Treewidth : structural properties and algorithmic insights

Treewidth

structural properties and algorithmic insights

Lambertus Marchal

This book was typeset by the author using LaTeX.

Treewidth: structural properties and algorithmic insights
© Lambertus Marchal, 2012

Published by Universitaire Pers Maastricht
ISBN 978 94 6159 134 0
Printed in the Netherlands by Datawyse

Treewidth

structural properties and algorithmic insights

Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van Rector Magnificus,
Prof. mr. G.P.M.F. Mols,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op donderdag 5 april 2012 om 14.00 uur

door

Lambertus Marchal

UNIVERSITAIRE
PERS MAASTRICHT

Promotor:
Prof. dr. ir. C.P.M. van Hoesel

Copromotor:
Dr. A. Grigoriev

Beoordelingscommissie:
Prof. dr. R.J. Müller (voorzitter)
Dr. A. Berger
Dr. H.L. Bodlaender (Universiteit Utrecht)
Prof. dr. ir. A.M.C.A. Koster (RWTH Aachen University)

Dit onderzoek werd financieel mogelijk gemaakt door Maastricht Research School of Economics of Technology and Organizations (METEOR).

Acknowledgements

This dissertation would not have been realized without the help of several individuals who in one way or another extended their valuable assistance and contributed to its creation and completion. I would like to take the opportunity to acknowledge some of them here.

First, I want to express my gratitude to my promotor, Stan van Hoesel. On an enjoyable spring day in 2005, he succeeded to kindle my interest in a challenging topic, the full scope of which is still beyond my mind to grasp. Throughout the years of my doctoral studies he provided his expertise, while allowing me the room to work in my own way. His patience and reassuring composure have benefited me greatly.

Furthermore, I am heartily thankful to my copromotor, Alexander Grigoriev, for the many sessions that got things going again when I got stuck at writing. His creativity has been an inexhaustible source of new research directions to explore. Writing a dissertation can at times be a daunting and frustrating process. His ability to put things into perspective has always been most welcome. One could not wish for a more enthusiastic supervisor.

Most of the research that led to this dissertation has been conducted in close collaboration with several co-authors. I am indebted to Hans Ensinck and Arie Koster for their valuable contributions to respectively Chapter 3 and 4 of this dissertation.

I would like to thank my colleagues for their useful suggestions and for creating a pleasant work atmosphere in Maastricht. I thank my friends and family for providing me with the necessary distractions from writing.

Finally, my biggest thanks go to Natalya. She has been much more than the co-author of Chapters 5 and 6. Her loving support, encouragement and leading example have been the inspiration to complete this work. I feel very privileged to have her at my side.

Bert Marchal
's-Gravenhage, February 2012

Contents

Chapter 1

Introduction

Graphs are structures that are used to model a broad spectrum of problems that arise from real-world situations. These problems can originate from a variety of applications such as data analysis, electrical circuit design, logistics, scheduling, social networks, telecommunication and many others. To solve such a problem using a graph theoretic approach, one constructs a graph that represents the real-world setting of the problem in a mathematical way. Once this is done, the problem is stated in graph theoretical terms and an algorithm taking the graph as an input is used to generate a solution to the problem. Finally, the mathematical solution obtained this way must be translated back into the real-world context.

The running time of an algorithm depends heavily on the difficulty of the problem itself, but also on the complexity of the graph that models the framework of the problem. Some problems are easy in the sense that there is an algorithm that can solve them in an efficient way, independent of the structure of the input graph. In this context, efficiency of the algorithm usually means that the running time of the algorithm only grows polynomially in the size of the input graph. However, a lot of practically relevant problems are hard, meaning that the running time of algorithms solving them grows at least exponentially in the size of the input graph.

When dealing with hard problems, one might settle for a close-to-optimal solution that can be obtained in polynomial time. One can also try to exploit the structure of the input graph. It turns out that a lot of problems that are hard on general input graphs, can still be solved efficiently on input graphs that possess some special structural property. Based on properties as 'drawable in the plane without edges crossing' or 'all vertices having k neighbors', graphs are categorized in a vast number of classes. One graph class in particular that facilitates many otherwise hard problems, is the class of trees. Trees are connected graphs that do not contain cycles. The ease with which some hard problems can be solved on trees, often carries over to input graphs that are in some sense 'treelike'. A parameter

that measures the similarity of a graph to a tree is the treewidth of the graph. The lower its treewidth, the more the graph resembles a tree and the faster many problems can be solved on this particular graph.

