(3)$. For the other subcases with $d = 3, r_2 = 1, r_3 = 2$ the number of edges from $N_U[v]$ to the rest of $G[U]$ is at least 3 by a parity argument. Hence, we can now use $d_2 = 3$ for these cases.

Strategies	none	1	1-2	1-3	1-4	1-5	1-7	1-8	1-9
$\mathcal{O}(c^n)$:	1.3323	1.3315	1.3296	1.3280	1.3265	1.3248	1.3240	1.3228	1.3226
$\Omega(c^n)$:	1.3160	1.2968	1.2968	1.2968	1.2968	1.2753	1.2753	1.2753	1.2753

Table A.1. Bounds on the running times of the algorithms in the improvement series.



The leftmost vertex in every subgraph corresponds to a vertex we could branch on in Algorithm 6.3 and grey vertices represent more efficient alternatives. If multiple vertices are grey, simultaneously branch on these vertices generating four or eight subproblems. Crossed vertices represent vertices branched on directly hereafter, but only in the subproblems where this induces extra 1, 2 or 3-cliques. Sometimes small connected components that are paths remain in a subproblem; these are immediately branched upon also.
Unfinished edges always connect to vertices outside the drawn subgraph, and there are no other edges in $G[U]$ between vertices with at least one drawn endpoint. Dashed edges are optional.

Figure A.1. More efficient branching strategies on possible subgraphs of $G[U]$.

3. $d = 3, r_2 = 3$. Similar to Case 2: $\Delta_1 = w(3)+3w(2)+w(1), \Delta_2 = w(3)+3w(2)+\Delta w(3)$. For the remaining subcases, we have that $d_2 = 3$.

4. $d = 3, r_2 = 2, r_3 = 1$. Case 1 reappears; consider four more subcases representing $d_2 = 2$.

   (a) Both edges with only one endpoint in $N_U[v]$ are incident to the same vertex $u \in N_U(v)$. Branch on u and apply Reduction Rule 6.1 if possible; this is similar to Cases 2 and 3. $\Delta_1 = 2w(3) + 2w(2) + 2w(1), \Delta_2 = 2w(3) + 2w(2) + 2\Delta w(3)$.

   Let $u, w \in N_U(v)$ be incident to the edges with only one endpoint in $N_U[v]$ such that u has U-degree two and w has U-degree three.

   (b) Both edges in $G[U]$ with only one endpoint in $N_U[v]$ are incident to the same vertex $x \notin N_U(v)$. Branch on x, and if it is put in the vertex cover

also branch on v. Because paths on two vertices are removed from $G[U]$: $\Delta_1 = \Delta_2 = \Delta_3 = 2w(3) + 3w(2)$.

(c) If the vertex outside of $N_U[v]$ adjacent to w in $G[U]$ has U-degree one, branch on w. This represents a different case of the numerical optimisation problem: $d = 3, r_1 = r_2 = r_3 = 1$.

(d) If the vertex $x \notin N_U[v]$ that is adjacent to w in $G[U]$, has U-degree two or three, branch simultaneously on x and u. $\Delta_1 = 2w(3) + 3w(2) + \min\{w(2),\ 2w(1)\}$, $\Delta_2 = \Delta_3 = 2w(3) + 3w(2) + w(1) + \Delta w(3)$, $\Delta_4 = 2w(3) + 3w(2) + 2\Delta w(3)$. For Δ_1 we use the minimum extra reduction over the subcase where two edges with one endpoint in $N_U[v]$ are incident to the same vertex or to two different vertices outside $N_U[v]$.

For all other subcases with $d = 3, r_2 = 2, r_3 = 1$ we now have $d_2 = 4$.

5. $d = 3, r_3 = 3$. Because of Reduction Rule 6.1: $d_2 = 2$. In Section 6.6, we discuss variants of our algorithm for which we do not have a reduction rule dealing with this subcase. Therefore we consider the subcase with $d_2 = 0$ as if the reduction rule was not in our algorithm: remove it using two subproblems by branching on any vertex. $\Delta_1 = \Delta_2 = 4w(3)$

   The rest of this case is identical to Subcases 4(b-d), with $\Delta_1 = \Delta_2 = \Delta_3 = 4w(3) + w(2)$ in Subcase (b), and $\Delta_1 = 4w(3) + w(2) + \min\{w(2),\ 2w(1)\}$, $\Delta_2 = \Delta_3 = 4w(3) + w(2) + w(1) + \Delta w(3)$, $\Delta_4 = 4w(3) + w(2) + 2\Delta w(3)$ in Subcase (d). For all remaining subcases set $d_2 = 4$.

6. $d = 3, r_2 = 1, r_3 = 2$. As we handled Case 2 earlier, we have $d_2 = 3$. Suppose that the U-degree two neighbour of v is adjacent to another neighbour of v in $G[U]$. See Case 7 when this extra condition does not apply. Let T be the 3-clique (triangle) in $G[U]$ containing v.

   (a) A vertex $u \neq v$ in $G[U]$ is a neighbour of two vertices in $N_U(v)$. Branch on the neighbour of v incident to two edges with one endpoint in $N_U[v]$. In the subproblem where u is not removed also branch on u. $\Delta_1 = 3w(3) + 2w(2) + w(1), \Delta_2 = \Delta_3 = 3w(3) + 2w(2) + \Delta w(3)$.

   (b) In $G[U]$ a vertex $u \in T$ has a U-degree one neighbour: branch on u.

   (c) In the remaining subcase branch on both vertices in $N_U(T)$. $\Delta_1 = 3w(3) + 2w(2) + \min\{w(2) + w(1), 3w(1)\}$, $\Delta_2 = 3w(3) + 2w(2) + 2w(1) + \Delta w(3)$, $\Delta_3 = 3w(3) + 2w(2) + w(1) + 2\Delta w(3)$, $\Delta_4 = 3w(3) + 2w(2) + 3\Delta w(3)$. Notice that since $G[U]$ is simple, the minimum in the formula for Δ_1 does not need to consider $w(3)$: not all edges with only one endpoint drawn in Figure A.1 can be incident to the same vertex.

7. $d = 3, r_2 = 1, r_3 = 2$, again $d_2 = 3$ as we handled Case 2 earlier. Because of Case 6 suppose that the U-degree two neighbour of v is not adjacent to another neighbour of v in $G[U]$. Let T be the 3-clique in $G[U]$ with $v \in T$.

   (a) If any vertex in $N_U(T)$ has U-degree one, branch on its neighbour in T.

(b) If any vertex in $N_U(T)$ has U-degree three, we proceed to Case 8, where we let v be the vertex in T that is a neighbour to this U-degree three vertex. This case is illustrated in Figure A.1, Case 8 (not Case 7).

For all other subcases, $N_U(T)$ consists of vertices of U-degree two.

(c) A vertex in $u \in U$ is adjacent to two vertices in T. Branch on the vertex in T not adjacent to u. $\Delta_1 = 3w(3)+2w(2)+\Delta w(3)$, $\Delta_2 = 3w(3)+w(2)+\Delta w(2)$.

(d) Two vertices in U adjacent to T are adjacent to each other. Branch on a vertex $u \in T$ adjacent to one of these two vertices. When u is put in the vertex cover, branch on the other vertex adjacent to one of these two vertices. $\Delta_1 = \Delta_2 = 3w(3)+2w(2)+\Delta w(2)$, $\Delta_3 = 2w(3)+2w(2)+\Delta w(3)+\Delta w(2)$.

(e) The U-degree two neighbour of v is adjacent to a neighbour of a vertex in T in $G[U]$. Notice that this case is isomorphic to Subcase (d) as the triangle T is adjacent to three degree two vertices, two of which form a 4-cycle with T. Hence, this case can be dealt with similarly.

(f) Left is the subcase where no vertices in U neighbouring T are adjacent: branch on v.

Together with Case 6 this allows us to add an additional $2(\Delta w(2) - \Delta w(3))$ to Δ_{indep} for $d = 3, r_2 = 1, r_3 = 2$. This holds because the only remaining subcase is Subcase 7(f) where putting v in the independent set gives at least two vertices of U-degree two whose U-degree is reduced, in contrast to the original analysis, where we did not have the U-degrees of these vertices specified.

8. $d = 3, r_3 = 3$ with $d_2 = 4$ since we handled Case 5 earlier. We consider many subcases.

If not all vertices neighbouring the triangle T containing v in $G[U]$ are different, or they are adjacent to each other, then we again give alternative ways of branching. These specific cases are shown visually as Case 8' in Figure A.1: the first picture corresponds to a vertex being adjacent to two vertices of T; the other pictures on the top row correspond to T having two neighbours of degree two; and the pictures on the bottom row represent neighbourhoods of T with at most one degree two vertex. We will not explicitly state the recurrences representing these cases: they can be derived easily in a way similar to the above analysis.

We assume all three vertices that are neighbours of T to be different and non-adjacent.

(a) Again if any U-degree one vertex is a neighbour of a vertex $u \in T$, branch on u.

(b) Otherwise, if at most one of the vertices adjacent to T in $G[U]$ has degree two, branch on all three vertices neighbouring T in $G[U]$ simultaneously. In the worst case, one vertex has degree two resulting in:
$\Delta_1 = 5w(3)+w(2)+\min\{w(3)+w(2),\ w(3)+2w(1),\ 2w(2)+w(1),\ w(2)+3w(1), 5w(1)\}$, $\Delta_2 = \Delta_3 = 5w(3) + w(2) + \min\{w(2) + w(1),\ 3w(1)\} + 2\Delta w(3)$, $\Delta_4 = 5w(3) + w(2) + w(1) + 4\Delta w(3)$, $\Delta_5 = 5w(3) + w(2) +$

$\min\{2w(2),\ w(2)+2w(1),\ 4w(2)\}+\Delta w(3)$, $\Delta_6=\Delta_7=5w(3)+w(2)+2w(1)+3\Delta w(3)$, $\Delta_8=5w(3)+w(2)+5\Delta w(3)$.

The first four reductions correspond to the degree two vertex being put in the independent set and the last four reductions correspond to it being put in the vertex cover.

The only case that remains is the case where two vertices adjacent to T have degree two, and we can assume that these are not adjacent to v. Similar to the previous case, we add an additional $2(\Delta w(2)-\Delta w(3))$ to Δ_{indep} in the recurrence representing branching on v since at least two vertices at distance two from v now have degree two. We remark that this improved case remains a worst case in the associated numerical optimisation problem.

9. $d=4, r_4=4$. If all vertices in $N_U[v]$ are pairwise adjacent we have a clique that can be filtered out by Reduction Rule 6.1. It can also be removed using three subproblems by branching on any two vertices: $\Delta_1=\Delta_2=\Delta_3=5w(4)$.

   If there are two edges in $G[U]$ with only one endpoint in $N_U[v]$ then we branch on both vertices in $N_U[v]$ incident to these edges. $\Delta_1=5w(4)+\min\{w(2),2w(1)\}$, $\Delta_2=\Delta_3=5w(4)+w(1)+\Delta w(4)$, $\Delta_4=5w(4)+2\Delta w(4)$. This results in $d_2=4$ for all other subcases.

Having introduced all the alternative branching rules, we will now prove Theorem 6.9. This theorem states that there exists an algorithm that solves Edge Dominating Set in $\mathcal{O}(1.3226^n)$ time and polynomial space.

Proof of Theorem 6.9. Reconsider the numerical optimisation problem used to prove the running time of Theorem 6.7 and modify the values of Δ_{indep} as justified by the above case analysis. This gives:

$$\Delta_{\text{indep}} = w(d)+\sum_{i=1}^{d} r_i w(i)+d_2\Delta w(d)+\begin{cases} 2(\Delta w(2)-\Delta w(3)) & \text{if } d=3, r_2=1, r_3=2 \\ & \text{or } d=r_3=3 \\ 0 & \text{otherwise} \end{cases}$$

$$d_2 = \left[\left(\sum_{i=1}^{d}(i-1)r_i\right) \bmod 2\right]+\begin{cases} 4 & \text{if } d=3, r_2=2, r_3=1 \quad \text{(cases 1 and 4)} \\ 2 & \text{if } d=3, r_2=1, r_3=2 \quad \text{(case 2; also 6, 7)} \\ 2 & \text{if } d=r_2=3 \quad \text{(case 3)} \\ 2 & \text{if } d=r_3=3 \quad \text{(case 5; also 8)} \\ 4 & \text{if } d=r_4=4 \quad \text{(case 9)} \end{cases}$$

Furthermore, we add additional recurrences corresponding to the alternative branching strategies for all the subcases listed above. In order to keep the numerical optimisation problem finite, set $w(i)=1$ for $i\geq p$ for some $p\geq 4$ (see Remark A.1).

The solution (see Section 5.6) to this modified numerical optimisation problem leads to a running time of $\mathcal{O}(1.3226^n)$ using to following set of weights:

$$w(1)=0.779307 \qquad w(2)=0.920664 \qquad w(3)=0.997121 \qquad \forall_{i\geq 4}\ w(i)=1$$

The modified algorithm uses only polynomial space for the same reason as in Theorem 6.7. □

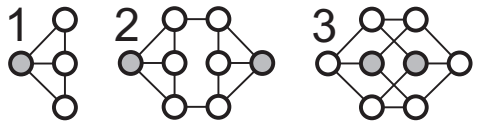


Figure A.2. Lower Bound Graphs.

Remark A.1. If we would have set $w(i) = 1$, for all $i \geq 3$, as in the proof of Theorem 6.7, then the recurrence relation for $d = r_4 = 4$ becomes independent of the weight function w: $w(i)$ for $i < 3$ does not occur in its formulas. The solution to this recurrence relation is close to $\alpha = 1.3247$ and is independent of w. Hence, $w(3)$ needs to be variable in order to get any solution below 1.3247.

The proof of the lower bound on the running time of Algorithm 6.3 is no longer valid after introducing the first alternative branching strategy. We prove different lower bounds for the algorithms in the improvement series (also see Table A.1):

Proposition A.13. *The worst case running time of the i-th algorithm in the improvement series is $\Omega(1.2968^n)$ if $i \leq 4$ and $\Omega(1.2753^n)$ if $i \geq 5$.*

Proof. If $i \leq 4$, consider the class of graphs consisting of l disjoint copies of Graph 2 in Figure A.2. In this case, the i-th algorithm in the series can branch on the rightmost grey vertex v. When v is put in the independent set, we are left with Subgraph 1 of Figure A.1 in $G[U]$ which generates two subproblems. When v is put in the vertex cover, the algorithm can branch on the leftmost grey vertex. This results in either a path on three vertices (two subproblems) or a cycle of length six (four subproblems) remaining in $G[U]$. Altogether this leads to a total of eight subproblems for each copy consisting of eight vertices. Since $8^l = 8^{n/8} > 1.2968^n$, these algorithms run in time $\Omega(1.2968^n)$.

If $i \geq 5$, then the i-th algorithm in the series uses alternative branching strategies 1 up to $i \geq 5$. In this case, the previous lower bound is no longer valid since the algorithm can no longer branch on the grey vertices: an alternative is introduced by alternative branching strategy 5. Now consider Graph 3 of Figure A.2 in which the algorithm can branch on one of the grey vertices. In the subproblem where this vertex is put in the independent set, a star shaped graph remains in $G[U]$ which generates two subproblems by branching on its centre vertex. In the other subproblem, the algorithm can branch on the other grey vertex resulting a cycle of length six in $G[U]$ or the removal of this copy of the graph from U. This gives a total sum of seven subproblems on a graph on eight vertices. Since $7^l = 7^{n/8} > 1.2753^n$, these algorithms run in time $\Omega(1.2753^n)$. □

A.4. Omitted Proofs for the Independent Set Algorithm

In Chapter 7, we claimed an $\mathcal{O}(1.08537^n)$-time and polynomial-space algorithm for Independent Set on graphs in which each connected component has average degree at most three. This algorithm is a branch-and-reduce algorithm for which the reduction rules can be found in Section 7.2.1 and the branching rules are described in the proofs of four lemmas which are formulated in Section 7.2.2. However, the proofs of only two of these lemmas are given in Chapter 7; the other two can be found in this appendix.

In Section A.4.1, we give the proof of Lemma 7.8 involving the branching on 3-regular graphs. Thereafter, we give the proof of Lemma 7.6 involving the branching on graphs with vertices of degree four in Section A.4.2. Finally, we give the details on the small separators rule due to Fürer [158] in Section A.4.3 for completeness.

Before giving the two omitted proofs, we first define what we call an *external edge*. When considering to remove a set of vertices $S \subseteq V$ from the graph in some branch, *external edges* are all edges incident to vertices in S that have not yet been considered as edges between vertices in S. That is, these edge are either edges between S and the rest of G, or edges between two vertices in S whose adjacency has not yet been considered.

We further clarify this concept by example. Consider a 3-regular graph and consider branching on a vertex v with a 3-cycle in $N[v]$. In the branch where v is taken in I and hence $N[v]$ is removed, there are 4 *external edges*: 4 vertices of degree three give us 12 endpoints from which 3 edges between v and $N(v)$ are formed; 4 external edges remain. However, some of these external edges can actually be the same edge if there is a second 3-cycle in $G[N[v]]$. That is, when counting external edges, we count unused endpoints of edges, while two of these endpoints can be of the same edges. If these edges are the same edges, we call this an *extra adjacency*: this lowers the actual number of edges removed. We note that, in the given example, we disregard the fact that if the extra adjacency exists, then the graph would be reduced by the domination rule.

In this way, we often construct simple counting arguments of the following form (the values used for the variables represent the above example): we remove $n' = 4$ vertices and $m' = 8$ edges among which there are $e = 4$ external edges. As these $e = 4$ external edges can lead to at most $e' = 1$ extra adjacency by reason X, this reduces the measure by $m' - n' - 1 = 3$ and gives $T(k - m' + n' + 1) = T(k - 3)$.

A.4.1. Proof of Lemma 7.8

The first proof that we give in this appendix is the proof of Lemma 7.8.

Lemma A.14 (before Lemma 7.8). *Let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$. If G is 3-regular and 3- and 4-cycle free, then we can branch such that $T(k)$ satisfies $T(k) \leq T_1(k-2) + T_3(k-5)$, or a better sequence of branchings exists. Here, we denote by $T_1(k)$ and $T_3(k)$ the recurrences from Cases 1 and 3 from Lemma 7.6 applied to an instance of measure k, respectively.*

Proof. Our reduction rules guarantee that no trees can be separated from G since we branch on a degree-three vertex in a graph of maximum degree three. We can also

assume that no other reduction rules than the degree one and two rules can be applied after branching. Namely, if such a reduction rule can be applied in the branch where we discard v, then we have the sufficient relation of $T(k) \leq T(k-4) + T(k-5)$. And, if such a reduction rule can be applied only in the branch where we take v in I, then we follow the proof below to obtain a $T_3(k-2)$-branch when discarding v to obtain a branching which corresponds to the sufficient relation of $T(k) \leq T_3(k-2) + T(k-7)$.

Let v be a vertex of G with neighbours x, y and z. We systematically consider the possible local neighbourhoods around v and observe what happens to this local neighbourhood when we branch on v. Because of 3- and 4-cycle freeness, v is the only common neighbour of any two vertices from the set $\{x, y, z\}$. By the same argument, there cannot be any adjacent vertices within $N(x)$, $N(y)$ or $N(z)$. There can, however, be at most six adjacencies between vertices from two different neighbourhoods. These adjacencies are important in the branch where v is discarded. Here, the remaining neighbours of x, y and z are folded to single vertices, and hence these adjacencies determine the newly formed local structure on which we will branch next. Notice that when there are two adjacencies between the same neighbourhoods, then the vertex folding after discarding v will lead to the removal of an extra edge since we do not allow double edges.