Algorithms that exploit the treelike property of graphs often take as input a so called tree decomposition of the graph rather than the graph itself. A graph has usually numerous tree decompositions, each having their own width. The lower the width of a tree decomposition, the faster the algorithm runs on this tree decomposition. The ability to construct tree decompositions that have low width is therefore highly valuable when dealing with hard problems. The treewidth of the graph equals the lowest width over all tree decompositions of a graph. Unfortunately, constructing a tree decomposition of lowest possible width (and thus determining the treewidth of the graph) is a hard problem in itself for general graphs. This thesis deals with various aspects of treewidth and tree decompositions.

In the remainder of this chapter we provide some background information on the theory of graphs and algorithms on graphs. In particular we focus on the role that treewidth and tree decompositions play in many of such algorithms. The basic terminology that will be utilized throughout the thesis is clarified in Chapter 2. In Chapter 3, we present an exact algorithm that determines the treewidth of a graph. A heuristic for bounding the treewidth from above is introduced in Chapter 4. The relationship between treewidth of planar graphs and some other parameters of planar graphs will be examined in Chapter 5. To conclude the thesis, various algorithms are presented in Chapter 6 that illustrate how tree decompositions can be exploited to solve a particular graph problem.

1.1 Graph theory

A graph is a mathematical object that captures the notion of connectivity between a set of objects. The objects are represented by the vertices of the graph and a pairwise relation between two objects is represented by an edge between the vertices that correspond to the objects. In mathematics and computer science, graph theory is the study of graphs.

Some say that graph theory originated from a puzzle that was once posed by the townsfolk of Königsberg, Prussia in the early 1700's, see e.g. (16). Königsberg was built largely on an island in the Pregel river. The island is located near where two branches of the river join, and the borders of the town spread over to the banks of the rivers. Between the four land masses that the town occupied, seven bridges had been erected, as is illustrated in the leftmost picture in Figure 1.1.

The people of Königsberg supposedly posed the following question: Is it possible to take a walk through town, crossing each of the seven bridges just once? In 1736, Swiss mathematician Leonhard Euler, having heard of the problem, used the graph in the rightmost picture from Figure 1.1 to show that such a walk could not be made. The four land masses

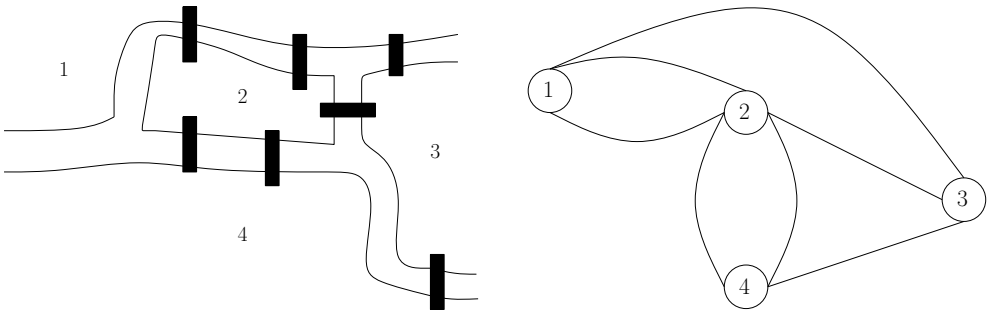


Figure 1.1: the seven bridges of Königsberg and a graph representing the setting

are represented by vertices in the graph and every bridge between two land masses is represented by an edge between the two vertices that correspond to the land masses. The degree of a vertex in a graph is the number of edges emanating from the vertex. Euler's argument was quite simple: suppose one could make such a walk, then there are at least two land masses that neither form the start nor the beginning of the walk. Whenever such a land mass is entered during the walk, it has to be left again to finish the walk. Hence there must be at least two vertices in the graph that have even degree, which is not the case.

Graphs have since then become an essential means for transforming real-life settings into abstract mathematical models. Problems that arise from such real-world settings are often decision problems or optimization problems. A decision problem is a question that can be answered by either "yes" or "no", depending on the input parameters. An optimization problem is a problem of optimizing (either maximizing or minimizing) some value based on the input parameters, for example to find the shortest route between two cities on a road map. Optimization problems can be cast as decision problems, e.g., instead of asking for the length of the shortest route between two cities one can ask whether between the two cities a route exists of length at most k, for some integer k.

For this specific problem, the real-world setting would be the network of roads between the two cities. A graph theoretical representation of this setting could be a graph in which the vertices represent all intersections and ends of the roads while the edges represent the road segments between all intersections and ends of the roads. Simple models like this normally allow for many extensions to make them more conform to the real-world setting. One might for example assign a weight to each edge that denotes the length of the corresponding road segment. For more information on graph theory and its applications, we refer to (57) and (68).

1.2 Algorithms and complexity

Apart from their purpose, algorithms can differ in running time and memory space they require to fabricate an answer. The demand for these resources is usually expressed in the input size of the problem, i.e., the number of bits needed to denote the input. For problems on graphs, the input size of the problem is usually the number of vertices in the graph. In computer science, an algorithm is called a polynomial-time algorithm if the time needed to run it grows only polynomially in the size of the input. If its running time grows exponentially in the input size, the algorithm is called an exponential-time algorithm. Polynomial-space and exponential-space algorithms can be defined in a similar manner. Very roughly speaking, polynomial algorithms are fast and efficient while exponential algorithms are slow or require too much memory to be useful in practice. For an overview of some algorithms on graphs, we refer to (107).

Based on the time and space requirements of algorithms to solve them, problems on graphs are clustered into a number of complexity classes. At the moment of writing this thesis, there are nearly 500 complexity classes (see (1) for an exhaustive overview) of which we will mention only a few here. One of the most fundamental complexity classes is the class P. It contains all decision problems that can be solved in polynomial time. As a rule of thumb, we say that P is the class of computational problems which are 'efficiently solvable' or 'tractable'. A superset of P is the class NP of so called "nondeterministic polynomial-time" problems. NP is defined as the set of decision problems for which a possible solution can be verified in polynomial time. Class P is a subset of NP, but there are a lot of graph problems in NP for which no polynomial-time algorithms are known. For this reason it is widely believed that P is a strict subset of NP.

A problem in NP is said to be NP-complete if a polynomial-time algorithm for this problem would imply the existence of polynomial-time algorithms for all other problems in NP. This is why NP-complete problems are considered to be the hardest problems in NP. For more details on the theory of NP-completeness, we refer to (63).

A problem is called NP-hard if it is as least as hard as any other problem in the class NP. More formally, a problem H is NP-hard if and only if there is an NP-complete problem L that is in polynomial-time reducible to H. This basically means that if some oracle would give us a solution to H, then we can solve L (and thereby any problem in the class NP) in polynomial time. We refer to (76) for an elaborate list of practically relevant NP-hard optimization problems.

Figure 1.2 displays the relation between the classes P, NP, NP-complete and NP-hard for the case where $P \neq NP$ versus the case where $P=NP$. One of the central topics in this thesis is the determination of the treewidth of a graph. The decision version of this problem, i.e. "given a graph G and an integer k, decide whether the treewidth of G is at most k", was

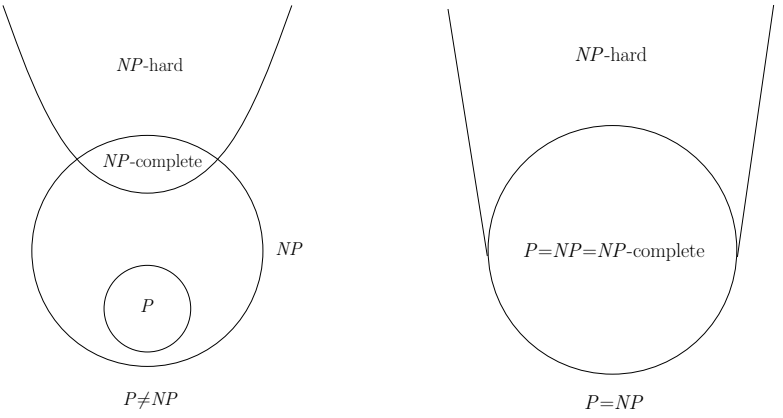


Figure 1.2: Euler diagram for problems in *P*, *NP*, *NP*-complete and *NP*-hard

shown to be *NP*-complete in (7). This implies that treewidth determination is an *NP*-hard problem.

Parameterized complexity is a branch of computational complexity theory that focuses on classifying problems according to their difficulty with respect to multiple input parameters. While in the classical setting the complexity of a problem is measured as a function in the number of bits of input, in parameterized complexity, it is measured as a function in two or more input parameters.