We first show that we can easily deal with cases involving more than three of these adjacencies between the neighbourhoods of x, y and z. If there are six, then we are looking at a connected component of constant size that can be solved in constant time. If there are five, then there are only two external edges out of $N^2[v]$ and hence there exists a small separator. Finally, if there are four such adjacencies, then these cannot all be between the same two neighbourhoods as this creates a 4-cycle. The alternative is that all three neighbourhoods are adjacent. In the branch where we discard v, the neighbourhoods of x, y and z are folded resulting in two degree-three vertices between which a double edge is removed that are in a 3-cycle with a degree-four vertex. This allows us to apply Lemma 7.6 Case 2 to obtain the following recurrence relations with a better branching behaviour than we are proving: $T(k) \leq T_3(k-5) + T(k-3-4) + T(k-3-6)$ or $T(k) \leq T_3(k-5) + 2T(k-3-8) + 2T(k-3-12)$.

We will now show that we can always obtain a $T_3(k-5)$ branch when taking v in I and discarding its neighbours. Removing $N(v)$ results in the creation of six degree-two vertices that will be folded. If any of these vertices are folded to degree-four vertices, we can apply Lemma 7.6 and we are done.

Hence, we need to consider only the case in which no degree-four vertices are created. This can happen only if the vertices that now have degree two form paths of even length: vertex folding replaces adjacent degree-two vertices by an edge. By the argument in the previous paragraph, we know that there are at most three adjacencies between the vertices in $N^2(v)$. Since the vertices in $N^2(v)$ are the new degree-two vertices, the only possible way for them to form even length paths is when they form three pairs of adjacent vertices (paths of length two connected to the rest of G). In this particular local structure, v lies on three 5-cycles, each pair of which overlaps in v and a different neighbour of v. In this very specific case, we decide not to branch on v: either this connected graph G has a vertex with a different local configuration, or G has no such vertex and thus each local configuration equals this specific case. If one draws this graph, one find that there is only one graph in which each vertex has this local configuration: the graph of the dodecahedron. We finish the argument by noting

that the dodecahedron has 20 vertices and can be solved in constant time.

What remains is the $T_1(k-2)$ branch when discarding v. In this branch, vertex folding results in three folded vertices. Because the graph is 3- and 4-cycle free before applying this lemma, a new 3- or 4-cycle created by folding after discarding v must involve the folded vertices. If such a 3- or 4-cycle is created, we can apply Lemma 7.6 Case 1. Notice that the folded vertices are of degree four unless folding results in the implicit removal of double edges between folded vertices. If all folded vertices are of degree four, we can apply Lemma 7.6 Case 1 obtaining our result of $T_1(k-2)$. If, on the other hand, additional edges are removed and we also consider the possibility that 3- or 4-cycles involving only degree-three vertices are created, we can apply the slightly worse Case 2 of Lemma 7.6 on the graph with measure $k-3$. This results in $T(k) \leq T_3(k-5)+T(k-3-4)+T(k-3-6)$ or $T(k) \leq T_3(k-5)+2T(k-3-8)+2T(k-3-12)$.

The only cases that remain are those in which no new 3- or 4-cycles are created by folding. We consider six different cases depending on the location of the three vertices that are the result of folding relative to each other in the graph obtained after discarding v. These three vertices will be of degree four unless there are double adjacencies between the neighbourhoods $N(x)$, $N(y)$, and $N(z)$: in these case folding results in double edges that will be removed. Hence, the following six cases arise:

- 0: Three non-adjacent degree-four vertices.
- 1: Three degree-four vertices only two of which are adjacent.
- 2a: Three degree-four vertices on a path of length three.
- 2b: Two adjacent degree-three vertices and a non-adjacent degree-four vertex.
- 3a: Two degree-three vertices adjacent to a degree-four vertex.
- 3b: Three degree-four vertices that form a 3-cycle.

Note that the numbers correspond to the number of edges between the vertices that are the result of the folding.

For each of these six cases, we will give efficient sequences of branchings based on the following reasoning. This reasoning is quite similar to exploiting mirrors. Let x', y' and z' be the result of folding the neighbourhoods of x, y and z to single vertices, respectively. If v is discarded, we know that we need to take at least two of the three neighbours of v in I: if we take only one, we could equally well have taken v which is done in the other branch already. This observation becomes slightly more complicated because we just folded the neighbours of v. The original vertex x is taken in the independent set if and only if the vertex x' is discarded in the reduced graph. Thus, the fact that we needed to take at least two vertices from $N(v)$ results in us being allowed to restrict ourselves to taking at most one vertex from the three degree-four vertices created by folding the neighbours of v. That is, taking any vertex in I from the three folded vertices allows us to discard the other two.

0: Three Non-Adjacent Degree-Four Vertices

We will perform a $T(k) \leq T_3(k-3) + T(k-9)$ branching after discarding v using the reasoning given above. This reasoning tells us that if we pick any vertex from the three folded vertices, then this allows us to discard the other two (see Figure A.3).

If we discard x', we remove 4 edges and 1 vertex. Moreover, at least one degree-four vertex remains in the graph after discarding x' giving $T_3(k-3)$, or at least one

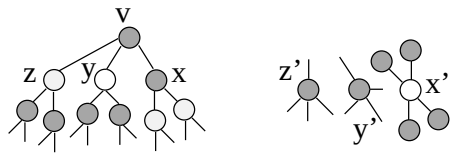


Figure A.3. The situation before and after discarding the degree-three vertex v and folding its neighbours. If x' is taken in the independent set (white), then this corresponds to discarding x (gray) and by a replacement argument taking y and z in the independent set. This then again corresponds to discarding y' and z'.

extra edge is removed by folding resulting in $T(k-4)$ (which in composition with the initial branching on v is even better). If, in the other branch, we take x' in I, then we can discard all four degree-three neighbours and both y' and z' resulting in the removal of 20 edges, from which 16 external edges, and 7 vertices. Because y' and z' are non adjacent and they can be adjacent to only a single neighbour of x' (or a 4-cycle would exist), there are at most two extra adjacencies and thus at least 12 external edges remain. If at most two trees are separated from the graph, this gives the $T(k-20+7+2+2) = T(k-9)$ branch.

We now show that at most two trees are separated from the graph when there are two extra adjacencies (i.e., when y' and z' are adjacent to neighbours of x'). In case where trees are separated from G, there must exist some very specific local structures. Because of the triangle and 4-cycle freeness, every tree vertex t can have neighbours that are distance three away from each other in $G[V \setminus \{t\}]$ only. The only size one trees that can be separated are adjacent to both y' and z' and a neighbour of x' that is not adjacent to either y' or z'. There can be at most one such tree, since two of these trees adjacent to the same two vertices also create a 4-cycle. And, it can exist only if y' and z' are adjacent to different neighbours of x'. This results in 9 remaining external edges that because of the small separators rule can form only one larger tree. If there is no size one tree, larger trees use more external edges and hence there can also be at most two of them.

In the above paragraph, we have assumed that there are two extra adjacencies. There can, however, also be fewer extra adjacencies leaving more external edges to form trees. In this case, the fewer extra adjacencies lead to extra edges compensating for the possible extra tree.

When considering this branching together with the initial branching on the vertex v, we obtain $T(k) \leq T_3(k-2-3) + T(k-2-9) + T_3(k-5) \leq 2T(k-8) + T(k-11) + 2T(k-12)$.

1: Three Degree-Four Vertices Only Two of Which Are Adjacent

Identical to the previous case, we try to perform a $T(k) \leq T_3(k-3)+T(k-9)$ branching after discarding v. In the worst case, this gives $T(k) \leq 2T(k-8)+T(k-11)+2T(k-12)$. Without loss of generality, assume that x' is adjacent to y' and that z' is not adjacent to x' or y'.

If we discard x', we remove 4 edges and 1 vertex. Now, either a degree-four vertex remains giving the $T_4(k-3)$, or an extra edge is removed by folding giving the in this

case slightly better $T(k-4)$. If we take x', we can also discard z' resulting in the removal of 17 edges from which 13 external edges and 6 vertices. There can be at most one extra adjacency, namely between z' and a degree-three neighbour of x'. Any tree vertex must again be adjacent to vertices that are distance at least 3 away from each other in this structure. This can only be both z' and any neighbour of x'. Hence, there cannot be any size one tree: it would need two neighbours of x' which causes a 4-cycle. Actually, there can be no tree at all since every tree leaf needs to be adjacent to z' in order to avoid 4-cycles in $N(x')$, but this also implies a 4-cycle. Hence, we have $T(k-17+6+1) = T(k-10)$.

If there is no extra adjacency, there can again be no 1-tree since it can be adjacent to at most one neighbour of x'. Larger trees remove enough external edges to prove $T(k-9)$.

2a: Three Degree-Four Vertices on a Path of Length Three

Again, we combine applying Lemma 7.6 to the branch where we take v with applying a $T(k) \leq T_3(k-3)+T(k-9)$ or better branch to the branch where we discard v. This again leads to $T(k) \leq 2T(k-8)+T(k-11)+2T(k-12)$. Let y' be adjacent to both x' and z', and let x' and z' be non-adjacent.

If we discard x', we remove 4 edges and 1 vertex while z' remains of degree four giving the $T_4(k-3)$. If we take x' in I, we can also discard z' resulting in the removal of 16 edges from which 11 external edges and 6 vertices. Notice that in the last branch there cannot be any extra adjacencies since they imply triangles or 4-cycles. There also cannot be any trees consisting of 1 or 2 vertices because tree leaves can be adjacent only to z' and a degree-three neighbour of x'. Finally, any larger tree decreases the number of external edges enough to obtain $T(k-16+6+1) = T(k-9)$.

2b: Two Adjacent Degree-Three Vertices and a Non-Adjacent a Degree-Four Vertex

We now have a graph with measure $k-3$ instead of $k-2$ due to the removal of a double edge. The graph has two adjacent degree-three vertices that are the result of folding, say y' and z', and a degree-four vertex x'. Furthermore, while y' and z' are adjacent, they are not adjacent to x'. Of these vertices x' cannot be involved in any triangle or 4-cycle, or we apply Lemma 7.6 Case 3 as discussed with the general approach.

We branch on x'. This leads to $T(k-3-3)$ when discarding x'. Similar to the above cases, we can still discard both y' and z' when taking x' in the independent set. Therefore, taking x' leads to removing 17 edges from which 12 external edges and 7 vertices. If there is an extra adjacency, this is between y' or z' and a neighbour of x'. In this case, there can be at most one tree since y' and z' together have only 3 external edges left and every tree leaf can be adjacent to at most one neighbour of x' or a 4-cycle with x' would exists. This leads to $T(k-3-17+7+1+1) = T(k-11)$. If there is no extra adjacency, every tree leaf can still be adjacent to no more than one neighbour of x', which together with the 4 external edges of y' and z' lead to at most two trees and $T(k-11)$. We obtain $T(k) \leq T_3(k-5)+T(k-6)+T(k-11)$.

3a: Two Degree-Three Vertices Adjacent to a Degree-Four Vertex

We again have a graph with measure $k-3$ with two degree-three vertices y', z' and a degree-four vertex x' which are all the result of folding. Furthermore, y' is adjacent to x' and z' while x' and z' are non-adjacent. Of these vertices, x' cannot be involved in any triangle or 4-cycle since we then apply Lemma 7.6 Case 2 as discussed with the general approach.

Similar to the previous case, we branch on x' giving $T(k-3-3)$ when discarding x', and we discard y' and z' when taking x'. In the second branch, this leads to the removal of 14 edges and 6 vertices obtaining $T(k) \leq T_3(k-5)+T(k-6)+T(k-11)$ as before unless trees are separated.

If trees are separated in the branch where x' is taken in I, observe that every tree leaf can again be adjacent to at most one neighbour of x', and hence all tree leaves must be adjacent to z'. Also observe that the third neighbour of y' cannot be adjacent to x' or any of its neighbours. Since z' has only two external edges, this means the only tree that can exist is a size two tree with both leaves connected to z' and a different neighbour of x' not equal to y' (or z dominates a tree vertex). Notice that this implies a triangle involving the tree and z'. In this case, we branch on y'. When taking y' in I, we remove 10 edges and 4 vertices: $T(k-6)$. When discarding y', the tree forms a triangle in which z' is taken in I because of the domination rule. Since we can take at most one of the folded vertices, this also results in x' being discarded. In total, this results in the removal of 11 edges and 4 vertices, and in this very specific structure no trees can exist: $T(k-7)$. This gives a much better recurrence when combined with branching on v than required.

3b: Three Degree-Four Vertices that Form a 3-Cycle

We treat this last subcase in an entirely different way. Reconsider the graph before branching on v, and let $N(x) = \{v, a, b\}$, $N(y) = \{v, c, d\}$ and $N(z) = \{v, e, f\}$. From the general proof of the $T_3(k-5)$ branch when taking v in I, we know that if there are three adjacencies between the three neighbourhoods $N(x)$, $N(y)$ and $N(z)$ then these are not pairwise distributed over the six possible vertices. This was proved by deciding to branch on another vertex which is always possible unless G equals the dedocahedron. Hence, we know that at least one vertex from $\{a, \ldots, f\}$ has a neighbour in both other neighbourhoods. Also, since there cannot be more than three adjacencies between these neighbourhoods, we know that there can be at most two such vertices with neighbours in both other neighbourhoods, and if there are two then they must be adjacent.

We begin with the case where one vertex has neighbours in both other neighbourhoods. Without loss of generality, let this vertex be a and let $N(a) = \{x, c, e\}$ and let d and f be adjacent. We now branch on x in stead of v. If we discard x, the neighbourhoods of a, b and v are folded to single vertices and we implicitly remove a double edge coming from the old edges $\{c, y\}$ and $\{e, z\}$. Furthermore, $N(b)$ is folded to a degree-four vertex giving a $T_3(k-3)$ branch. If we take x in I and discard $N(x)$, then c, y and e, z are adjacent degree-two vertices that are replaced by a single edge. Since these adjacent degree-two vertices form a path from d to f and the neighbours of b are non adjacent, a degree-four vertex must be created and we have

$T_3(k-5)$ in the other branch. Together this gives the sufficiently efficient recurrence of $T(k) \leq T_3(k-3) + T_3(k-5) \leq T(k-6) + T(k-8) + T(k-10) + T(k-12)$.

We end our proof with the last case involved. Without loss of generality let both a and c have neighbours in both other neighbourhoods: let $N(a) = \{x, c, e\}$ and $N(c) = \{y, a, f\}$. We obtain the same branching of $T(k) \leq T_3(k-3) + T_3(k-5)$ when branching on x. If we discard x, the edges $\{c, f\}$ and $\{e, z\}$ lead to a double edge between the folded neighbourhoods $N(a)$ and $N(v)$. Also, $N(b)$ will become a degree-four vertex giving the $T_3(k-3)$ branch. In the other branch, we take x in I again leading to two pairs of adjacent degree-two vertices that are replaced by single edges. In this case, these edges are incident to c and f. By the same argument as before, the neighbours of b will be folded to at least one degree-four vertex giving $T_3(k-5)$. □

A.4.2. Proof of Lemma 7.6

The proof in the previous section of this appendix and the running time of our algorithm in Chapter 7 rely on the fact that we can perform very efficient branchings on graphs with degree-four vertices. Moreover, these results require even more efficient branchings when every 3- and 4-cycle contains a degree-four vertex. In this section, we prove Lemma 7.6 which describes the branching rule that satisfies these requirements.

The proof of this lemma uses observations similar to those used in the proof of Lemmas 7.5 and 7.7. Namely, in a graph that contains a 3-cycle containing a degree-three vertex v, we can often branch on a neighbour $u \in N(v)$; this allows us to apply the domination rule on v the branch where we discard u. Also, in a graph containing a 4-cycle containing a degree-three vertex v, we can use the fact that the vertex opposite v on the cycle is a mirror of v. Actually, the only 4-cycles in which this does not happen in a graph of maximum degree four is a 4-cycle consisting of degree-four vertices only.

Lemma A.15 (before Lemma 7.6). *Let $T(k)$ be the number of subproblems generated when branching on a graph G with measure $k = m - n$. If G is of maximum degree four but not 3-regular, then we can branch such that $T(k)$ satisfies the following recurrence relations, or a better sequence of branchings exists.*

1. *if G has a degree-four vertex that is part of a 3- or 4-cycle also containing at least one degree-three vertex, and there are no 3- or 4-cycles containing only degree-three vertices, then $T(k) \leq T(k-5) + T(k-6)$ or $T(k) \leq 2T(k-8) + 2T(k-12)$.*

2. *if G has a degree-four vertex that is part of a 3- or 4-cycle also containing at least one degree-three vertex, and the constraint on the degree-three vertices from the previous case does not apply, then $T(k) \leq T(k-4) + T(k-6)$ or $T(k) \leq 2T(k-8) + 2T(k-12)$.*

3. *if the above cases do not apply, then $T(k) \leq T(k-3) + T(k-7)$.*

Proof. We start the proof by noticing that our reduction rules guarantee that no trees can be separated from G when we branch on a degree-three vertex. Furthermore, no trees are separated from G when discarding a vertex that by domination leads to a single degree-three vertex to be taken in I.

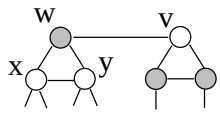


Figure A.4. Vertex v has degree three and is adjacent to a triangle with two degree-four vertices.

We first consider all possible 3-cycles containing both degree-three and degree-four vertices. Then, we consider all possible 4-cycles containing both degree-three and degree-four vertices in a 3-cycle free graph. In each subcase, Cases 1 and 2 from the statement of the lemma are proved. Thereafter, the remaining Case 3 will be proved.

3-Cycles with Two Degree-Four Vertices and a Degree-Three Vertex

Let x, y, w be a 3-cycle in the graph with $d(x) = d(y) = 4$ and $d(w) = 3$, also let v be the third neighbour of w. Notice that discarding v causes the domination rule to apply to w; this results in w being taken in the maximum independent set I.

If v is of degree four, discarding v and taking w leads to the removal of 11 edges and 4 vertices: $T(k-7)$. Taking v and removing $N[v]$ results in the removal of 3 edges incident to w and at least 8 more edges (in the worst case, v is in a triangle) and 5 vertices. If in this last case all neighbours of v are of degree three, then there are at most 6 external edges not incident to w and hence there can be at most one tree; the neighbours of w are fixed and cannot form a tree because of the small separators. Otherwise, any degree-four neighbours of v cause even more edges to be removed compensating for any possible tree. This results in $T(k-5)$: remove 11 edges, 5 vertices, and possibly separate one tree.

If v is of degree three (see Figure A.4), discarding v and taking w leads to the removal of of at least 10 edges and 4 vertices: $T(k-6)$. In this case, if v is not part of another 3-cycle or v has a degree-four neighbour (Case 1 of the lemma) taking v removes at least 9 edges and 4 vertices: $T(k-5)$. On the other hand, if v is part of a 3-cycle of degree-three vertices (Case 2 of the lemma) taking v removes 9 edges and 4 vertices $T(k-4)$.

3-Cycles with One Degree-Four Vertex and Two Degree-Three Vertices

When there is only one degree-four vertex on the 4-cycle, then the situation becomes a lot more complicated. Let x, a and b be the 3-cycle vertices with $d(x) = 4$ and $d(a) = d(b) = 3$, also let v be the third neighbour of a, and let w be the third neighbour of b (see Figure A.5).

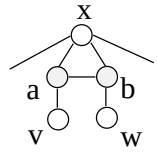


Figure A.5. Triangles with one degree-four vertex and two degree-three vertices.

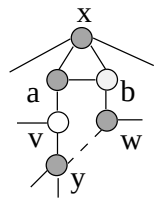


Figure A.6. Vertex v has a degree-four neighbour y.

By domination, we know that v and w are not adjacent to x, and that $v \neq w$. If v and w are adjacent, we can safely discard x reducing the graph. This follows from the fact that if we pick v we would also pick b, and if we discard v, its mirror b is also discarded which results in a being picked. In both cases a neighbour of x is in a maximum independent set, hence x can safely be discarded. So, we can assume that v and w are non-adjacent.

If v or w, say v, is of degree four, taking v removes at least 11 edges (when there is a 3-cycle in $N(v)$) and 5 vertices. Since there are 6 external edges, a tree can be separated. In total this leads to a branch with $T(k-5)$. If there are more external edges (less edges in $N(v)$) the number of edges removed increases. Discarding v and by domination taking a leads to the removal of 10 edges and 4 vertices: $T(k-6)$. This branching is sufficient to prove the lemma for this case. Thus, we can assume that v and w are of degree three from now on.

Consider the case where v or w, say v, has a degree-four neighbour y (see Figure A.6). Suppose that y does not form a 3-cycle with v, then taking v removes at least 10 edges and 4 vertices: $T(k-6)$. Discarding v and by domination taking a removes at least 9 edges and 4 vertices: $T(k-5)$. If, on the other hand, y does from a 3-cycle with v, then we branch on w. If, in this case, w has a degree-four neighbour or is not involved in a 3-cycle (Case 1 of the lemma), then taking w results as before in $T(k-5)$. Discarding w by domination results in taking b which again by dominating results in taking v. In total 15 edges are removed from which 7 external edges and 7 vertices. Because of the small separators rule, there can be at most 2 extra adjacencies in the worst case leaving 3 external edges: $T(k-6)$. Note that, in this case, separating a tree is beneficial over extra adjacencies since this uses more external edges. This leaves the case where w has only degree-three neighbours with which it forms a 3-cycle (now we are in Case 2 of the lemma). In this case, taking w leads to $T(k-4)$ only, which now is enough. So, we can assume that v and w are of degree three and that they have no degree-four neighbours.

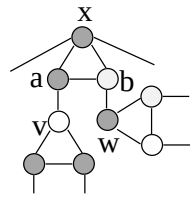


Figure A.7. The vertices v and w both have only degree three neighbours and are both part of a triangle.

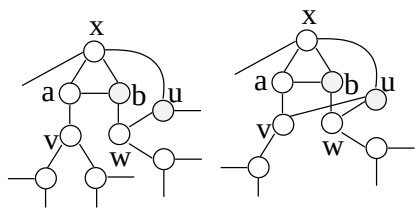


Figure A.8. Vertex w has a neighbour $u \neq a, b$ that is adjacent to x.

Suppose that v or w, say v, is part of a 3-cycle (see Figure A.7). Notice that we are now in Case 2 of the lemma. We branch on w. In the first branch, we take w in I. The worst case arises when w is also part of a 3-cycle; in this case, 8 edges and 4 vertices are removed leading to a branch with $T(k-4)$. In the other branch, we discard w and by domination b and v are put in the independent set I removing a total of at least 14 edges from which 6 external edges and 7 vertices. Because of the small separators rules, the external edges can form at most one extra adjacency or tree leading. This gives the required branch with $T(k-6)$. So, at this point, we can also assume that v and w are not part of any 3-cycle.

Suppose v or w, say w, has a neighbour $u \neq a, b$ that is adjacent to x (see Figure A.8). We branch on v. In the branch where we discard v, a is taken in I by domination. In this branch, we still have $T(k-5)$. In the branch where we take v, we have the situation that b becomes a degree-two vertex with neighbours x and w that will be folded. Notice that both x and w are adjacent to u, and hence this folding removes an additional edge. This gives a branch with $T(k-6)$. The only case in which the above does not holds is when v and w are both a neighbour of u. We reduce this exceptional case by noting that the tree separators fires when considering branching on u (without actually branching on u of course) because this would create the size two tree $\{a, b\}$. Hence, now we can also assume that v and w have no neighbours besides a and b that are adjacent x.

The rest of the analysis of this case consists of three more subcases depending on the number of vertices in $X = (N(w) \cup N(v)) \setminus \{a, b\}$. Because of the degrees of w and v we know that $2 \leq |X| \leq 4$.

If $|X| = 2$, v and w are adjacent to both vertices in X (see Figure A.9). Notice that if we take v in the independent set it is optimal to also pick w and vice versa. Hence, we branch by taking both v and w in I or discarding both. If we take both v and w in I, 11 edges are removed and 6 vertices: $T(k-5)$. If we discard both v

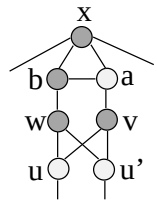


Figure A.9. Vertices v and w are adjacent to both u and u'.

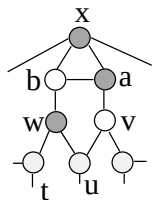


Figure A.10. The case $|X| = 3$.

and w, then we can take a in the independent set and remove 11 edges and 5 vertices: $T(k-6)$. In this special kind of branching we have to check that we do not separate trees from G. This cannot be the case when taking both v and w in I since there are only 4 external edges. This can also not be the case when discarding both v and w because, in this case, two tree leaves are formed; these are either adjacent resulting in a small separator, or adjacent to the only degree-two vertices (neighbours of x) also resulting in a small separator or even a constant size component. Also, there cannot be any extra adjacencies because this also results in the existence of a small separator.

If $|X| = 3$, let $u \in X$ be the common neighbour of v and w and let $t \in X$ be the third neighbour of w (see Figure A.10). We branch on t. If we take t in I, we also take b by domination. This results in the removal of 7 vertices and 15 edges if t has a degree-four neighbour or if there is no triangle involving t. Otherwise, only 14 edges are removed. We have $T(k-6)$ or $T(k-5)$ since there can be at most 8 external edges with this number of removed edges, and hence at most 2 extra adjacencies or trees. If we discard t, 3 edges and 1 vertex are removed and the folding of w results in the merging of vertices b and u. The new vertex can be discarded directly since it is dominated by a resulting in an additional removal of 4 edges and 1 vertex. This leads to $T(k-5)$ in total. Furthermore, we cannot separate trees in this way since there can be at most one vertex of degree less than two (adjacent to t and u, but no to w) which cannot become an isolated vertex. Depending on whether t is in a triangle, we are in Case 1 or 2 or the lemma and we have a good enough branching.

Finally, if $|X| = 4$, all neighbours of v and w are disjoint. We branch on v. If we take v in I, we remove 9 edges and 4 vertices, and if we discard v, we take a and again remove 9 edges and 4 vertices. This $T(k) \leq T(k-5)+T(k-5)$ branching is not good enough; therefore we inspect both branches more closely.

If we take a in I, w is folded resulting in the removal of an extra edge if its neighbours have another common neighbour. In this case, we are done. But if this is not the case, the folding of w results in a degree-four vertex. In the other branch where we take v, b is folded resulting in another degree-four vertex. We now apply the worst case of this lemma (Case 3) inductively to our two $T(k-5)$ branches and obtain $T(k) \leq 2T(k-8)+2T(k-12)$ as in the lemma.

We remark that $T(k) \leq T(k-5)+T(k-5)$ has a smaller solution than $T(k) \leq 2T(k-8)+2T(k-12)$. However, after the bad branch in a 3-regular graph of Lemma 7.8 the second recurrence gives a better solution when applied in the $T_1(k-2)$ branch. This is because it is a composition of three branchings that are all a lot better than the bad 3-regular graph branching with branching vector $(2,5)$.

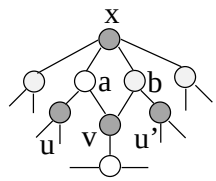


Figure A.11. Vertex x is a degree-four vertex that is a mirror of the degree-three vertex v.

4-Cycles in which a Degree-Four Vertex is a Mirror of a Degree-Three Vertex

Let x be the degree-four vertex that is a mirror of the degree-three vertex v, let a and b be their common neighbours, and let w be the third neighbour of v (see Figure A.11). If we branch on v and take v in I, we remove at least 9 edges and 4 vertices, and when we discard v and also x because it is a mirror of v, we remove 7 edges and 2 vertices: $T(k) \leq T(k-5) + T(k-5)$. This recurrence is not good enough. Below, we will show that we can always obtain $T(k-6)$ in one of both branches.

Notice that if we discard v and x, we cannot separate any trees since this would cause v and x to form a small separator. Also, any extra adjacency (a and b adjacent) results in triangles involving degree-three and degree-four vertices which are handled in the previous cases of this proof.

First assume that a, b or w is of degree four. In this case, we have $T(k-6)$ or better when taking v. Hence, we can assume that a, b and w are of degree three. We can also assume that a and b have no common neighbour outside this 4-cycle: if they would have such a neighbour, then the tree separators rule fires on a with possible size one tree b.

Let u and u' be the third neighbours of a and b, respectively. When discarding v and x, both a and b are taken in I and u and u' are discarded. This means that 13 edges form which 7 external edges and 6 vertices are removed. If u and u' are vertices of degree three, then the only possible extra adjacencies are those between u and u', or between u or u' and v. But there can be only one extra adjacency because otherwise there is a small separator. So we end up removing 12 edges from which 5 external edges and 6 vertices which cannot create trees: $T(k-6)$. Also, if u or u' is of degree four, then the extra edges that are removed compensate for any possible extra adjacencies or separated trees.

4-Cycles that Contain Degree-Three-and-Four Vertices while no Degree-Four Vertex is a Mirror of a Degree-Three Vertex

This can be the case only if the cycle consists of two degree-four vertices x, y and two degree-three vertices u, v with x and y not adjacent. There are no other adjacencies than the cycle between these vertices because then we would apply branchings from the analysis of 3-cycles.

Suppose that either u or v, say v, has a third degree-four neighbour z (see Figure A.12). Notice that this neighbour z cannot be adjacent to x or y. If we branch on v and take v in I, we remove 12 edges and 4 vertices, and if we discard v and its mirror u we remove 6 edges and 2 vertices. Thus, we have $T(k) \leq T(k-8) + T(k-4)$

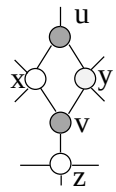


Figure A.12. Vertex v has a third degree-four neighbour z.

giving a better branching behaviour than required by this lemma. So, we can assume that u and v have no degree-four neighbours not on the 4-cycle.

Let w be the degree-three neighbour of v. It is not adjacent to u since that would imply that x and y are mirrors of w. Since w cannot be adjacent to x or y, taking v results in the removal of 11 edges and 4 vertices: $T(k-7)$. Discarding v and u leads to the removal of 6 edges and 2 vertices and hence a graph with measure $k-4$. In this case x and y will be folded. Because they are not adjacent to other created degree-two vertices (then there would be triangles involving degree-three and degree-four vertices), a vertex of degree at least four is created, or at least one additional edge is removed. It is also possible that a reduction rule different from the degree 0, 1 or 2 rules fires on the new graph. In this last case, we have $T(k-4-2)$ giving $T(k) \leq T(k-6)+T(k-7)$. If an additional edge is removed, then this leads to $T(k) \leq T(k-5)+T(k-7)$. Otherwise, if both options do not apply, we apply the worse case of this lemma inductively to our new degree-four vertex and obtain $T(k) \leq 2T(k-7)+T(k-11)$. These recurrences are sufficient to prove our running time.

A Degree-Four Vertex that is not Involved in any Triangle or 4-Cycle with any Degree-Three Vertex

We finally arrive at Case 3 of our lemma. Let x be this degree-four vertex. If all its neighbours are of degree three, branching on it results in $T(k) \leq T(k-7)+T(k-3)$. In this case, there cannot be any separates trees since any tree leaf is of degree at least three before branching and therefore must have at least two neighbours in $N(x)$ to become a leaf. But, in this last case, there would exist four cycles with degree-three and degree-four vertices on it; these are handled in other cases of the proof of this lemma.

If x has degree-four neighbours, the number of edges removed increases and there can still be no trees unless at least three neighbours of x are of degree four and every tree leaf vertex originally was a degree-four vertex. If x has three neighbours of degree four there are at least 13 edges removed, in which case there are 7 external edges. This can lead to at most one tree and $T(k-7)$ as required. If there are more external edges, there will also be more edges removed keeping this reduction. Finally, if x has four degree-four neighbours, we remove at least 12 edges from which 4 external edges; this again gives us a branch with $T(k-7)$. Here, any tree implies more external edges, and hence more edges are removed also keeping this reduction. □

A.4.3. Details on the Small Separators Rule

In the last section of this appendix, we give the details of the small separators rule that is due to Fürer [158]. We include the full proof of this rule for completeness.

Lemma A.16 (before Lemma 7.2, [158]). *If the graph contains a vertex separator S of size one or two, then we can simplify the instance in the following way. We recursively solve two or four subproblems that correspond to the smallest associated connected component C with each possible combination of vertices from S added. Given the solutions computed by the recursive calls, we can remove S and C from the graph and adjust the remaining graph to obtain an equivalent instance. If C has size at most some constant c, then this operation can be done in polynomial time.*

Proof. We first consider the case where G has a separator of size one. Let v be an articulation point of G, and let $C \subset V$ be the vertices of the smallest associated connected component (vertices in C have edges only to v or to other vertices in C). If the algorithm finds such an articulation point v, it recursively computes the maximum independent sets $I_{\not v}$ in the subgraph $G[C]$ and I_v in the subgraph $G[C \cup \{v\}]$. Notice that $|I_v|$ can be at most one larger than $|I_{\not v}|$, and if this is the case then $v \in I_v$. If the sizes of these maximum independent sets are the same, the algorithm recursively computes the maximum independent set I in $G[V \setminus (C \cup \{v\})]$ and returns $I \cup I_v$. This is correct since taking v in the independent set restricts the possibilities in $G[V \setminus (C \cup \{v\})]$ more, while it does not increase the maximum independent set in $C \cup \{v\}$. Otherwise, if $|I_v| = 1 + |I_{\not v}|$, then the algorithm computes the maximum independent set I in $G[V \setminus C]$ and returns $I \cup (I_v \setminus \{v\})$. This is also correct since adding v to C increases the size of the maximum independent set in $G[C]$ by one; this choice is now left to the recursive call on $G[V \setminus C]$.

Now, we will deal with separators in G of size two. If the algorithm finds such a separator $\{u, v\}$ with smallest association connected component $C \subset V$, then it computes a maximum independent set in the four subgraphs induced by C and any combination of vertices from the separator. Let $I_{\not v,\not u}$ be the computed maximum independent set in $G[C]$, let $I_{v,\not u}$ be the computed maximum independent set in $G[C \cup \{v\}]$, let $I_{\not v,u}$ be the computed maximum independent set in $G[C \cup \{u\}]$, and let $I_{v,u}$ be the computed maximum independent set in $G[C \cup \{u, v\}]$.

We now consider the following five possible cases. Correctness of the procedure in each case follows from first deciding whether discarding u or v is optimal: this is the case if adding them to C does not increase the size of the maximum independent set in $G[C]$. Otherwise, each case lets the recursive call on the larger component decide on their membership of the computed maximum independent set.

1. $|I_{v,u}| = |I_{\not v,\not u}| + 2$, and hence $|I_{v,\not u}| = |I_{\not v,u}| = |I_{\not v,\not u}| + 1$. The algorithm now computes a maximum independent set in $G[V \setminus C]$ and returns $I \cup J$ where J is the set from $\{I_{\not v,\not u}, I_{v,\not u}, I_{\not v,u}, I_{v,u}\}$ that agrees with I on u and v.

   This is correct as either taking u or v in the maximum independent set I does not influence the size of $I \cap C$, and the recursive call on $G[V \setminus C]$ now decides whether it is beneficial to take u or v in I.

2. $|I_{v,\not u}| = |I_{\not v,u}| = |I_{v,u}| = |I_{\not v,\not u}| + 1$. Let G' be $G[V \setminus C]$ with an extra edge added between u and v. Similar to the previous case, the algorithm computes a

maximum independent set in G' and returns $I \cup J$, where J is one of the four possible independent sets that agree on u and v.

Correctness follows from the same reasoning as above, only the extra edge enforces that the recursive call can only take u or v in the maximum independent set and not both. This is necessary since taking both u and v now does restrict the number of vertices that we can take from C.

3. $|I_{v,\not u}| = |I_{\not v,\not u}|$ and $|I_{\not v,u}| = |I_{v,u}| = |I_{\not v,\not u}| + 1$ (and the symmetric case). We recursively compute the maximum independent set I in $G[V \setminus (C \cup \{v\}]$ and return $I \cup J$, where J is the independent set from $\{I_{\not v,\not u}, I_{\not v,u}\}$ that agrees on u.

   In this case, v can safely be discarded since it does not help increasing the size of the independent set in $C \cup \{v\}$.

4. $|I_{\not v,u}| = |I_{v,\not u}| = |I_{\not v,\not u}|$ and $|I_{v,u}| = |I_{\not v,\not u}| + 1$. Let G' be $G[V \setminus C]$ with u and v merged into a single vertex w. The algorithm makes a recursive call on G' returning I. If $w \in I$, then we return $(I \setminus \{w\}) \cup I_{v,u}$, otherwise, we return $I \cup I_{\not v,\not u}$.

   Correctness here is similar to Case 3. Instead of adding an edge to make sure that not both u and v are chosen to increase the size of the maximum independent set, we here merge both vertices to ensure that both need to be chosen to increase the size of the maximum independent set by only one.

5. $|I_{v,u}| = |I_{\not v,u}| = |I_{v,\not u}| = |I_{\not v,\not u}|$. Now it is safe to use $I_{\not v,\not u}$. We make a recursive call on $G[V \setminus (C \cup \{u, v\})]$ resulting in I and return $I \cup I_{\not v,\not u}$.

We complete the proof by noticing that if the smallest associated connected component C has size at most a constant c, then the recursive calls on $G[C]$ with one or two vertices can all be performed in constant time. In this case, the reduction rule can be executed in polynomial time. □

B List of Problems

This appendix contains definitions of most computational problems used in this thesis. The appendix is divided in three sections: standard problems (pages 321-331), parameterised problems (pages 331-333), and counting problems (pages 333-334). The problems in each section are given in alphabetical order.

Standard Problems

Binary Knapsack

Input: A set of items, numbered $1, 2, \ldots, n$ with, for each item i, two positive integers v_i, w_i, where v_i is the value of item i and w_i is the weight of item i, and two integers k, l.

Question: Does there exist a subset of the items with total weight at most l and total value at least k?

This problem is called Binary Knapsack because each item may be taken at most once.

Capacitated Dominating Set

Input: A graph $G = (V, E)$, a capacity function $c : V \to \mathbb{N}$, and an integer $k \in \mathbb{N}$.

Question: Does there exist a capacitated dominating set $D \subseteq V$ in G of size at most k respecting the capacities c?

In this problem, vertices $v \in D$ can dominate a number of neighbours that is at most its capacity $c(v)$. Formally, a *capacitated dominating set* is a vertex subset $D \subseteq V$ such that there exists a *domination function* f for D respecting the capacities c. This domination function $f : V \setminus D \to D$ assigns to each vertex $v \in V \setminus D$ the vertex $u \in N[v] \cap D$ that dominates it, with the constraint that there exist at most $c(v)$ vertices u with $f(u) = v$.

Chromatic Index
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exists an edge colouring of G using at most k colours?

An *edge colouring* of a graph G is an assignment of colours to the edges of G such that no two edges that are incident to the same vertex have the same colour.

***k*-Colouring**
Input: A graph $G = (V, E)$.
Question: Does there exists a colouring of the vertices of G using at most k colours such that no two vertices of the same colour are adjacent?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Graph Colouring.