A number of problems that are *NP*-hard in the classical setting, are so called *fixed parameter tractable* in parameterized complexity. We speak of fixed parameter tractability of a decision problem if the function measuring its complexity is exponential in only one (small) input parameter k and polynomial in the input size of the problem. By fixing the value of the input parameter k, we say that the problem is parameterized in k. While doing so, the exponential factor disappears from the computational complexity of the problem, making the problem tractable. The class of fixed parameter tractable problems is denoted by *FPT*.

A major incentive for investigating the treewidth problem is that a lot of *NP*-hard problems on graphs are in the class *FPT*, where the treewidth of the graph plays the role of the fixed parameter, i.e., these problems are tractable when they are parameterized by treewidth.

1.3 Dealing with *NP*-hardness

In the previous section we indicated that determination of treewidth is an *NP*-hard problem. Various ways of dealing with *NP*-hard problems have been suggested over the course of time. The most uncomplicated is by brute force, i.e., just enumerate and evaluate all possible

solutions and pick the best one. The amount of time needed limits the usability of this approach to only small graph instances. Methods like branch-and-bound are based on the same principle of enumerating all solution candidates. However, they do this in a systematic way in which subsets of fruitless candidates are discarded in massive numbers. Although they are able to handle *NP*-hard problems on larger instances than the brute force method, their running times are still at least exponential. Indeed, assuming that $P \neq NP$, there are simply no polynomial-time algorithms that return exact solutions to *NP*-hard problems.

A more practical way to cope with *NP*-hard problems is therefore using approximation algorithms or heuristics. Approximation algorithms are efficient and guarantee that the returned solution is at most a factor (either fixed or depending on some input parameter) away from optimal. Heuristics are quick and the solution they return is usually reasonably good, but can in the worst case be arbitrarily far away from optimal. Their performance typically relies on educated guesses and common sense rather than on mathematically well-founded methods.

A third approach to *NP*-hard problems is by taking advantage of certain properties of the input graph. It is well known that many problems that are *NP*-hard in general become efficiently solvable when they are restricted to input graphs that belong to a certain graph class. For an exhaustive list of graph classes, the complexity of recognizing these classes and a lot of additional information, we refer to (45; 66).

One property that has proven to be particularly useful in this context is for the input graph to be a tree. Most real-world scenarios can only be modeled by graphs that are more complex than trees. This is where treewidth comes into play. The treewidth of a graph is a measure for how close the graph resembles a tree. The lower the treewidth, the more similar the graph is to a tree. *NP*-hard problems that can be efficiently solved on trees are also easily solvable on graphs that have low treewidth, because there are so called FPT-algorithms (Fixed Parameter Tractable) that solve these problems in time that is only exponential in the treewidth of the graph.

1.4 Treewidth

The notion of *treewidth* that will be used throughout this thesis was introduced by Robertson and Seymour in a series of papers on graph minor theory, see (95). Independently, Arnborg and Proskurowski introduced the equivalent notion of *partial k-trees*, i.e., a graph has treewidth at most k if and only if it is a partial k-tree (8).

There is little doubt that treewidth is one of the most important concepts that was introduced in the field of mathematics and computer science during the last decades. Ever since their introduction, treewidth and closely related notions like pathwidth and branchwidth have been the subject of study by numerous scientists in the field. The hardness of deter-

mining the treewidth of a graph makes the topic challenging from a theoretical viewpoint. From a more practical angle, the study of treewidth attracted a lot of interest because it can serve as a mathematical tool in dealing with other *NP*-hard combinatorial problems.

The definition of treewidth invented by Robertson and Seymour is based on the notion of *tree decomposition*. A tree decomposition of a graph G is a way to represent G as a tree. It consists of a tree T and a set of subsets of vertices of G. Each subset forms a node of T and is usually referred to as a 'bag'. The tree T and its bags form a valid tree decomposition of graph G, if the following three conditions hold:

1. Each vertex of G is present in at least one bag of T.

2. For each edge of G, there is at least one bag in T that contains both end vertices of the edge.

3. For each vertex v of G, the bags of T that contain v form a connected subtree of T.

The *width* of a tree decomposition is equal to the number of vertices in its largest bag minus one. The treewidth of a graph is defined to be the minimum width over all tree decompositions of the graph.

Another way to describe treewidth utilizes the notion of *chordalization*. A chordalization of a graph can be obtained by adding edges to the graph (if necessary) until for all cycles in the graph there is an edge connecting two vertices that are non-adjacent on the cycle. A subset of vertices in a graph that are pairwise connected by an edge is called a clique. The width of a chordalization is equal to the size of the largest clique in the chordalization minus one. The treewidth of a graph equals the minimum width over all its chordalizations. Formal definitions of tree decomposition and chordalization will be introduced in Chapter 2.