Connected Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a connected dominating set $D \subseteq V$ in G of size at most k?

A *connected dominating set* is a connected vertex subset $D \subseteq V$ such that $N[D] = V$.

Connected Red-Blue Dominating Set
Input: A graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertices $\mathcal{R}$ and blue vertices $\mathcal{B}$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a connected red-blue dominating set $D \subseteq \mathcal{R}$ in G of size at most k?

A *connected red-blue dominating set* is a connected subset $D \subseteq \mathcal{R}$ such that $N(D) \supseteq \mathcal{B}$.

***k*-Dimensional Matching**
Input: A series of mutually disjoint sets $U_1, U_2, \ldots, U_k$ of equal size and a collection of sets $\mathcal{S}$ in which each set contains exactly one element from each of the sets U_i.
Question: Does there exist a collection $\mathcal{C} \subseteq \mathcal{S}$ such that each element from $\bigcup_{1 \leq i \leq k} U_i$ is contained in exactly one set in $\mathcal{C}$?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.

Directed Dominating Set
Input: A directed graph $G = (V, A)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set $D \subseteq V$ in G of size at most k?

Edges (or *arcs*) in a directed graph are ordered pairs of vertices $(u, v) \in A$. A *dominating set* $D \subseteq V$ in a directed graph is a vertex subset such that, for every $v \in V$, either $v \in D$, or there exists a vertex $d \in D$ such that $(d, v) \in A$.

Disjoint Connected Subgraphs
Input: A graph $G = (V, E)$ and mutually disjoint non-empty sets $Z_1, Z_2, \ldots, Z_k \subseteq V$.
Question: Do there exist mutually vertex-disjoint connected subgraphs $G_1, G_2, \ldots, G_k$ of G (with $G_i = (V_i, E_i)$) such that Z_i is contained in V_i for every $1 \leq i \leq k$?

See also: k-Disjoint Connected Subgraphs.

k-Disjoint Connected Subgraphs
Input: A graph $G = (V, E)$ and mutually disjoint non-empty sets $Z_1, Z_2, \ldots, Z_k \subseteq V$.
Question: Do there exist mutually vertex-disjoint connected subgraphs $G_1, G_2, \ldots, G_k$ of G (with $G_i = (V_i, E_i)$) such that Z_i is contained in V_i for every $1 \leq i \leq k$?

The difference between this problem and Disjoint Connected Subgraphs is that, in this problem, the integer k is part of the problem description instead of part of the input. This leads to a different problem for every $k \in \mathbb{N}$.
See also: Disjoint Connected Subgraphs.

Distance-r Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a distance-r dominating set $D \subseteq V$ in G of size at most k?

A *distance-r dominating set* is a vertex subset $D \subseteq V$ such that, for every vertex $v \in V$, there exists a vertex $d \in D$ that lies at distance at most r from v.

Domatic Number
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Can G be partitioned into at most k dominating sets?

See also: k-Domatic Number and Dominating Set.

k-Domatic Number
Input: A graph $G = (V, E)$.
Question: Can G be partitioned into at most k dominating sets?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Domatic Number and Dominating Set.

Dominating Clique
Input: A graph $G = (V, E)$.
Question: Does there exist a dominating clique $D \subseteq V$ in G?

A *dominating clique* is a vertex subset $D \subseteq V$ that is both a dominating set and a clique, i.e., $N[D] = V$ and $\{u, v\} \in E$ for every two distinct vertices $u, v \in D$.

Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set $D \subseteq V$ in G of size at most k?

A *dominating set* is a vertex subset $D \subseteq V$ such that $N[D] = V$.

p-Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a p-dominating set $D \subseteq V$ in G of size at most k?

A *p-dominating set* is a vertex subset $D \subseteq V$ such that every vertex $v \in V \setminus D$ has at least p neighbours in D, i.e, $|N(v) \cap D| \geq p$.

Notice that p is fixed, and that this leads to a different problem for every $p \in \mathbb{N}$. Also notice that this problem is different form the parameterised problem k-Dominating Set.

Edge Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of size at most k?

An *edge dominating set* is an edge subset $D \subseteq E$ such that, for every edge $e \in E$, there exists an edge $d \in D$ that has an endpoint in common with e.

Exact Cover by k-Sets
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$ in which each set is of size exactly k.
Question: Does there exist a collection $\mathcal{C} \subseteq \mathcal{S}$ such that each element from $\mathcal{U}$ is contained in exactly one set in $\mathcal{C}$?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.

Exact Hitting Set
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$.
Question: Does there exist a subset of $\mathcal{U}$ that contains exactly one element from every $S \in \mathcal{S}$?

Exact Satisfiability
Input: A set of clauses C using a set of variables X.
Question: Does there exist a truth assignment to the variables in X such that each clause in C contains exactly one literal set to true?

See also: Exact k-Satisfiability.

Exact k-Satisfiability
Input: A set of clauses C with each clause of size at most k using a set of variables X.
Question: Does there exist a truth assignment to the variables in X such that each clause in C contains exactly one true literal?

See also: Exact Satisfiability, Maximum Exact k-Satisfiability.

Feedback Vertex Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a feedback vertex set $F \subseteq V$ in G of size at most k?

A *feedback vertex set* is a vertex subset $F \subseteq V$ such that $G[V \setminus F]$ does not contain any cycle, i.e., such that $G[V \setminus F]$ is a collection of trees.

Graph Colouring
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exists a colouring of the vertices of G using at most k colours such that no two vertices of the same colour are adjacent?

See also: k-Colouring.

Hamiltonian Cycle
Input: A graph $G = (V, E)$.
Question: Does G have a Hamiltonian cycle?

A *Hamiltonian cycle* is a set of edges in G that forms a cycle that visits every vertex exactly once.
See also: Travelling Salesman Problem.

Hitting Set
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a subset of $\mathcal{U}$ of size at most k that contains at least one element from every $S \in \mathcal{S}$?

This problem is equivalent to Set Cover.
See also: k-Hitting Set.

***k*-Hitting Set**
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$ with each set of size at most k and an integer $l \in \mathbb{N}$.
Question: Does there exist a subset of $\mathcal{U}$ of size at most l that contains at least one element from every $S \in \mathcal{S}$?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Hitting Set.

Hypergraph 2-Colouring
Input: A hypergraph $H = (Q, \mathcal{S})$.
Question: Does H have a 2-colouring?

A *2-colouring* of H is a partition of Q into two sets Q_1 and Q_2 such that each set $S \in \mathcal{S}$ contains at least one element from Q_1 and at least one element from Q_2.

2-Hypergraph 2-Colouring
Input: A 2-hypergraph $H = (Q, \mathcal{L}, \mathcal{R})$.
Question: Does H have a 2-colouring?

A *2-colouring* of H is a partition of Q into two sets Q_l and Q_r such that each hyperedge $L \in \mathcal{L}$ contains an element from Q_l and each hyperedge $R \in \mathcal{R}$ contains an element from Q_r.

Independent Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an independent dominating set $D \subseteq V$ in G of size at most k?

An *independent dominating set* is a vertex subset $D \subseteq V$ that is both an independent set and a dominating set, i.e., $N[D] = V$ and $\{u, v\} \notin E$ for every two distinct vertices $u, v \in D$.

Independent Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an independent set $I \subseteq V$ in G of size at least k?

An *independent set* is a vertex subset $I \subseteq V$ such that $\{u, v\} \notin E$ for every two distinct vertices $u, v \in I$.
This problem is equivalent to Vertex Cover.

Induced p-Regular Subgraph
Input: A graph $G = (V, E)$.
Question: Does there exist an induced subgraph H of G that is p-regular?

For a vertex set $V' \subseteq V$, $H = (V', (V' \times V') \cap E)$ is the subgraph induced by V'.
A graph is p-regular if all vertices $v \in G$ have degree p.
Notice that p is fixed, and that this leads to a different problem for every $p \in \mathbb{N}$.

Matrix Dominating Set
Input: An $n \times m$ 0-1 matrix and an integer $k \in \mathbb{N}$.
Question: Does there exist a set S of 1-entries in M of size at most k such that every 1-entry is on the same row or column as an entry in S?

Maximum Cut
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a partitioning of the vertices of G into two sets that induce a cut of size at least k?

The *cut* induced by a partitioning of V into two sets X_1, X_2 is the set of edges with one endpoint in X_1 and the other endpoint in X_2.

Maximum Exact k-Satisfiability
Input: A set of clauses C with each clause of size at most k using a set of variables X and an integer $l \in \mathbb{N}$.
Question: Does there exist a truth assignment to the variables in X such that at least l clauses in C contain exactly one true literal?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Exact Satisfiability, Exact k-Satisfiability.

Maximum k-Satisfiability
Input: A set of clauses C with each clause of size at most k using a set of variables X.
Question: Does there exist a truth assignment to the variables in X that satisfies at least k clauses in C?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Satisfiability, k-Satisfiability.

Maximum Triangle Packing
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Do there exists mutually disjoint 3-element subsets $S_1, S_2, \ldots, S_k$ of V such that for each S_i the graph $G[S_i]$ is a triangle?

Minimum Clique Partition
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Can G be partitioned into at most k cliques?

A clique is a vertex subset $C \subseteq V$ such that $\{u, v\} \in E$ for every two distinct vertices $u, v \in C$.

Minimum Independent Edge Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist an independent edge dominating set $D \subseteq E$ in G of size at most k?

An *independent edge dominating set* is an edge subset $D \subseteq E$ such that, for every edge $e \in E$, there exists an edge $d \in D$ that has an endpoint in common with e, and such that no two distinct edges $d, e \in D$ have an end point in common.
This problem is equivalent to Minimum Maximal Matching.

Minimum Maximal Matching
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of size at most k?

A *matching* is an edge subset $M \subseteq E$ such that no two distinct edges $d, e \in M$ have an end point in common. A *maximal matching* is an inclusion-wise maximal matching, i.e., a matching $M \subseteq E$ such that there exists no superset L, $M \subset L \subseteq E$, that also is a matching.
This problem is equivalent to Minimum Independent Edge Dominating Set.

Minimum Weight Dominating Set
Input: A graph $G = (V, E)$, a weight function $\omega : V \to \mathbb{R}_+$, and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist a dominating set $D \subseteq V$ in G of weight at most k?

The weight of a dominating set D (see Dominating Set) is defined as $\omega(D) = \sum_{v \in D} \omega(v)$.

Minimum Weight Edge Dominating Set
Input: A graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{R}_+$, and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of weight at most k?

The weight of an edge dominating set D (see Edge Dominating Set) is defined as $\omega(D) = \sum_{e \in D} \omega(e)$.

Minimum Weight Independent Edge Dominating Set
Input: A graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{R}_+$, and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist an independent edge dominating set $D \subseteq E$ in G of weight at most k?

The weight of an independent edge dominating set D (see Minimum Independent Edge Dominating Set) is defined as $\omega(D) = \sum_{e \in D} \omega(e)$.
This problem is equivalent to Minimum Weight Maximal Matching.

Minimum Weight Maximal Matching
Input: A graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{R}_+$, and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of weight at most k?

The weight of a (maximal) matching M (see Minimum Maximal Matching) is defined as $\omega(M) = \sum_{e \in M} \omega(e)$.
This problem is equivalent to Minimum Weight Independent Edge Dominating Set.

***H*-Minor Containment**
Input: A graph $G = (V, E)$.
Question: Does G contain the graph H as a minor?

The graph H is a *minor* of a graph G if it can be obtained from G by a series of vertex or edge deletions and edge contractions.
Notice that the graph H is fixed, and that this leads to a different problem for every graph H.

Not-All-Equal Satisfiability
Input: A set of clauses C using a set of variables X.
Question: Does there exist a truth assignment to the variables in X such that every clause in C contains at least one literal set to true and at least one literal set to false?

Partial Dominating Set
Input: A graph $G = (V, E)$, an integer $t \in \mathbb{N}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a partial dominating set $D \subseteq V$ in G of size at most k that dominates at least t vertices?

Any vertex subset $D \subseteq V$ is a *partial dominating set*. A partial dominating set D dominates all vertices in $N[D]$. I.e., the problem asks whether there exists a vertex subset $D \subseteq V$ such that $|D| \leq k$ and $|N[D]| \geq t$.

Partial Red-Blue Dominating Set
Input: A bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertices $\mathcal{R}$ and blue vertices $\mathcal{B}$, an integer $t \in \mathbb{N}$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a partial red-blue dominating set $D \subseteq \mathcal{R}$ in G of size at most k that dominates at least t vertices in $\mathcal{B}$?

Any vertex subset $D \subseteq \mathcal{R}$ is a *partial red-blue dominating set*. A partial red-blue dominating set D dominates all vertices in $N(D) \cap \mathcal{B}$. I.e., the problem asks whether there exists a vertex subset $D \subseteq \mathcal{R}$ such that $|D| \leq k$ and $|N(D) \cap \mathcal{B}| \geq t$.

Partition Into *l*-Cliques
Input: A graph $G = (V, E)$.
Question: Can the vertices of G be partitioned into cliques of size l?

A clique is a vertex subset $C \subseteq V$ such that $\{u, v\} \in E$ for every two distinct vertices $u, v \in C$.

Notice that l is fixed, and that this leads to a different problem for every $l \in \mathbb{N}$.

Partition Into Triangles
Input: A graph $G = (V, E)$.
Question: Can V be partitioned into 3-element sets $S_1, S_2, \ldots, S_{|V|/3}$ such that for each S_i the graph $G[S_i]$ is a triangle?

Perfect Code
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a perfect code $D \subseteq V$ in G?

A *perfect code* is a vertex subset $D \subseteq V$ that is both a perfect dominating set and an independent set. That is, every vertex $v \in V \setminus D$ has exactly one neighbour in D, and $\{u, v\} \notin E$ for every two distinct vertices $u, v \in D$. A perfect code is also known as an *efficient dominating set*.

Perfect Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a perfect dominating set $D \subseteq V$ in G of size at most k?

A *perfect dominating set* is a vertex subset such that every vertex $v \in V \setminus D$ has exactly one neighbour in D.

Red-Blue Dominating Set
Input: A bipartite graph $G = (\mathcal{R} \cup \mathcal{B}, E)$ with red vertices $\mathcal{R}$ and blue vertices $\mathcal{B}$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a red-blue dominating set $D \subseteq \mathcal{R}$ in G of size at most k?

A *red-blue dominating set* is a subset $D \subseteq \mathcal{R}$ such that $N(D) \supseteq \mathcal{B}$.

Satisfiability
Input: A set of clauses C using a set of variables X.
Question: Does there exist a truth assignment to the variables in X that satisfies all clauses in C?

A clause is satisfied if it contains at least one literal that is set to $True$.
See also: k-Satisfiability.

***k*-Satisfiability**
Input: A set of clauses C with each clause of size at most k using a set of variables X.
Question: Does there exist a truth assignment to the variables in X that satisfies all clauses in C?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Satisfiability, Maximum k-Satisfiability.

Set Cover
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a set cover $\mathcal{C} \subseteq \mathcal{S}$ of size at most k?

A *set cover* $\mathcal{C}$ is a subset of $\mathcal{S}$ such that $\bigcup_{S \in \mathcal{C}} S = \mathcal{U}$.
This problem is equivalent to Hitting Set.
See also: k-Set Cover.

k-Set Cover
Input: A multiset of sets $\mathcal{S}$ over a universe $\mathcal{U}$ in which each set is of size at most k and an integer $l \in \mathbb{N}$.
Question: Does there exist a set cover $\mathcal{C} \subseteq \mathcal{S}$ of size at most l?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.
See also: Set Cover.

Steiner Tree
Input: A graph $G = (V, E)$, a set of terminals $T \subseteq V$, and an integer $k \in \mathbb{N}$.
Question: Does there exist a Steiner tree in G connecting all terminals in T using at most k edges?

A *Steiner tree* is a connected set of edges that connects all vertices in T.

Strong Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a strong dominating set $D \subseteq V$ in G of size at most k?

A *strong dominating set* is a vertex subset $D \subseteq V$ such that every vertex $v \in V \setminus D$ has a neighbour in D of equal or larger degree.

Strong Stable Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a strong stable set $I \subseteq V$ in G of size at least k?

A *strong stable set* is a vertex subset $I \subseteq V$ such that the distance in G between any pair of vertices from I is at least two.

Subset Sum
Input: A set of integers $a_1, a_2, \ldots, a_n$ and an integer b.
Question: Does there exist a subset of the integers $a_1, a_2, \ldots, a_n$ whose sum equals b?

Total Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a total dominating set $D \subseteq V$ in G of size at most k?

A *total dominating set* is a vertex subset $D \subseteq V$ such that every $v \in V$ (i.e., also those in D) is adjacent to a vertex in D.

Total Perfect Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a total perfect dominating set $D \subseteq V$ in G?

A *total perfect dominating set* is a vertex subset $D \subseteq V$ such that every vertex $v \in V$ (i.e., also those in D) has exactly one neighbour in D.

Travelling Salesman Problem
Input: A graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{R}_+$, and a non-negative real number $k \in \mathbb{R}_+$.
Question: Does G have a Hamiltonian cycle whose total weight is at most k?

The weight of a Hamiltonian cycle C (see Hamiltonian Cycle) is defined as $\omega(C) = \sum_{e \in C} \omega(e)$.
See also: Hamiltonian Cycle.

Vertex Cover
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a vertex cover $C \subseteq V$ in G of size at most k?

A *vertex cover* is a vertex subset $C \subseteq V$ such that every edge $e \in E$ has an endpoint in C.
This problem is equivalent to Independent Set.

Weak Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a weak dominating set $D \subseteq V$ in G of size at most k?

A *weak dominating set* is a vertex subset $D \subseteq V$ such that every vertex $v \in V \setminus D$ has a neighbour in D of equal or smaller degree.

Parameterised Problems

***k*-Dominating Set**
Input: A graph $G = (V, E)$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set $D \subseteq V$ in G of size at most k?

A *dominating set* is a vertex subset $D \subseteq V$ such that $N[D] = V$.
Notice that this problem is different form the problem p-Dominating Set. Also notice that the parametric dual of this problem is k-Nonblocker.
See also: Dominating Set.

***k*-Edge Dominating Set**
Input: A graph $G = (V, E)$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of size at most k?

An *edge dominating set* is an edge subset $D \subseteq E$ such that, for every edge $e \in E$, there exists an edge $d \in D$ that has an endpoint in common with e.
See also: Edge Dominating Set.

***k*-Independent Set**
Input: A graph $G = (V, E)$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist an independent set $I \subseteq V$ in G of size at least k?

An *independent set* is a vertex subset $I \subseteq V$ such that $\{u, v\} \notin E$ for every two distinct vertices $u, v \in I$.
Notice that the parametric dual of this problem is k-Vertex Cover.
See also: Independent Set.

***k*-Minimum Maximal Matching**
Input: A graph $G = (V, E)$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of size at most k?

A *matching* is an edge subset $M \subseteq E$ such that no two distinct edges $d, e \in M$ have an end point in common. A *maximal matching* is an inclusion-wise maximal matching, i.e., a matching $M \subseteq E$ such that there exists no superset L, $M \subset L \subseteq E$, that also is a matching.
See also: Minimum Maximal Matching.

***k*-Minimum Weight Edge Dominating Set**
Input: A graph $G = (V, E)$ and a weight function $\omega : E \to \mathbb{R}_{\geq 1}$.
Parameter: A non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist an edge dominating set $D \subseteq E$ in G of weight at most k?

The weight of an edge dominating set D (see k-Edge Dominating Set) is defined as $\omega(D) = \sum_{e \in D} \omega(e)$.
See also: Minimum Weight Edge Dominating Set.

***k*-Minimum Weight Maximal Matching**
Input: A graph $G = (V, E)$ and a weight function $\omega : E \to \mathbb{R}_{\geq 1}$.
Parameter: A non-negative real number $k \in \mathbb{R}_+$.
Question: Does there exist a maximal matching $M \subseteq E$ in G of weight at most k?

The weight of a (maximal) matching M (see k-Minimum Maximal Matching) is defined as $\omega(M) = \sum_{e \in M} \omega(e)$.
See also: Minimum Weight Maximal Matching.

***k*-Nonblocker**
Input: A graph $G = (V, E)$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a non-blocking set $D \subseteq V$ in G of size at least k?

A *non-blocking set* is a vertex subset $D \subseteq V$ such that $N[V \setminus D] = V$, i.e., such that $V \setminus D$ is a dominating set.
Notice that the parametric dual of this problem is k-Dominating Set.
See also: Dominating Set.

***k*-Not-All-Equal Satisfiability**
Input: A set of clauses C using a set of variables X.
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a truth assignment to the variables in X such that at least k clauses in C contain a literal set to true and a literal set to false?

See also: Not-All-Equal Satisfiability.

***k*-Set Splitting**
Input: A collection of sets $\mathcal{S}$ over a universe $\mathcal{U}$.
Parameter: An integer $k \in \mathbb{N}$.
Question: Can $\mathcal{U}$ be partitioned into two colour classes such that at least k sets from $\mathcal{S}$ are split?

A set $S \in \mathcal{S}$ is defined to be *split* if S contains at least one element of each of the two colour classes, i.e., at least one red element and at least one green element.
This problem is also known as k-Maximum Hypergraph 2-Colouring (see Hypergraph 2-Colouring).

***k*-Vertex Cover**
Input: A graph $G = (V, E)$
Parameter: An integer $k \in \mathbb{N}$.
Question: Does there exist a vertex cover $C \subseteq V$ in G of size at most k?

A *vertex cover* is a vertex subset $C \subseteq V$ such that every edge $e \in E$ has an endpoint in C.
Notice that the parametric dual of this problem is k-Independent Set.
See also: Vertex Cover.

Counting Problems

#Dominating Set
Input: A graph $G = (V, E)$.
Question: How many dominating sets $D \subseteq V$ in G exist of minimum size?

A *dominating set* is a vertex subset $D \subseteq V$ such that $N[D] = V$.
See also: Dominating Set.

#Independent Set
Input: A graph $G = (V, E)$.
Question: How many independents set $I \subseteq V$ in G exist of maximum size?

An *independent set* is a vertex subset $I \subseteq V$ such that $\{u, v\} \notin E$ for every two distinct vertices $u, v \in I$.
See also: Independent Set.

#Perfect Matching
Input: A graph $G = (V, E)$.
Question: How many perfect matchings exist in G?

A *perfect matching* in G is a set of edges $M \subseteq E$ such that every vertex $v \in V$ is incident to exactly one edge in M.

#Minimum Weight Dominating Set
Input: A graph $G = (V, E)$ and a weight function $\omega : V \to \mathbb{R}_+$.
Question: How many dominating sets $D \subseteq V$ in G exist of minimum weight?

The weight of a dominating set D (see Dominating Set) is defined as $\omega(D) = \sum_{v \in D} \omega(v)$.

#k-Satisfiability
Input: A set of clauses C with each clause of size at most k using a set of variables X.
Question: How many truth assignments to the variables in X exist that satisfy all clauses in C?

Notice that k is fixed, and that this leads to a different problem for every $k \in \mathbb{N}$.

Bibliography

[1] Faisal N. Abu-Khzam, Amer E. Mouawad, and Mathieu Liedloff. An exact algorithm for connected red-blue dominating set. In T. Calamoneri and J. Díaz, editors, *7th International Conference on Algorithms and Complexity, CIAC 2010*, volume 6078 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2010.

[2] Jochen Alber, Hans L. Bodlaender, Henning Fernau, Ton Kloks, and Rolf Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, 33(4):461–493, 2002.

[3] Jochen Alber and Rolf Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In S. Rajsbaum, editor, *5th Latin American Theoretical Informatics Symposium, LATIN 2002*, volume 2286 of *Lecture Notes in Computer Science*, pages 613–628. Springer, 2002.

[4] Omid Amini, Fedor V. Fomin, and Saket Saurabh. Implicit branching and parameterized partial cover problems. In R. Hariharan, M. Mukund, and V. Vinay, editors, *28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008*, volume 2 of *Leibniz International Proceedings in Informatics*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[5] Omid Amini, Fedor V. Fomin, and Saket Saurabh. Counting subgraphs via homomorphisms. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, editors, *36th International Colloquium on Automata, Languages and Programming (1), ICALP 2009*, volume 5555 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2009.

[6] Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, 2010.

[7] Gunnar Andersson and Lars Engebretsen. Better approximation algorithms for SET SPLITTING and NOT-ALL-EQUAL SAT. *Information Processing Letters*, 65(6):305–311, 1998.

[8] Ola Angelsmark. *Constructing Algorithms for Constraint Satisfaction and Related Problems: Methods and Applications*. PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 2005.

[9] Ola Angelsmark and Johan Thapper. Partitioning based algorithms for some colouring problems. In B. Hnich, M. Carlsson, F. Fages, and F. Rossi, editors, *Joint ERCIM/CoLogNET International Workshop on Constraint Solving and*

Constraint Logic Programming, CSCLP 2005, volume 3978 of *Lecture Notes in Computer Science*, pages 44–58. Springer, 2005.

[10] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, 1987.

[11] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[12] Gábor Bacsó and Zsolt Tuza. Dominating subgraphs of small diameter. *Journal of Combinatorics, Information and System Sciences*, 22(1):51–62, 1997.

[13] Sven Baumer and Rainer Schuler. Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2004.

[14] Eric T. Bax. Inclusion and exclusion algorithm for the Hamiltonian path problem. *Information Processing Letters*, 47(4):203–207, 1993.

[15] Richard Beigel. Finding maximum independent sets in sparse and general graphs. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pages 856–857. Society for Industrial and Applied Mathematics, 1999.

[16] Richard Beigel and David Eppstein. 3-coloring in time $O(1.3446^n)$: A no-MIS algorithm. In *36th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1995*, pages 444–452. IEEE Computer Society, 1995.

[17] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962.

[18] Claude Berge. *The Theory of Graphs and its Applications*. Methuen, 1962.

[19] Marshall W. Bern, Eugene L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8(2):216–235, 1987.

[20] Nadja Betzler, Michael R. Fellows, Christian Komusiewicz, and Rolf Niedermeier. Parameterized algorithms and hardness results for some graph motif problems. In P. Ferragina and G. M. Landau, editors, *19th Annual Symposium on Combinatorial Pattern Matching, CPM 2008*, volume 5029 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2008.

[21] Sergei L. Bezrukov, Robert Elsässer, Burkhard Monien, Robert Preis, and Jean-Pierre Tillich. New spectral lower bounds on the bisection width of graphs. *Theoretical Computer Science*, 320(2-3):155–174, 2004.

[22] Daniel Binkele-Raible, Ljiljana Brankovic, Henning Fernau, Joachim Kneis, Dieter Kratsch, Alexander Langer, Mathieu Liedloff, and Peter Rossmanith. A parameterized route to exact puzzles: Breaking the 2^n-barrier for irredundance. In T. Calamoneri and J. Díaz, editors, *7th International Conference on Algorithms and Complexity, CIAC 2010*, volume 6078 of *Lecture Notes in Computer Science*, pages 311–322. Springer, 2010.

[23] Daniel Binkele-Raible and Henning Fernau. Enumerate & measure: Improving parameter budget management. In V. Raman and S. Saurabh, editors, *5th International Symposium on Parameterized and Exact Computation, IPEC 2010*, volume 6478 of *Lecture Notes in Computer Science*, pages 38–49. Springer, 2010.

[24] Andreas Björklund. *Algorithmic Bounds for Presumably Hard Combinatorial Problems*. PhD thesis, Department of Computer Science, Lund University, Lund, Sweden, 2007.

[25] Andreas Björklund. Determinant sums for undirected Hamiltonicity. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 173–182. IEEE Computer Society, 2010.

[26] Andreas Björklund. Exact covers via determinants. In J.-Y. Marion and T. Schwentick, editors, *27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010*, volume 3 of *Leibniz International Proceedings in Informatics*, pages 95–106. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[27] Andreas Björklund and Thore Husfeldt. Exact algorithms for exact satisfiability and number of perfect matchings. *Algorithmica*, 52(2):226–249, 2008.

[28] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets Möbius: Fast subset convolution. In D. S. Johnson and U. Feige, editors, *39th Annual ACM Symposium on Theory of Computing, STOC 2007*, pages 67–74. ACM Press, 2007.

[29] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. The travelling salesman problem in bounded degree graphs. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *35th International Colloquium on Automata, Languages and Programming (1), ICALP 2008*, volume 5125 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2008.

[30] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Counting paths and packings in halves. In A. Fiat and P. Sanders, editors, *17th Annual European Symposium on Algorithms, ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 578–586. Springer, 2009.

[31] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Covering and packing in linear space. In S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, editors, *37th International Colloquium on Automata, Languages and Programming (1), ICALP 2010*, volume 6198 of *Lecture Notes in Computer Science*, pages 727–737. Springer, 2010.

[32] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Trimmed Moebius inversion and graphs of bounded degree. *Theory of Computing Systems*, 47(3):637–654, 2010.

[33] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, 39(2):546–563, 2009.

[34] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In T. Lepistö and A. Salomaa, editors, *15th International Colloquium on Automata, Languages and Programming, ICALP 1988*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 1988.

[35] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11(4):631–643, 1990.

[36] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–22, 1993.

[37] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[38] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Prívara and P. Ruzicka, editors, *22nd International Symposium on Mathematical Foundations of Computer Science, MFCS 1997*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1997.

[39] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

[40] Hans L. Bodlaender, Erik D. Demaine, Michael R. Fellows, Jiong Guo, Danny Hermelin, Daniel Lokshtanov, Moritz Müller, Venkatesh Raman, Johan M. M. van Rooij, and Frances A. Rosamond. Open problems in parameterized and exact computation - IWPEC 2008. Technical Report UU-CS-2008-017, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2008.

[41] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. In Y. Azar and T. Erlebach, editors, *14th Annual European Symposium on Algorithms, ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 672–683. Springer, 2006.

[42] Hans L. Bodlaender, Fedor V. Fomin, Daniel Lokshtanov, Eelko Penninkx, Saket Saurabh, and Dimitrios M. Thilikos. (Meta) kernelization. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*, pages 629–638. IEEE Computer Society, 2009.

[43] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995.

[44] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[45] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[46] Hans L. Bodlaender and Dieter Kratsch. An exact algorithm for graph coloring with polynomial memory. Technical Report UU-CS-2006-015, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2006.

[47] Hans L. Bodlaender and Rolf H. Möhring. The pathwidth and treewidth of cographs. *SIAM Journal on Discrete Mathematics*, 6(2):181–188, 1993.

[48] Hans L. Bodlaender and Dimitrios M. Thilikos. Constructive linear time algorithms for branchwidth. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *24th International Colloquium on Automata, Languages and Programming, ICALP 1997*, volume 1256 of *Lecture Notes in Computer Science*, pages 627–637. Springer, 1997.

[49] Hans L. Bodlaender and Babette van Antwerpen-de Fluiter. Reduction algorithms for graphs of small treewidth. *Information and Computation*, 167(2):86–119, 2001.

[50] Hans L. Bodlaender, Erik Jan van Leeuwen, Johan M. M. van Rooij, and Martin Vatshelle. Faster algorithms on branch and clique decompositions. In P. Hlinený and A. Kucera, editors, *35th International Symposium on Mathematical Foundations of Computer Science, MFCS 2010*, volume 6281 of *Lecture Notes in Computer Science*, pages 174–185. Springer, 2010.

[51] Hans L. Bodlaender and Johan M. M. van Rooij. Exact algorithms for intervalising colored graphs. In A. Marchetti-Spaccamela and M. Segal, editors, *TAPAS 2011*, volume 6595 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2011.

[52] Vincent Bouchitté, Dieter Kratsch, Haiko Müller, and Ioan Todinca. On treewidth approximations. *Discrete Applied Mathematics*, 136(2-3):183–196, 2004.

[53] Nicolas Bourgeois. *Algorithmes Exponentiels pour la Résolution Exacte et Approchée de Problèmes NP-Complets*. PhD thesis, Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la DEcision, Université Paris Dauphine, Paris, France, 2010.

[54] Nicolas Bourgeois, Federico Della Croce, Bruno Escoffier, and Vangelis Th. Paschos. Exact algorithms for dominating clique problems. In Y. Dong, D.-Z. Du, and O. H. Ibarra, editors, *20th International Symposium on Algorithms and Computation, ISAAC 2009*, volume 5878 of *Lecture Notes in Computer Science*, pages 4–13. Springer, 2009.

[55] Nicolas Bourgeois, Bruno Escoffier, and Vangelis Th. Paschos. An $O^*(1.0977^n)$ exact algorithm for max independent set in sparse graphs. In M. Grohe and R. Niedermeier, editors, *3th International Workshop on Parameterized and Exact Computation, IWPEC 2008*, volume 5018 of *Lecture Notes in Computer Science*, pages 55–65. Springer, 2008.

[56] Nicolas Bourgeois, Bruno Escoffier, and Vangelis Th. Paschos. Fast algorithms for min independent dominating set. In B. Patt-Shamir and T. Ekim, editors, *17th International Colloquium Structural Information and Communication Complexity, SIROCCO 2010*, volume 6058 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2010.

[57] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Fast algorithms for max independent set in graphs of small average degree. Cahier du LAMSADE 277, Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la DEcision, Université Paris Dauphine, Paris, France, 2008.

[58] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. A bottom-up method and fast algorithms for max independent set. In H. Kaplan, editor, *12th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2010*, volume 6139 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 2010.

[59] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Fast algorithms for max independent set. *Algorithmica*, 2010. Accepted for publication, to appear.

[60] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Maximum independent set in graphs of average degree at most three in $O(1.08537^n)$. In J. Kratochvíl, A. Li, J. Fiala, and P. Kolman, editors, *7th Annual Conference on Theory and Applications of Models of Computation, TAMC 2010*, volume 6108 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2010.

[61] Tobias Brüggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1-3):303–313, 2004.

[62] Binh-Minh Bui-Xuan, Jan Arne Telle, and Martin Vatshelle. Boolean-width of graphs. In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2009.

[63] Binh-Minh Bui-Xuan, Jan Arne Telle, and Martin Vatshelle. H-Join decomposable graphs and algorithms with runtime single exponential in rankwidth. *Discrete Applied Mathematics*, 158(7):809–819, 2010.

[64] Jonathan F. Buss and Judy Goldsmith. Nondeterminism within P. *SIAM Journal on Computing*, 22(3):560–572, 1993.

[65] Jesper M. Byskov. Enumerating maximal independent sets with applications to graph colouring. *Operations Research Letters*, 32(6):547–556, 2004.

[66] Jesper M. Byskov. *Exact Algorithms for Graph Colouring and Exact Satisfiability*. PhD thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, 2004.

[67] Jesper M. Byskov, Bolette A. Madsen, and Bjarke Skjernaa. New algorithms for exact satisfiability. *Theoretical Computer Science*, 332(1-3):515–541, 2005.

[68] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2009.

[69] Robert D. Carr, Toshihiro Fujito, Goran Konjevod, and Ojas Parekh. A $2\frac{1}{10}$-approximation algorithm for a generalization of the weighted edge-dominating set problem. *Journal of Combinatorial Optimization*, 5(3):317–326, 2001.

[70] Mathieu Chapelle. Parameterized complexity of generalized domination problems on bounded tree-width graphs. *arXiv.org. The Computing Research Repository*, abs/1004.2642, 2010.

[71] Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. Strong computational lower bounds via parameterized complexity. *Journal of Computer and System Sciences*, 72(8):1346–1367, 2006.

[72] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[73] Jianer Chen, Iyad A. Kanj, and Ge Xia. Labeled search trees and amortized analysis: Improved upper bounds for NP-hard problems. *Algorithmica*, 43(4):245–273, 2005.

[74] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010.

[75] Jianer Chen, Lihua Liu, and Weijia Jia. Improvement on vertex cover for low-degree graphs. *Networks*, 35(4):253–259, 2000.

[76] Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM*, 55(5), 2008.

[77] Jianer Chen and Songjian Lu. Improved parameterized set splitting algorithms: A probabilistic approach. *Algorithmica*, 54(4):472–489, 2009.

[78] Nicos Christofides. An algorithm for the chromatic number of a graph. *The Computer Journal*, 14(1):38–39, 1971.

[79] Alan Cobham. The intrinsic computational difficulty of functions. In *1964 Congress on Logic, Mathematics and the Philosophy of Science*, pages 24–30. North-Holland, 1964.

[80] Stephen A. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing, STOC 1971*, pages 151–158. ACM Press, 1971.

[81] William Cook and Paul D. Seymour. An algorithm for the ring-routing problem. Bellcore Technical Memorandum, Bellcore, 1993.

[82] William Cook and Paul D. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

[83] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[84] Derek G. Corneil, Michel Habib, Jean-Marc Lanlignel, Bruce A. Reed, and Udi Rotics. Polynomial time recognition of clique-width ≤ 3 graphs. In G. H. Gonnet, D. Panario, and A. Viola, editors, *4th Latin American Theoretical Informatics Symposium, LATIN 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 126–134. Springer, 2000.

[85] Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Handle-rewriting hypergraph grammars. *Journal of Computer and System Sciences*, 46(2):218–270, 1993.

[86] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theoretical Computer Science*, 33(2):125–150, 2000.

[87] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000.

[88] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. *arXiv.org. The Computing Research Repository*, abs/1103.0534, 2011.

[89] Marek Cygan and Marcin Pilipczuk. Faster exact bandwidth. In H. Broersma, T. Erlebach, T. Friedetzky, and D. Paulusma, editors, *34th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2008*, volume 5344 of *Lecture Notes in Computer Science*, pages 101–109. Springer, 2008.

[90] Marek Cygan, Marcin Pilipczuk, and Jakub O. Wojtaszczyk. Capacitated domination faster than 2^n. In H. Kaplan, editor, *12th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2010*, volume 6139 of *Lecture Notes in Computer Science*, pages 74–80. Springer, 2010.

[91] Marek Cygan, Marcin Pilipczuk, and Jakub O. Wojtaszczyk. Irredundant set faster than 2^n. In T. Calamoneri and J. Díaz, editors, *7th International Conference on Algorithms and Complexity, CIAC 2010*, volume 6078 of *Lecture Notes in Computer Science*, pages 288–298. Springer, 2010.

[92] Ole-Johan Dahl, Edsger W. Dijkstra, and Charles A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[93] Vilhelm Dahllöf and Peter Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 292–298. Society for Industrial and Applied Mathematics, 2002.

[94] Vilhelm Dahllöf, Peter Jonsson, and Richard Beigel. Algorithms for four variants of the exact satisfiability problem. *Theoretical Computer Science*, 320(2-3):373–394, 2004.

[95] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting satisfying assignments in 2-SAT and 3-SAT. In O. H. Ibarra and L. Zhang, editors, *8th Annual International Conference on Computing and Combinatorics, COCOON 2002*, volume 2387 of *Lecture Notes in Computer Science*, pages 535–543. Springer, 2002.

[96] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1-3):265–291, 2005.

[97] Peter Damaschke. Bounded-degree techniques accelerate some parameterized graph algorithms. In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 98–109. Springer, 2009.