Trees contain no cycles, so they form their own chordalization and having a largest clique of size two (omitting graphs on a single vertex), the treewidth of a tree equals one. Cliques are also chordalizations of themselves and hence the treewidth of a clique on n vertices is equal to $n-1$. As another example, the infamous Petersen graph is shown as the leftmost picture in Figure 1.3. For the Petersen graph, it is known that the treewidth equals 4. The tree decomposition and chordalization of the Petersen graph that are also depicted in Figure 1.3 both have a width of 4 and hence they are optimal.

1.5 Use of tree decompositions

In general, solving a problem becomes easier (faster in terms of running time) when the graph instance gets smaller. Based on this idea, hard problems can often be solved by a so called "divide-and-conquer" strategy. Given a problem on a graph, the idea is to recursively

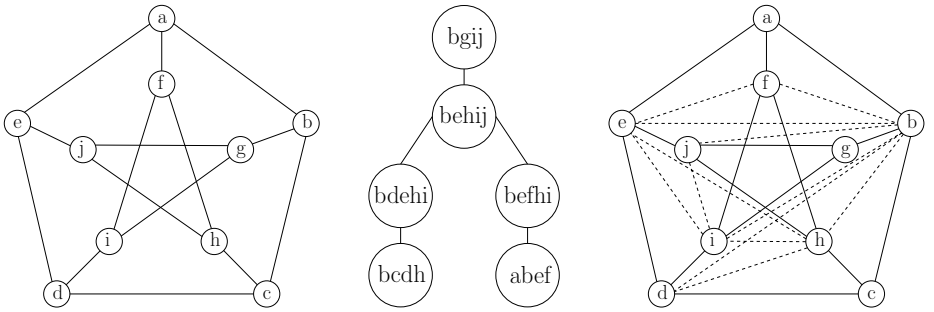


Figure 1.3: From left to right are depicted the Petersen graph, a tree decomposition of the Petersen graph and a chordalization of the Petersen graph.

split up the input graph into a number of smaller graphs. On each of these small graphs a problem of the same nature is defined that can be solved quickly.

To obtain the solution for the original problem, the solutions on the small graphs have to be joined together in a consistent way. However, the quality of a decision that is made in one small part of the input graph will very likely depend on the decisions that were made in other parts of the graph. The effect of a decision made in one part of the graph propagates to another part via all paths in the graph connecting the two parts. The more such paths there are, the more ways there are to combine solutions from the two parts. This makes it hard to find a solution for the combined parts that is feasible. Combining solutions to the two separate parts into an optimal solution for the combined parts is even harder.

In dealing with how to combine the partial solutions the property of the input graph having low or in some sense 'bounded' treewidth turns out to be algorithmically helpful. Getting some insight as to why this is the case can best be realized by means of simple trees. By absence of any cycles, two parts of a tree are connected to each other via just one single path. Hence, the influence of a local decision made in one part of the tree to a decision in another part of the tree can be captured by information concerning only the unique path between the two parts. To some extent, this easiness carries over to graphs that have low treewidth.

Consider a graph G on which some problem is defined. Assume we are given a tree decomposition of G of width k, where k is a small constant. By deleting an internal bag from the tree decomposition we split up the tree into at least two subtrees. Consider the subgraphs of G that are induced by the vertices in these subtrees. They may overlap, but the bag that was deleted from the tree decomposition contains all information that is needed to combine solutions on the subgraphs into a feasible solution for the problem on G. The internal bag of a tree decomposition thus serves as a separator in the graph and also as an interface between the two induced subgraphs. We can keep decomposing the subgraphs

by deleting other bags from the subtrees until all subgraphs have size at most $k+1$. Since k is small, solutions to the subgraphs can easily be obtained. The interfaces between the subgraphs also have size at most $k+1$ and hence different solutions to the subgraphs can be efficiently combined into an optimal solution to the problem on original graph G.

A class of graphs is said to have *bounded treewidth* if their treewidth is bounded from above by a constant that does not depend on the size of the graph. Bounded treewidth graphs have tree decompositions of bounded width. For an overview of classes of graphs that have bounded treewidth, we refer to (27; 66).