[98] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon M. Kleinberg, Christos H. Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[99] Evgeny Dantsin, Edward A. Hirsch, Sergei Ivanov, and Maxim Vsemirnov. Algorithms for SAT and upper bounds on their complexity. *Electronic Colloquium on Computational Complexity*, 8(12), 2001.

[100] Evgeny Dantsin and Alexander Wolpert. On moderately exponential time for SAT. In O. Strichman and S. Szeider, editors, *13th International Conference on Theory and Applications of Satisfiability Testing, SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 313–325. Springer, 2010.

[101] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[102] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[103] Babette L. E. de Fluiter. *Algorithms for Graphs of Small Treewidth.* PhD thesis, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 1997.

[104] Frank K. H. A. Dehne, Michael R. Fellows, Henning Fernau, Elena Prieto, and Frances A. Rosamond. Nonblocker: Parameterized algorithmics for minimum dominating set. In J. Wiedermann, G. Tel, J. Pokorný, M. Bieliková, and

J. Stuller, editors, *32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2006*, volume 3831 of *Lecture Notes in Computer Science*, pages 237–245. Springer, 2006.

[105] Frank K. H. A. Dehne, Michael R. Fellows, Michael A. Langston, Frances A. Rosamond, and Kim Stevens. An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. *Theory of Computing Systems*, 41(3):479–492, 2007.

[106] Frank K. H. A. Dehne, Michael R. Fellows, and Frances A. Rosamond. An FPT algorithm for set splitting. In H. L. Bodlaender, editor, *29th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2003*, volume 2880 of *Lecture Notes in Computer Science*, pages 180–191. Springer, 2003.

[107] Frank K. H. A. Dehne, Michael R. Fellows, Frances A. Rosamond, and Peter Shaw. Greedy localization, iterative compression, modeled crown reductions: New FPT techniques, an improved algorithm for set splitting, and a novel $2k$ kernelization for vertex cover. In R. G. Downey, M. R. Fellows, and F. K. H. A. Dehne, editors, *1st International Workshop on Parameterized and Exact Computation, IWPEC 2004*, volume 3162 of *Lecture Notes in Computer Science*, pages 271–280. Springer, 2004.

[108] Erik D. Demaine, Fedor V. Fomin, MohammadTaghi Hajiaghayi, and Dimitrios M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and H-minor-free graphs. *Journal of the ACM*, 52(6):866–893, 2005.

[109] Erik D. Demaine and MohammadTaghi Hajiaghayi. The bidimensionality theory and its algorithmic applications. *The Computer Journal*, 51(3):292–302, 2008.

[110] Frederic Dorn. Dynamic programming and fast matrix multiplication. In Y. Azar and T. Erlebach, editors, *14th Annual European Symposium on Algorithms, ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 280–291. Springer, 2006.

[111] Frederic Dorn. *Designing Subexponential Algorithms: Problems, Techniques & Structures*. PhD thesis, Department of Informatics, University of Bergen, Bergen, Norway, 2007.

[112] Frederic Dorn. Dynamic programming and planarity: Improved tree-decomposition based algorithms. *Discrete Applied Mathematics*, 158(7):800–808, 2010.

[113] Frederic Dorn, Fedor V. Fomin, and Dimitrios M. Thilikos. Catalan structures and dynamic programming in H-minor-free graphs. In S.-H. Teng, editor, *19th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008*, pages 631–640. Society for Industrial and Applied Mathematics, 2008.

[114] Frederic Dorn, Eelko Penninkx, Hans L. Bodlaender, and Fedor V. Fomin. Efficient exact algorithms on planar graphs: Exploiting sphere cut decompositions. *Algorithmica*, 58(3):790–810, 2010.

[115] Rodney G. Downey and Michael R. Fellows. Fixed-parameter intractability. In *7th Annual IEEE Structure in Complexity Theory Conference, CoCo 1992*, pages 36–49. IEEE Computer Society, 1992.

[116] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness. *Congressus Numerantium*, 87:161–178, 1992.

[117] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.

[118] Limor Drori and David Peleg. Faster exact solutions for some NP-hard problems. *Theoretical Computer Science*, 287(2):473–499, 2002.

[119] Olivier Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64, 1991.

[120] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:445–467, 1965.

[121] David Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3):275–291, 2000.

[122] David Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2001*, pages 329–337. Society for Industrial and Applied Mathematics, 2001.

[123] David Eppstein. Small maximal independent sets and faster exact graph coloring. *Journal of Graph Algorithms and Applications*, 7(2):131–140, 2003.

[124] David Eppstein. Quasiconvex analysis of multivariate recurrence equations for backtracking algorithms. *ACM Transactions on Algorithms*, 2(4):492–509, 2006.

[125] Paul Erdös. On a combinatorial problem, I. *Nordisk Matematisk Tidskrift*, 11:5–10, 1963.

[126] Paul Erdös. On a combinatorial problem, II. *Acta Mathematica Hungarica*, 15(3):445–447, 1964.

[127] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

[128] Michael R. Fellows, Daniel Lokshtanov, Neeldhara Misra, Matthias Mnich, Frances A. Rosamond, and Saket Saurabh. The complexity ecology of parameters: An illustration using bounded max leaf number. *Theory of Computing Systems*, 45(4):822–848, 2009.

[129] Michael R. Fellows, Daniel Lokshtanov, Neeldhara Misra, Frances A. Rosamond, and Saket Saurabh. Graph layout problems parameterized by vertex cover. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *19th International Symposium on Algorithms and Computation, ISAAC 2008*, volume 5369 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2008.

[130] Michael R. Fellows, Frances A. Rosamond, Udi Rotics, and Stefan Szeider. Clique-width is NP-complete. *SIAM Journal on Discrete Mathematics*, 23(2):909–939, 2009.

[131] Henning Fernau. Edge dominating set: Efficient enumeration-based exact algorithms. In H. L. Bodlaender and M. A. Langston, editors, *2nd International Workshop on Parameterized and Exact Computation, IWPEC 2006*, volume 4169 of *Lecture Notes in Computer Science*, pages 142–153. Springer, 2006.

[132] Henning Fernau. Minimum dominating set of queens: A trivial programming exercise? *Discrete Applied Mathematics*, 158(4):308–318, 2010.

[133] Henning Fernau, Joachim Kneis, Dieter Kratsch, Alexander Langer, Mathieu Liedloff, Daniel Raible, and Peter Rossmanith. An exact algorithm for the maximum leaf spanning tree problem. In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2009.

[134] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006.

[135] Fedor V. Fomin, Serge Gaspers, Dieter Kratsch, Mathieu Liedloff, and Saket Saurabh. Iterative compression and exact algorithms. *Theoretical Computer Science*, 411(7-9):1045–1053, 2010.

[136] Fedor V. Fomin, Serge Gaspers, Artem V. Pyatkin, and Igor Razgon. On the minimum feedback vertex set problem: Exact and enumeration algorithms. *Algorithmica*, 52(2):293–307, 2008.

[137] Fedor V. Fomin, Serge Gaspers, and Saket Saurabh. Improved exact algorithms for counting 3- and 4-colorings. In G. Lin, editor, *13th Annual International Conference on Computing and Combinatorics, COCOON 2007*, volume 4598 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2007.

[138] Fedor V. Fomin, Serge Gaspers, Saket Saurabh, and Alexey A. Stepanov. On two techniques of combining branching and treewidth. *Algorithmica*, 54(2):181–207, 2009.

[139] Fedor V. Fomin, Petr A. Golovach, Jan Kratochvíl, Dieter Kratsch, and Mathieu Liedloff. Sort and search: Exact algorithms for generalized domination. *Information Processing Letters*, 109(14):795–798, 2009.

[140] Fedor V. Fomin, Petr A. Golovach, Jan Kratochvíl, Dieter Kratsch, and Mathieu Liedloff. Branch and recharge: Exact algorithms for generalized domination. *Algorithmica*, 2010. Accepted for publication, to appear.

[141] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Measure and conquer: Domination - a case study. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *32nd International Colloquium on Automata, Languages and Programming, ICALP 2005*, volume 3580 of *Lecture Notes in Computer Science*, pages 191–203. Springer, 2005.

[142] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS*, 87:47–77, 2005.

[143] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Solving connected dominating set faster than 2^n. *Algorithmica*, 52(2):153–166, 2008.

[144] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM*, 56(5), 2009.

[145] Fedor V. Fomin, Fabrizio Grandoni, Artem V. Pyatkin, and Alexey A. Stepanov. Bounding the number of minimal dominating sets: A measure and conquer approach. In X. Deng and D.-Z. Du, editors, *16th International Symposium on Algorithms and Computation, ISAAC 2005*, volume 3827 of *Lecture Notes in Computer Science*, pages 573–582. Springer, 2005.

[146] Fedor V. Fomin, Fabrizio Grandoni, Artem V. Pyatkin, and Alexey A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Transactions on Algorithms*, 5(1), 2008.

[147] Fedor V. Fomin and Kjartan Høie. Pathwidth of cubic graphs and exact algorithms. *Information Processing Letters*, 97(5):191–196, 2006.

[148] Fedor V. Fomin, Kazuo Iwama, Dieter Kratsch, Petteri Kaski, Mikko Koivisto, Lukasz Kowalik, Yoshio Okamoto, Johan M. M. van Rooij, and Ryan Williams. 08431 open problems – moderately exponential time algorithms. In F. V. Fomin, K. Iwama, and D. Kratsch, editors, *Moderately Exponential Time Algorithms*, number 08431 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[149] Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. Springer, 2010.

[150] Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact algorithms for treewidth and minimum fill-in. *SIAM Journal on Computing*, 38(3):1058–1079, 2008.

[151] Fedor V. Fomin, Dieter Kratsch, and Gerhard J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In J. Hromkovic, M. Nagl, and B. Westfechtel, editors, *30th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2004*, volume 3353 of *Lecture Notes in Computer Science*, pages 24–256. Springer, 2004.

[152] Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, and Saket Saurabh. Subexponential algorithms for partial cover problems. In R. Kannan and K. Narayan Kumar, editors, *29th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009*, volume 4 of *Leibniz International Proceedings in Informatics*, pages 193–201. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009.

[153] Fedor V. Fomin and Dimitrios M. Thilikos. A simple and fast approach for solving problems on planar graphs. In V. Diekert and M. Habib, editors, *21st International Symposium on Theoretical Aspects of Computer Science, STACS 2004*, volume 2996 of *Lecture Notes in Computer Science*, pages 56–67. Springer, 2004.

[154] Fedor V. Fomin and Dimitrios M. Thilikos. Dominating sets in planar graphs: Branch-width and exponential speed-up. *SIAM Journal on Computing*, 36(2):281–309, 2006.

[155] Fedor V. Fomin and Yngve Villanger. Treewidth computation and extremal combinatorics. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *35th International Colloquium on Automata, Languages and Programming (1), ICALP 2008*, volume 5125 of *Lecture Notes in Computer Science*, pages 210–221. Springer, 2008.

[156] Fedor V. Fomin and Yngve Villanger. Finding induced subgraphs via minimal triangulations. In J.-Y. Marion and T. Schwentick, editors, *27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010*, volume 3 of *Leibniz International Proceedings in Informatics*, pages 383–394. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[157] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[158] Martin Fürer. A faster algorithm for finding maximum independent sets in sparse graphs. In J. R. Correa, A. Hevia, and M. A. Kiwi, editors, *7th Latin American Symposium on Theoretical Informatics, LATIN 2006*, volume 3887 of *Lecture Notes in Computer Science*, pages 491–501. Springer, 2006.

[159] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

[160] Martin Fürer and Shiva Prasad Kasiviswanathan. Algorithms for counting 2-SAT solutions and colorings with applications. *Electronic Colloquium on Computational Complexity*, 12(33), 2005.

[161] Robert Ganian and Petr Hlinený. On parse trees and Myhill-Nerode-type tools for fandling graphs of bounded rank-width. *Discrete Applied Mathematics*, 158(7):851–867, 2010.

[162] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[163] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[164] Serge Gaspers, Dieter Kratsch, and Mathieu Liedloff. On independent sets and bicliques in graphs. In H. Broersma, T. Erlebach, T. Friedetzky, and D. Paulusma, editors, *34th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2008*, volume 5344 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2008.

[165] Serge Gaspers, Dieter Kratsch, Mathieu Liedloff, and Ioan Todinca. Exponential time algorithms for the minimum dominating set problem on some graph classes. *ACM Transactions on Algorithms*, 6(1), 2009.

[166] Serge Gaspers and Mathieu Liedloff. A branch-and-reduce algorithm for finding a minimum independent dominating set in graphs. In F. V. Fomin, editor, *32nd International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2006*, volume 4271 of *Lecture Notes in Computer Science*, pages 78–89. Springer, 2006.

[167] Serge Gaspers and Gregory B. Sorkin. A universally fastest algorithm for max 2-Sat, max 2-CSP, and everything in between. In C. Mathieu, editor, *20th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009*, pages 606–615. Society for Industrial and Applied Mathematics, 2009.

[168] Fanica Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

[169] Mark K. Goldberg, Thomas H. Spencer, and David A. Berque. A low-exponential algorithm for counting vertex covers. *Graph Theory, Combinatorics, Algorithms, and Applications*, 1:431–444, 1995.

[170] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004.

[171] Fabrizio Grandoni. *Exact Algorithms for Hard Graph Problems*. PhD thesis, Department of Computer Science, Systems and Production, Universitá degli Studi di Roma "Tor Vergata", Rome, Italy, 2004.

[172] Fabrizio Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4(2):209–214, 2006.

[173] Chris Gray, Frank Kammer, Maarten Löffler, and Rodrigo I. Silveira. Removing local extrema from imprecise terrains. *arXiv.org. The Computing Research Repository*, abs/1002.2580, 2010.

[174] Siegmund Günther. Zur mathematischen theorie des schachbretts. *Grunert's Archiv der Mathematik und Physik*, 56(3):281–292, 1874.

[175] Jiong Guo, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences*, 72(8):1386–1396, 2006.

[176] Torben Hagerup. Sorting and searching on the word RAM. In M. Morvan, C. Meinel, and D. Krob, editors, *15th International Symposium on Theoretical Aspects of Computer Science, STACS 1998*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998.

[177] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[178] Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105–142, 1999.

[179] Teresa W. Haynes, Stephen Hedetniemi, and Peter Slater. *Fundamentals of Domination in Graphs*, volume 208 of *Pure and Applied Mathematics*. Marcel Dekker, 1998.

[180] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial & Applied Mathematics*, 10:196–210, 1962.

[181] Timon Hertli, Robin A. Moser, and Dominik Scheder. Improving PPSZ for 3-SAT using critical variables. In T. Schwentick and C. Dürr, editors, *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011*, volume 9 of *Leibniz International Proceedings in Informatics*, pages 237–248. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[182] Illya V. Hicks. Branchwidth heuristics. *Congressus Numerantium*, 159:31–50, 2002.

[183] Illya V. Hicks. Graphs, branchwidth, and tangles! Oh my! *Networks*, 45(2):55–60, 2005.

[184] Illya V. Hicks, Arie M. C. A. Koster, and Elif Kolotoğlu. Branch and tree decomposition techniques for discrete optimization. *Tutorials in Operations Research*, pages 1–19, 2005.

[185] Thomas Hofmeister, Uwe Schöning, Rainer Schuler, and Osamu Watanabe. A probabilistic 3-SAT algorithm further improved. In H. Alt and A. Ferreira, editors, *19th International Symposium on Theoretical Aspects of Computer Science, STACS 2002*, volume 2285 of *Lecture Notes in Computer Science*, pages 192–202. Springer, 2002.

[186] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.

[187] Falk Hüffner, Christian Komusiewicz, Hannes Moser, and Rolf Niedermeier. Fixed-parameter algorithms for cluster vertex deletion. *Theory of Computing Systems*, 47(1):196–217, 2010.

[188] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[189] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[190] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[191] Kazuo Iwama. Worst-case upper bounds for k-SAT. *Bulletin of the EATCS*, 82:61–71, 2004.

[192] Kazuo Iwama, Kazuhisa Seto, Tadashi Takai, and Suguru Tamaki. Improved randomized algorithms for 3-SAT. In O. Cheong, K.-Y. Chwa, and K. Park, editors, *21st International Symposium on Algorithms and Computation (1), ISAAC 2010*, volume 6506 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2010.

[193] Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-SAT. In J. I. Munro, editor, *15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pages 328–329. Society for Industrial and Applied Mathematics, 2004.

[194] François Jaeger, Dirk L. Vertigan, and Dominic J. A. Welsh. On the computational complexity of the Jones and Tutte polynomials. *Mathematical Proceedings of the Cambridge Philosophical Society*, 108(1):35–53, 1990.

[195] Tang Jian. An $O(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Transactions on Computers*, 35(9):847–851, 1986.

[196] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[197] David S. Johnson and Mario Szegedy. What are the least tractable instances of max independent set? In *10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pages 927–928. Society for Industrial and Applied Mathematics, 1999.

[198] Viggo Kann. Maximum bounded 3-dimensional matching is MAX SNP-complete. *Information Processing Letters*, 37(1):27–35, 1991.

[199] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *A Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, 1972.

[200] Richard M. Karp. Dynamic programming meets the principle of inclusion-exclusion. *Operations Research Letters*, 1(2):49–51, 1982.

[201] Subhash Khot and Venkatesh Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theoretical Computer Science*, 289(2):997–1008, 2002.

[202] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[203] Joachim Kneis, Alexander Langer, and Peter Rossmanith. A fine-grained analysis of a simple independent set algorithm. In R. Kannan and K. Narayan Kumar, editors, *29th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009*, volume 4 of *Leibniz International Proceedings in Informatics*, pages 287–298. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009.

[204] Joachim Kneis, Daniel Mölle, Stefan Richter, and Peter Rossmanith. A bound on the pathwidth of sparse graphs with applications to exact algorithms. *SIAM Journal on Discrete Mathematics*, 23(1):407–427, 2009.

[205] Joachim Kneis, Daniel Mölle, and Peter Rossmanith. Partial vs. complete domination: t-Dominating set. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, and F. Plasil, editors, *33rd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2007*, volume 4362 of *Lecture Notes in Computer Science*, pages 367–376. Springer, 2007.

[206] Daniel Kobler and Udi Rotics. Edge dominating set and colorings on graphs with fixed clique-width. *Discrete Applied Mathematics*, 126(2-3):197–221, 2003.

[207] Samuel Kohn, Allan Gottlieb, and Meryle Kohn. A generating function approach to the traveling salesman problem. In *Proceedings of the 1977 Annual Conference of the ACM*, pages 294–300. ACM Press, 1977.

[208] Mikko Koivisto. Partitioning into sets of bounded cardinality. In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2009.

[209] Mikko Koivisto and Pekka Parviainen. A space-time tradeoff for permutation problems. In M. Charikar, editor, *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 484–492. Society for Industrial and Applied Mathematics, 2010.

[210] Arist Kojevnikov and Alexander S. Kulikov. A new approach to proving upper bounds for MAX-2-SAT. In C. Stein, editor, *17th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, pages 11–17. ACM Press, 2006.

[211] Ephraim Korach and Nir Solel. Linear time algorithm for minimum weight Steiner tree in graphs with bounded treewidth. Technical Report 632, Technion, Computer Science Department, Israel Institute of Technology, Haifa, Israel, 1990.

[212] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40(3):170–180, 2002.

[213] Ioannis Koutis and Ryan Williams. Limits and applications of group algebras for parameterized problems. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, editors, *36th International Colloquium on Automata, Languages and Programming (1), ICALP 2009*, volume 5555 of *Lecture Notes in Computer Science*, pages 653–664. Springer, 2009.

[214] Dieter Kratsch and Mathieu Liedloff. An exact algorithm for the minimum dominating clique problem. *Theoretical Computer Science*, 385(1-3):226–240, 2007.

[215] Alexander S. Kulikov. An upper bound $O(2^{0.16254n})$ for exact 3-satisfiability: A simpler proof. *Zapiski nauchnyh seminarov POMI*, 293:118–128, 2002. English translation: *Journal of Mathematical Sciences*, 126(3):1995-1999, 2005.

[216] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[217] Oliver Kullmann. Fundamentals of branching heuristics. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 205–244. IOS Press, 2009.

[218] Oliver Kullmann and Horst Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Technical report, Fachbereich Mathematik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1997.

[219] Konstantin Kutzkov and Dominik Scheder. Using CSP to improve deterministic 3-SAT. *arXiv.org. The Computing Research Repository*, abs/1007.1166, 2010.

[220] Eugene L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5(3):66–67, 1976.

[221] Eugene L. Lawler, Jan Karel Lenstra, and Alexander H. G. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[222] Leonid A. Levin. Universal search problems. *Problemy Peredaci Informacii*, 9(3):265–266, 1973.

[223] Mathieu Liedloff. *Algorithmes Exacts et Exponentiels pour les Problèmes NP-Difficiles: Domination, Variantes et Généralisation.* PhD thesis, Laboratoire d'Informatique Théorique et Appliquée, Université Paul Verlaine, Metz, France, 2007.

[224] Mathieu Liedloff. Finding a dominating set on bipartite graphs. *Information Processing Letters*, 107(5):154–157, 2008.

[225] Mathieu Liedloff, Ioan Todinca, and Yngve Villanger. Solving capacitated dominating set by using covering by subsets and maximum matching. In D. M. Thilikos, editor, *36th International Workshop on Graph Theoretic Concepts in Computer Science, WG 2010*, volume 6410 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2010.

[226] Richard J. Lipton. Fast exponential algorithms. Weblog: Gödel's Lost Letter and P=NP, February 13, 2009. http://rjlipton.wordpress.com/2009/02/13/polynomial-vs-exponential-time.

[227] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In D. Randall, editor, *22st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 777–789. Society for Industrial and Applied Mathematics, 2011.

[228] Daniel Lokshtanov and Jesper Nederlof. Saving space by algebraization. In L. J. Schulman, editor, *42nd Annual ACM Symposium on Theory of Computing, STOC 2010*, pages 321–330. ACM Press, 2010.

[229] Daniel Lokshtanov and Saket Saurabh. Even faster algorithm for set splitting! In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 288–299. Springer, 2009.

[230] Daniel Lokshtanov and Christian Sloper. Fixed parameter set splitting, linear kernel and improved running time. In H. Broersma, M. Johnson, and S. Szeider, editors, *1st Algorithms and Complexity in Durham Workshop, ACiD 2005*, volume 4 of *Texts in Algorithmics*, pages 105–113. King's College, London, 2005.

[231] L. Lovász. Coverings and colorings of hypergraphs. *Congressus Numerantium*, 8:3–12, 1973.

[232] Bolette A. Madsen. An algorithm for exact satisfiability analysed with the number of clauses as parameter. *Information Processing Letters*, 97(1):28–30, 2006.

[233] Meena Mahajan and Venkatesh Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999.

[234] William McCuaig and Bruce Shepherd. Domination in graphs with minimum degree two. *Journal of Graph Theory*, 13:749–762, 1989.

[235] Renée J. Miller and David E. Muller. A problem of maximum consistent subsets. IBM Research Report RC-240, J. T. Watson Research Center, Yorktown Heights, New York, USA, 1960.

[236] Matthias Mnich. *Algorithms in Moderately Exponential Time*. PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 2010.

[237] Daniel Mölle, Stefan Richter, and Peter Rossmanith. Enumerate and expand: Improved algorithms for connected vertex cover and tree cover. *Theory of Computing Systems*, 43(2):234–253, 2008.

[238] Burkhard Monien and Robert Preis. Upper bounds on the bisection width of 3- and 4-regular graphs. *Journal of Discrete Algorithms*, 4(3):475–498, 2006.

[239] Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.

[240] Burkhard Monien, Ewald Speckenmeyer, and Oliver Vornberger. Upper bounds for covering problems. *Methods of Operations Research*, 43:419–431, 1981.

[241] John W. Moon and Leo Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3:23–28, 1965.

[242] Robin A. Moser and Dominik Scheder. A full derandomization of Schoening's k-SAT algorithm. *arXiv.org. The Computing Research Repository*, abs/1008.4067, 2010.

[243] Jesper Nederlof. Inclusion exclusion for hard problems. Master's thesis, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2007.

[244] Jesper Nederlof. Fast polynomial-space algorithms using Möbius inversion: Improving on Steiner tree and related problems. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, editors, *36th International Colloquium on Automata, Languages and Programming (1), ICALP 2009*, volume 5555 of *Lecture Notes in Computer Science*, pages 713–725. Springer, 2009.

[245] Jesper Nederlof and Johan M. M. van Rooij. Inclusion/exclusion branching for partial dominating set and set splitting. In V. Raman and S. Saurabh, editors, *5th International Symposium on Parameterized and Exact Computation, IPEC 2010*, volume 6478 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2010.

[246] Michael Nett. Generating all relevant cases for the maximum independent set problem. Aachener Informatik Berichte AIB-2009-09, Department of Computer Science, RWTH Aachen University, Aachen, Germany, 2009.

[247] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, 2006.

[248] Sang-il Oum and Paul D. Seymour. Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, 96(4):514–528, 2006.

[249] Arnold Overwijk, Eelko Penninkx, and Hans L. Bodlaender. A local search algorithm for branchwidth. In I. Cerná, T. Gyimóthy, J. Hromkovic, K. G. Jeffery, R. Královic, M. Vukolic, and S. Wolf, editors, *37th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2011*, volume 6543 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2011.

[250] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.

[251] Daniël Paulusma and Johan M. M. van Rooij. On partitioning a graph into two connected subgraphs. In Y. Dong, D.-Z. Du, and O. H. Ibarra, editors, *20th International Symposium on Algorithms and Computation, ISAAC 2009*, volume 5878 of *Lecture Notes in Computer Science*, pages 1215–1224. Springer, 2009.

[252] Ján Plesník. Equivalence between the minimum covering problem and the maximum matching problem. *Discrete Mathematics*, 49(3):315–317, 1984.

[253] Ján Plesník. Constrained weighted matchings and edge coverings in graphs. *Discrete Applied Mathematics*, 92(2-3):229–241, 1999.

[254] Oriana Ponta, Falk Hüffner, and Rolf Niedermeier. Speeding up dynamic programming for some NP-hard graph recoloring problems. In M. Agrawal, D.-Z. Du, Z. Duan, and A. Li, editors, *5th Annual Conference on Theory and Applications of Models of Computation, TAMC 2008*, volume 4978 of *Lecture Notes in Computer Science*, pages 490–501. Springer, 2008.

[255] Stefan Porschen, Bert Randerath, and Ewald Speckenmeyer. Exact 3-satisfiability is decidable in time $O(2^{0.16254n})$. *Annals of Mathematics and Artificial Intelligence*, 43(1):173–193, 2005.

[256] Elena Prieto. *Systematic Kernelization in FPT Algorithm Design*. PhD thesis, School of Electrical Engineering and Computer Science, The University of Newcastle, Newcastle, Australia, 2005.

[257] Mihai Pătraşcu and Ryan Williams. On the possibility of faster SAT algorithms. In M. Charikar, editor, *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 1065–1075. Society for Industrial and Applied Mathematics, 2010.

[258] Jaikumar Radhakrishnan and Aravind Srinivasan. Improved bounds and algorithms for hypergraph 2-coloring. *Random Structures and Algorithms*, 16(1):4–32, 2000.

[259] Venkatesh Raman, Saket Saurabh, and Somnath Sikdar. Efficient exact algorithms through enumerating maximal independent sets and other techniques. *Theory of Computing Systems*, 41(3):563–587, 2007.

[260] Igor Razgon. Exact computation of maximum induced forest. In L. Arge and R. Freivalds, editors, *10th Scandinavian Workshop on Algorithm Theory, SWAT 2006*, volume 4059 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2006.

[261] Igor Razgon. Faster computation of maximum independent set and parameterized vertex cover for graphs with maximum degree 3. *Journal of Discrete Algorithms*, 7(2):191–212, 2009.

[262] Bruce A. Reed, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299–301, 2004.

[263] Felix Reidl and Fernando Sánchez Villaamil. Automatic verification of the correctness of the upper bound of a maximum independent set algorithm. Aachener Informatik Berichte AIB-2009-10, Department of Computer Science, RWTH Aachen University, Aachen, Germany, 2009.

[264] Tobias Riege. *The Domatic Number Problem: Boolean Hierarchy Completeness and Exact Exponential-Time Algorithms*. PhD thesis, Institut für Informatik, Heinrich-Heine University Düsseldorf, Düsseldorf, Germany, 2006.

[265] Tobias Riege and Jörg Rothe. An exact 2.9416^n algorithm for the three domatic number problem. In J. Jedrzejowicz and A. Szepietowski, editors, *30th International Symposium on Mathematical Foundations of Computer Science, MFCS 2005*, volume 3618 of *Lecture Notes in Computer Science*, pages 733–744. Springer, 2005.

[266] Tobias Riege, Jörg Rothe, Holger Spakowski, and Masaki Yamamoto. An improved exact algorithm for the domatic number problem. *Information Processing Letters*, 101(3):101–106, 2007.

[267] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[268] Neil Robertson and Paul D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.

[269] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[270] John M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.

[271] John M. Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical Report 1251-01, Laboratoire Bordelais de Recherche en Informatique, Université Bordeaux I, Bordeaux, France, 2001.

[272] Robert Rodosek. A new approach on solving 3-satisfiability. In J. Calmet, J. A. Campbell, and J. Pfalzgraf, editors, *3rd International Conference on Artificial Intelligence and Symbolic Mathematical Computation, AISMC-3*, volume 1138 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1996.

[273] Daniel Rolf. Improved bound for the PPSZ/Schöning-algorithm for 3-SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(2):111–122, 2006.

[274] Herbert J. Ryser. *Combinatorial Mathematics*. Number 14 in Carus Mathematical Monographs. Mathematical Association of America, 1963.

[275] Thomas J. Schaefer. The complexity of satisfiability problems. In *10th Annual ACM Symposium on Theory of Computing, STOC 1978*, pages 216–226. ACM Press, 1978.

[276] Dominik Scheder. Guided search and a faster deterministic algorithm for 3-SAT. In E. S. Laber, C. F. Bornstein, L. T. Nogueira, and L. Faria, editors, *8th Latin American Symposium on Theoretical Informatics, LATIN 2008*, volume 4957 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008.

[277] Ingo Schiermeyer. Solving 3-satisfiability in less than $O(1.579^n)$ steps. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M. M. Richter, editors, *6th Workshop on Computer Science Logic, CSL 1992*, volume 702 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 1993.

[278] Ingo Schiermeyer. Deciding 3-colourability in less than $O(1.415^n)$ steps. In J. van Leeuwen, editor, *19th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 1993*, volume 790 of *Lecture Notes in Computer Science*, pages 177–188. Springer, 1994.

[279] Ingo Schiermeyer. Efficiency in exponential time for domination-type problems. *Discrete Applied Mathematics*, 156(17):3291–3297, 2008.

[280] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1999*, pages 410–414. IEEE Computer Society, 1999.

[281] Uwe Schöning. New algorithms for k-SAT based on the local search principle. In J. Sgall, A. Pultr, and P. Kolman, editors, *26th International Symposium on Mathematical Foundations of Computer Science, MFCS 2001*, volume 2136 of *Lecture Notes in Computer Science*, pages 87–95. Springer, 2001.

[282] Uwe Schöning. Algorithmics in exponential time. In V. Diekert and B. Durand, editors, *22nd International Symposium on Theoretical Aspects of Computer Science, STACS 2005*, volume 3404 of *Lecture Notes in Computer Science*, pages 36–43. Springer, 2005.

[283] Richard Schroeppel and Adi Shamir. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.

[284] Alexander D. Scott and Gregory B. Sorkin. Linear-programming design and analysis of fast algorithms for max 2-CSP. *Discrete Optimization*, 4(3-4):260–287, 2007.

[285] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[286] Paul D. Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[287] Miklo Shindo and Etsuji Tomita. A simple algorithm for finding a maximum clique and its worst-case time complexity. *Systems and Computers in Japan*, 21(3):1–13, 1990.

[288] Alexey A. Stepanov. *Exact Algorithms for Hard Counting, Listing and Decision Problems*. PhD thesis, Department of Informatics, University of Bergen, Bergen, Norway, 2008.

[289] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.

[290] Karol Suchan and Yngve Villanger. Computing pathwidth faster than 2^n. In J. Chen and F. V. Fomin, editors, *4th International Workshop on Parameterized and Exact Computation, IWPEC 2009*, volume 5917 of *Lecture Notes in Computer Science*, pages 324–335. Springer, 2009.

[291] Robert E. Tarjan. Finding a maximum clique. Technical report, Department of Computer Science, Cornell University, Ithaca, New York, USA, 1972.

[292] Robert E. Tarjan and Anthony E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[293] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[294] Jan Arne Telle. Complexity of domination-type problems in graphs. *Nordic Journal of Computing*, 1(1):157–171, 1994.

[295] Jan Arne Telle. *Vertex Partitioning Problems: Characterization, Complexity and Algorithms on Partial k-Trees*. PhD thesis, Department of Computer and Information Science, University of Oregon, Eugene, Oregon, USA, 1994.

[296] Jan Arne Telle and Andrzej Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997.

[297] Dimitrios M. Thilikos, Maria J. Serna, and Hans L. Bodlaender. Cutwidth I: A linear time fixed parameter algorithm. *Journal of Algorithms*, 56(1):1–24, 2005.

[298] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[299] Johan M. M. van Rooij. Design by measure and conquer: An $O(1.5086^n)$ algorithm for dominating set and similar problems. Master's thesis, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2006. INF/SCR.06.05.

[300] Johan M. M. van Rooij. Polynomial space algorithms for counting dominating sets and the domatic number. In T. Calamoneri and J. Díaz, editors, *7th International Conference on Algorithms and Complexity, CIAC 2010*, volume 6078 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2010.

[301] Johan M. M. van Rooij and Hans L. Bodlaender. Exact algorithms for edge domination. Technical Report UU-CS-2007-051, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2007.

[302] Johan M. M. van Rooij and Hans L. Bodlaender. Design by measure and conquer, a faster exact algorithm for dominating set. In S. Albers and P. Weil, editors, *25th International Symposium on Theoretical Aspects of Computer Science, STACS 2008*, volume 1 of *Leibniz International Proceedings in Informatics*, pages 657–668. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[303] Johan M. M. van Rooij and Hans L. Bodlaender. Exact algorithms for edge domination. In M. Grohe and R. Niedermeier, editors, *3th International Workshop on Parameterized and Exact Computation, IWPEC 2008*, volume 5018 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2008.

[304] Johan M. M. van Rooij and Hans L. Bodlaender. Design by measure and conquer: Exact algorithms for dominating set. Technical Report UU-CS-2009-025, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2009.

[305] Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In A. Fiat and P. Sanders, editors, *17th Annual European Symposium on Algorithms, ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2009.

[306] Johan M. M. van Rooij, Jesper Nederlof, and Thomas C. van Dijk. Inclusion/exclusion meets measure and conquer: Exact algorithms for counting dominating sets. Technical Report UU-CS-2008-043, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2008.

[307] Johan M. M. van Rooij, Jesper Nederlof, and Thomas C. van Dijk. Inclusion/exclusion meets measure and conquer. In A. Fiat and P. Sanders, editors, *17th Annual European Symposium on Algorithms, ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 554–565. Springer, 2009.

[308] Johan M. M. van Rooij, Marcel E. van Kooten Niekerk, and Hans L. Bodlaender. Partitioning sparse graphs into triangles: Relations to exact satisfiability and very fast exponential time algorithms. Technical Report UU-CS-2010-005, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2010.

[309] Johan M. M. van Rooij, Marcel E. van Kooten Niekerk, and Hans L. Bodlaender. Partition into triangles on bounded degree graphs. In I. Cerná, T. Gyimóthy, J. Hromkovic, K. G. Jeffery, R. Královic, M. Vukolic, and S. Wolf, editors, *37th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2011*, volume 6543 of *Lecture Notes in Computer Science*, pages 558–569. Springer, 2011.

[310] Pim van 't Hof and Daniël Paulusma. A new characterization of P_6-free graphs. *Discrete Applied Mathematics*, 158(7):731–740, 2010.

[311] Pim van 't Hof, Daniël Paulusma, and Johan M. M. van Rooij. Computing role assignments of chordal graphs. In M. Kutylowski, W. Charatonik, and M. Gebala, editors, *17th International Symposium on Fundamentals of Computation Theory, FCT 2009*, volume 5699 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2009.

[312] Pim van 't Hof, Daniël Paulusma, and Johan M. M. van Rooij. Computing role assignments of chordal graphs. *Theoretical Computer Science*, 411(40-42):3601–3613, 2010.

[313] Pim van 't Hof, Daniël Paulusma, and Gerhard J. Woeginger. Partitioning graphs into connected parts. *Theoretical Computer Science*, 410(47-49):4834–4843, 2009.

[314] Magnus Wahlström. Exact algorithms for finding minimum transversals in rank-3 hypergraphs. *Journal of Algorithms*, 51(2):107–121, 2004.

[315] Magnus Wahlström. *Algorithms, measures, and upper bounds for satisfiability and related problems*. PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 2007.

[316] Magnus Wahlström. A tighter bound for counting max-weight solutions to 2SAT instances. In M. Grohe and R. Niedermeier, editors, *3th International Workshop on Parameterized and Exact Computation, IWPEC 2008*, volume 5018 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2008.

[317] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005.

[318] Ryan Williams. *Algorithms and Resource Requirements for Fundamental Problems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2007.

[319] Gerhard J. Woeginger. Exact algorithms for NP-hard problems: A survey. In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *5th International Workshop on Combinatorial Optimization - Eureka, You Shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–208. Springer, 2003.

[320] Gerhard J. Woeginger. Space and time complexity of exact algorithms: Some open problems. In R. G. Downey, M. R. Fellows, and F. K. H. A. Dehne, editors, *1st International Workshop on Parameterized and Exact Computation, IWPEC 2004*, volume 3162 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 2004.

[321] Gerhard J. Woeginger. Open problems around exact algorithms. *Discrete Applied Mathematics*, 156(3):397–405, 2008.

[322] Mingyu Xiao. New branching rules: Improvements on independent set and vertex cover in sparse graphs. *arXiv.org. The Computing Research Repository*, abs/0904.2712, 2009.

[323] Mingyu Xiao. Exact and parameterized algorithms for edge dominating set in 3-degree graphs. In W. Wu and O. Daescu, editors, *4th International Conference on Combinatorial Optimization and Applications (2), COCOA 2010*, volume 6509 of *Lecture Notes in Computer Science*, pages 387–400. Springer, 2010.

[324] Mingyu Xiao. A note on vertex cover in graphs with maximum degree 3. In M. T. Thai and S. Sahni, editors, *16th Annual International Conference on Computing and Combinatorics, COCOON 2010*, volume 6196 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2010.

[325] Mingyu Xiao. A simple and fast algorithm for maximum independent set in 3-degree graphs. In Md. S. Rahman and S. Fujita, editors, *4th International Workshop on Algorithms and Computation, WALCOM 2010*, volume 5942 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2010.

[326] Mihalis Yannakakis and Fanica Gavril. Edge dominating sets in graphs. *SIAM Journal on Applied Mathematics*, 38(3):364–372, 1980.

[327] Jiawei Zhang, Yinyu Ye, and Qiaoming Han. Improved approximations for max set splitting and max NAE SAT. *Discrete Applied Mathematics*, 142(1-3):133–149, 2004.

[328] Wenhui Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155(1):277–288, 1996.

[329] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In J. M. Kleinberg, editor, *38th Annual ACM Symposium on Theory of Computing, STOC 2006*, pages 681–690. ACM Press, 2006.

[330] Uri Zwick. Outward rotations: A tool for rounding solutions of semidefinite programming relaxations, with applications to MAX CUT and other problems. In *31th Annual ACM Symposium on Theory of Computing, STOC 1999*, pages 679–687. ACM Press, 1999.

Author Index

This index list all authors associated with a paper referenced in the bibliography as well as other researchers mentioned in the text. The first numbers list the pages where the author is cited, and the second numbers (between brackets) list the referenced papers in which this author was involved.

Subject Index

This index lists subjects used in the text. The numbers list the pages where the corresponding subject is used. Numbers printed in *italics* refer to pages where the subject is defined in a definition environment. Numbers printed in **boldface** are given for computational problems; these numbers refer to pages where the main theorems about these problems are stated.

Summary

In this PhD thesis, we study algorithms for a class of subset problems in graphs called domination problems. Subset problems in graphs are problems where one is given a graph $G = (V, E)$, and one is asked whether there exists some subset S of a set of items U in the graph (mostly U is either the set of vertices V or the set of edges E) that satisfies certain properties. Domination problems in graphs are subset problems in which one of these properties requires a solution subset S to dominate its complement $U \setminus S$; the domination criterion is based on a neighbourhood relation in the graph that decides which elements of U are dominated by a given subset S. The most well-known domination problem in graphs is the Dominating Set problem. In this problem, the set U is the set of vertices V of G, a vertex subset S dominates all vertices in V that have a neighbour in S, and one is asked to compute a smallest vertex subset $S \subseteq V$ that dominates all vertices in $V \setminus S$. Other examples of domination problems in graphs are Independent Set, Edge Dominating Set, Total Dominating Set, Red-Blue Dominating Set, Partial Dominating Set, and #Dominating Set.

We study exact exponential-time algorithms for domination problems in graphs. These algorithms use, in the worst case, a number of computation steps that is exponential in a complexity parameter of the input. In other words, these algorithms use exponential time. Exact exponential-time algorithms are algorithms of this type that are guaranteed to return optimal solutions to their corresponding problems.

Our study led to faster exact exponential-time algorithms for many well-known graph domination problems including the ones stated above. We also obtained a number faster algorithms for related problems. This includes a number of results in parameterised complexity. Prominent among these are faster algorithms operating on graph decompositions for many graph domination problems.

This thesis begins with an introduction to the field of exact exponential-time algorithms. We give an overview of known techniques and relevant concepts from complexity theory and from areas of algorithmic research that are closely related. In the last part of the introduction, we give an example of a 'very fast' exponential-time algorithm. That is, we give an $\mathcal{O}(1.02220^n)$-time and polynomial-space algorithm for the Partition Into Triangles problem restricted to graphs of maximum degree four; a problem for which we can show that no subexponential-time algorithms exist under reasonable complexity-theoretic assumptions.

The first graph domination problem that we study is Dominating Set. For this problem, we obtain an $\mathcal{O}(1.4969^n)$-time and polynomial-space algorithm. This result is based on a stepwise algorithm-design process that combines a careful analysis of the combinatorial structure of the problem with a running-time analyses using the measure-and-conquer technique. We use parts of the same algorithm to also obtain an $\mathcal{O}(1.4969^n)$-time and polynomial-space algorithm for Total Dominating Set, an

$\mathcal{O}(1.2279^n)$-time and polynomial-space algorithm for Red-Blue Dominating Set, and an $\mathcal{O}^*(1.9588^k)$-time and polynomial-space algorithm for k-Nonblocker.

The same approach is used in a different setting to obtain $\mathcal{O}(1.3226^n)$-time and polynomial-space algorithms for Edge Dominating Set and a number of related edge-domination problems. These results are, similar to previous results on edge-domination problems, based on enumerating certain minimal vertex covers in a graph. To obtain our results, we introduce a new reduction rule that allows us to enumerate fewer vertex covers. The approach also leads to $\mathcal{O}^*(2.4006^k)$-time algorithms for k-Edge Dominating Set and some related problems.

For Independent Set, we obtain faster algorithms on graphs with a small average degree than previously known. We give an $\mathcal{O}(1.08537^n)$-time and polynomial-space algorithm for Independent Set on graphs of average degree at most three using extensive case analyses and a running-time analysis based on average degrees. This result is then used to give faster algorithms on graphs of larger average degree. This leads to an $\mathcal{O}(1.2114^n)$-time and polynomial-space algorithm for Independent Set.

Next, we introduce a new technique that we call *inclusion/exclusion-based branching*. This technique uses the principle of inclusion/exclusion as a branching rule, which can be used in a standard branch-and-reduce algorithm. In this way, inclusion/exclusion can easily be combined with other approaches such as different branching rules, reduction rules, or treewidth based approaches on sparse graphs. Using our new technique, we give both an $\mathcal{O}(1.5673^n)$-time and polynomial-space algorithm and an $\mathcal{O}(1.5002^n)$-time-and-space algorithm for #Dominating Set. We also use this approach to obtain results on Domatic Number, Red-Blue Dominating Set, Dominating Set restricted to some graph classes, and, in a completely different setting, a result on Disjoint Connected Subgraphs.

We extend the technique to problems that deal with partial requirements; the corresponding domination problems are partial domination problems. We call the extension of our technique *extended inclusion/exclusion-based branching*. We use it to obtain an $\mathcal{O}(1.5673^n)$-time and polynomial-space algorithm for Partial Dominating Set, an $\mathcal{O}(1.5014^n)$-time-and-space algorithm for Partial Dominating Set, and an $\mathcal{O}^*(1.8213^k)$-time and polynomial-space algorithm for k-Set Splitting.

Next, we consider algorithms on graph decompositions. We combine standard approaches on tree decompositions, branch decompositions, and clique decompositions (also called k-expressions) with variants of the fast subset convolution algorithm. On tree decompositions of width k, we obtain algorithms for many graph domination problems running in $\mathcal{O}^*(s^k)$ time and space by using a technique that we call *generalised fast subset convolution*. Here, s is number of states required to represent partial solutions of a problem. In this way, we obtain faster algorithms on tree decompositions for the $[\rho,\sigma]$-domination problems and a class of clique covering, packing, and partitioning problems. This includes an $\mathcal{O}^*(3^k)$-time-and-space algorithm for Dominating Set and an $\mathcal{O}^*(2^k)$-time-and-space algorithm for #Perfect Matching.

The same approach leads to $\mathcal{O}^*(4^k)$-time-and-space algorithms for Dominating Set, Total Dominating Set, and Independent Dominating Set on clique decompositions. On branch decompositions, we obtain $\mathcal{O}^*(s^{\frac{\omega}{2}k})$-time-and-space algorithms for many graph domination problems by further extending the approach using asymmetric vertex states combined with fast matrix multiplication.

We conclude this thesis with a number of interesting open problems.

Nederlandse Samenvatting

Dit proefschrift gaat over algoritmen voor domineringsproblemen in grafen. Een graaf is een combinatorische structuur die bestaat uit een verzameling knopen en een verzameling lijnen die ieder twee verschillende knopen kunnen verbinden; deze lijnen noemen we kanten. Grafen worden gebruikt om allerlei zaken wiskundig te modelleren. Zo kan bijvoorbeeld een wegennetwerk gemodelleerd worden door voor ieder dorp of stad een knoop te introduceren, en elk paar van deze knopen waarbij er een weg ligt tussen de betreffende dorpen of steden te verbinden met een kant. Een ander voorbeeld van een structuur die vaak gemodelleerd wordt door middel van een graaf is een sociaal netwerk: nu stellen de knopen mensen voor en geven kanten aan dat de verbonden mensen elkaar kennen binnen dit netwerk.

Een graafprobleem is een combinatorisch probleem waarbij iets uitgerekend moet worden over grafen. Dit proefschrift gaat over een bepaald type graafproblemen, namelijk graafdomineringsproblemen. Het meest bekende graafdomineringsprobleem is het probleem met de naam Dominating Set. In dit probleem krijgt men een graaf gegeven en dient in deze graaf de kleinste verzameling knopen gevonden te worden met de volgende eigenschap: iedere knoop in de graaf moet verbonden zijn via een kant met een knoop in de gezochte verzameling of zelf in de gezochte verzameling zitten.

In het algemeen zijn graafdomineringsproblemen problemen waarin bij een gegeven graaf een bepaalde verzameling van objecten in de graaf (meestal een verzameling knopen of een verzameling kanten) gezocht dient te worden met bepaalde eigenschappen. Één van de eigenschappen die de gezochte verzameling objecten moet hebben is dat deze verzameling alle overgebleven objecten domineert. Wanneer een verzameling objecten een ander object domineert hangt af van het specifieke probleem, maar deze definitie hangt in alle gevallen samen met lokale eigenschappen van de graaf. Bij het Dominating Set probleem domineert een verzameling knopen een andere knoop als er een kant tussen deze knoop en een knoop uit de verzameling bestaat. In een ander graafdomineringsprobleem zou bijvoorbeeld een verzameling knopen een andere knoop kunnen domineren als de afstand via de kanten tussen de knoop en een willekeurige knoop uit de verzameling ten hoogste twee is. In weer een ander probleem zou een kant een andere kant kunnen domineren als beide kanten een eindpunt in dezelfde knoop hebben. Enkele andere voorbeelden van domineringsproblemen in grafen zijn Independent Set, Edge Dominating Set, Total Dominating Set, Red-Blue Dominating Set, Partial Dominating Set en #Dominating Set.

Een algoritme is een stapsgewijze beschrijving van hoe je iets uit kunt rekenen. De bekendste algoritmen zijn waarschijnlijk de rekenmethoden die we op de basisschool geleerd hebben: de manieren waarop we grote getallen optellen, aftrekken, vermenigvuldigen of delen. Zo hebben we geleerd het staartdelingalgoritme te gebruiken als rekenmethode om grote delingen uit te voeren. Algoritmen zijn essentieel in com-

puters om dingen uit te rekenen. Zo gebruikt een routeplanner een algoritme om de kortste route van A naar B te berekenen. In dit proefschrift bestuderen we algoritmen voor het oplossen van domineringsproblemen in grafen.

Een belangrijke eigenschap van een algoritme is hoeveel rekenstappen er in het slechtste geval nodig zijn om het algoritme uit te voeren: dit noemen we de looptijd van het algoritme. De looptijd van een algoritme voor een graafprobleem wordt vaak gegeven als een formule uitgedrukt in het aantal knopen of het aantal kanten in de gegeven graaf. In dit proefschrift beschouwen we exponentiële-tijd algoritmen: algoritmen waarvan de looptijd een exponentiële functie is. Dit wil zeggen, algoritmen waarvan de looptijd zich zo gedraagt dat als het algoritme uitgevoerd wordt op een grotere graaf de looptijd dan vermenigvuldigd wordt met een vast getal voor iedere knoop (of kant) dat de graaf groter is. Een exact exponentiële-tijd algoritme is een exponentiële-tijd algoritme dat na het uitvoeren gegarandeerd een optimale oplossing van het gegeven probleem oplevert.

Mijn studie naar exacte exponentiële-tijd algoritmen voor domineringsproblemen in grafen heeft geleid tot snellere algoritmen voor veel bekende graafdomineringsproblemen inclusief de voorbeelden die hierboven gegeven zijn. Deze studie resulteerde ook in een aantal snellere algoritmen voor problemen die gerelateerd zijn aan het onderwerp van het proefschrift, waaronder een aantal resultaten in de geparametriseerde complexiteit. De meest aansprekende hiervan zijn waarschijnlijk de snellere algoritmen voor veel graafdomineringsproblemen op graafdecomposities.

Dit proefschrift begint met een introductie tot het onderzoeksgebied van de exacte exponentiële-tijd algoritmen. Hier geven we een overzicht van bestaande technieken en behandelen we gerelateerde begrippen uit de complexiteitstheorie en uit algoritmische onderzoeksgebieden die verwant zijn aan de exponentiële-tijd algoritmen. In het laatste deel van de introductie geven we een voorbeeld van een 'zeer snel' exponentiële-tijd algoritme. Hier geven we namelijk een algoritme voor het Partition Into Triangles probleem op grafen met maximale graad vier dat $\mathcal{O}(1.02220^n)$ tijd en polynomiaal veel ruimte gebruikt. Voor dit probleem kunnen we laten zien dat er geen subexponentiële-tijd algoritmen voor bestaan tenzij geaccepteerde complexiteitstheoretische aannamen niet waar blijken te zijn.

Het eerste graafdomineringsprobleem dat we bestuderen is Dominating Set. Voor dit probleem geven we een algoritme dat $\mathcal{O}(1.4969^n)$ tijd en polynomiaal veel ruimte gebruikt. Dit resultaat is gebaseerd op een stapsgewijs ontwerpproces voor algoritmen waarbij een grondige analyse van de combinatorische structuur van het probleem gepaard gaat met een looptijdanalyse gebaseerd op de 'measure and conquer' techniek. We gebruiken delen van hetzelfde algoritme om ook een algoritme voor Total Dominating Set te construeren dat $\mathcal{O}(1.4969^n)$ tijd en polynomiaal veel ruimte gebruikt, een algoritme voor Red-Blue Dominating Set te construeren dat $\mathcal{O}(1.2279^n)$ tijd en polynomiaal veel ruimte gebruikt, en een algoritme voor k-Nonblocker te construeren dat $\mathcal{O}^*(1.9588^k)$ tijd en polynomiaal veel ruimte gebruikt.

Hierna gebruiken we dezelfde aanpak in een iets andere setting. Voor het Edge Dominating Set probleem en een aantal gerelateerde kantdomineringsproblemen geven we algoritmen die $\mathcal{O}(1.3226^n)$ tijd en polynomiaal veel ruimte gebruiken. Deze resultaten zijn net als eerdere resultaten over kantdomineringsproblemen gebaseerd op het opsommen van bepaalde minimum vertex covers in een graaf. Om onze resultaten

te verkrijgen introduceren we in deze algoritmen een nieuwe reductieregel die we gebruiken om minder van deze vertex covers op te hoeven sommen. De gebruikte aanpak leidt ook tot algoritmen voor k-Edge Dominating Set en een aantal gerelateerde problemen die $\mathcal{O}^*(2.4006^k)$ tijd en ruimte gebruiken.

Voor het Independent Set probleem geven we snellere algoritmen op grafen met een kleine gemiddelde graad. We geven een algoritme voor Independent Set op grafen met een gemiddelde graad van ten hoogste drie dat $\mathcal{O}(1.08537^n)$ tijd en polynomiaal veel ruimte gebruikt. Dit resultaat is gebaseerd op een grote gevalsanalyse waarbij de gemiddelde graad een belangrijke rol speelt in de looptijdanalyse. We gebruiken dit resultaat ook om snellere algoritmen te verkrijgen voor Independent Set op grafen met een grotere gemiddelde graad. Dit leidt uiteindelijk tot een algoritme voor het algemene Independent Set probleem dat $\mathcal{O}(1.2114^n)$ tijd en polynomiaal veel ruimte gebruikt.

In het derde deel van het proefschrift introduceren we een nieuwe techniek die we *inclusion/exclusion-based branching* noemen. Deze techniek gebruikt het principe van inclusie/exclusie als branchingregel die gebruikt kan worden in een standaard 'branch and reduce' algoritme. Inclusie/exclusie kan op deze manier gemakkelijk gecombineerd worden met andere algoritmische methodieken zoals verschillende branchingregels, reductieregels, of aanpakken op ijle grafen gebaseerd op boombreedte (treewidth). Gebruikmakend van onze nieuwe techniek geven we twee algoritmen voor #Dominating Set: een algoritme dat $\mathcal{O}(1.5673^n)$ tijd en polynomiaal veel ruimte gebruikt en een algoritme dat $\mathcal{O}(1.5002^n)$ tijd en ruimte gebruikt. We gebruiken deze aanpak ook om resultaten te behalen voor Domatic Number, Red-Blue Dominating Set, Dominating Set beperkt tot een aantal graaf klassen, en in een volledig andere context om een resultaat te behalen voor Disjoint Connected Subgraphs.

Hierna breiden we bovenstaande techniek uit zodat deze ook met partiële vereisten overweg kan. In de context van graafdomineringsproblemen wil dit zeggen dat we de techniek uitbreiden om van toepassing te zijn op partiële domineringsproblemen in grafen. We noemen deze uitbreiding van onze techniek *extended inclusion/exclusion-based branching*. Door gebruik te maken van deze uitbreiding construeren een algoritme voor Partial Dominating Set dat $\mathcal{O}(1.5673^n)$ tijd en polynomiaal veel ruimte gebruikt, een algoritme voor Partial Dominating Set dat $\mathcal{O}(1.5014^n)$ tijd en ruimte gebruikt, en algoritme voor k-Set Splitting dat $\mathcal{O}^*(1.8213^k)$ tijd en polynomiaal veel ruimte gebruikt.

In het vierde deel van het proefschrift beschouwen we dynamisch programmeeralgoritmen die werken op graafdecomposities. Hier combineren we standaardaanpakken op boomdecomposities, branchdecomposities, en cliquedecomposities (ook bekend als k-expressies) met varianten van het 'fast subset convolution' algoritme. We geven algoritmen op boomdecomposities met boombreedte k voor een breed scala aan graafdomineringsproblemen die allen $\mathcal{O}^*(s^k)$ tijd en ruimte gebruiken; hier is s het aantal 'states' dat nodig is om partiële oplossingen van het probleem te representeren. Deze algoritmen maken gebruik van een nieuwe techniek die we *generalised fast subset convolution* noemen. Op deze manier construeren we algoritmen voor de $[\rho, \sigma]$-domineringsproblemen en een klasse van overdekkings-, pakkings- en partitioneringsproblemen met cliques. Hieronder vallen onder andere een algoritme voor Dominating Set dat $\mathcal{O}^*(3^k)$ tijd en ruimte gebruikt en een algoritme voor #Perfect Matching dat $\mathcal{O}^*(2^k)$ tijd en ruimte gebruikt.

Dezelfde aanpak leidt tot algoritmen voor Dominating Set, Total Dominating Set en Independent Dominating Set op cliquedecomposities die $\mathcal{O}^*(4^k)$ tijd en ruimte gebruiken. Op branchdecomposisties geven we algoritmen voor een breed scala aan graafdomineringsproblemen die $\mathcal{O}^*(s^{\frac{\omega}{2}k})$ tijd en ruimte gebruiken door de aanpak verder uit te breiden met het gebruik van asymmetrische states op knopen en snelle matrixvermenigvuldigingen.

Tot slot geven we in dit proefschrift nog enkele interessante open problemen voor toekomstig onderzoek.

Curriculum Vitae

Johan M. M. van Rooij was born on June 19$^{\text{th}}$ 1983 in Schaijk, The Netherlands. In 2001, he received his VWO-diploma from the Titus Brandsma Lyceum in Oss.

From 2001 to 2006, he studied computer science at Utrecht University and followed the master's program in Algorithmic Systems. His master's thesis was entitled "Design by Measure and Conquer: An $\mathcal{O}(1.5086^n)$ Algorithm for Dominating Set and Similar Problems". From 2001 to 2007, he also studied mathematics at Utrecht University, obtaining his master's degree in the free variant. He completed both studies Cum Laude.

In 2007, he started as a PhD student in the Department of Information and Computing Sciences at Utrecht University under the supervision of dr. Hans L. Bodlaender and prof. dr. Jan van Leeuwen.