Many problems that are *NP*-complete on general graphs are solvable in polynomial or even linear time on bounded treewidth graphs (6; 8; 9; 26; 90). The most important characterization of *NP*-complete problems that can be solved on graphs of bounded treewidth was derived by Courcelle. In (52), he obtained as a fundamental result that all *NP*-complete problems that can be expressed in Monadic Second-Order Logic (MSOL) are linear-time solvable on graphs of bounded treewidth, if a bounded width tree decomposition of the graph is given. Algorithms solving these problems typically apply a divide-and-conquer strategy on a bounded width tree decomposition.

1.5.1 when treewidth is not bounded

What if we want to solve problems on graphs that do not have bounded treewidth? Does it make sense to use a tree decomposition based approach for solving *NP*-hard problems on these graphs where construction of an optimal tree decomposition is an *NP*-hard problem in itself? These are some reasonable questions that will be addressed in this subsection.

First of all, graphs that represent real-life settings like communication networks often have low treewidth. Although they are not necessarily part of a bounded treewidth class, they are typically quite sparse and are therefore treelike. An optimal tree decomposition can then often be found in acceptable time bounds and a tree decomposition based approach for solving the problem can be efficiently applied.

Furthermore, if an exact method to find an optimal tree decomposition is not quick enough, other methods (see Section 1.7.2 and 1.7.3) can be used to obtain tree decompositions of reasonably low width. It might very well be the case that the tree decomposition based approach to solve the problem is already fast enough for our purposes when we use this non-optimal tree decomposition.

A third reason for using a tree decomposition based technique to solve *NP*-complete problems is the re-usability of tree decompositions. It happens that several hard problems must be solved on the same graph instance. Once a good tree decomposition of such a graph is available, it can be used over and over for solving multiple problems. In this context, it will probably pay off to spend some more time and effort in constructing a single good tree decomposition.

1.6 Dynamic programming on tree decompositions

Dynamic programming is an algorithm design paradigm that systematically applies the before-mentioned divide-and-conquer strategy. The method was invented in 1953 by applied mathematician Richard E. Bellman. Tree decompositions lend themselves very well for application of dynamic programming algorithms.

Consider a problem on input graph G and suppose a tree decomposition of G is available. By choosing an arbitrary bag as the *root* of the tree decomposition, one can specify a relation between each pair of bags in the tree decomposition. Any bag X on the unique path from bag Y to the root (including the root itself) is called an *ancestor* of bag Y. If X is an ancestor of Y, then Y is a *descendant* of X. If bag X is an ancestor of bag Y and X and Y are connected by an edge, X is said to be a *parent* of Y and Y a *child* of X. Bags in the tree decomposition that have no descendants are called *leaves*.

The idea of the dynamic programming approach is to associate to each bag X the subgraph of G that is induced by all vertices in X and its descendants. For each bag in the tree decomposition, one computes a table that contains sufficient information to solve the problem on the subgraph of G corresponding to this bag. The tables are computed in a bottom-up manner, which means that one first has to compute the tables for the leaves. Tree decompositions can be made in such a way that the leaves are sufficiently small, hence computing the tables for the leaves is often trivial. To compute the table for any other bag, one just needs the information from the tables of its children. Tree decompositions can be made in such a way that computing the table for a bag from the tables of its children can be done very efficiently. Since the graph corresponding to the root coincides with graph G, the table of the root will contain enough information to determine the solution to the problem on G.

The maximum size of a table and the maximum time needed to compute a table are exponential only in the width of the tree decomposition. The dynamic program thus runs in time polynomial in the input size of the graph if and only if the graph has bounded treewidth. Problems that allow for this approach are thus fixed parameter tractable with respect to parameter treewidth.

As an example, let us consider the Weighted Independent Set problem. For this problem we are given a graph G with vertex weights for each vertex. We are looking for a subset S of vertices in G such that the vertices in S are pairwise non-adjacent and the sum of the weights of the vertices in S is maximized. Practical applications of this problem appear among others in information retrieval, signal transmission analysis, scheduling, coding theory and wireless networks. The problem is known to be *NP*-hard for general graphs. For trees however, it is solvable in linear time. In (34), a dynamic programming algorithm on a tree decomposition is presented that runs in time that is only exponential in the width of the tree decomposition. The Weighted Independent Set problem is thus fixed parameter

tractable with respect to treewidth.

For more details on dynamic programming on tree decompositions and for an overview of other graph decision problems for which this technique has been successfully applied, we refer to respectively (34) and (18).

1.7 Constructing tree decompositions

In the previous sections, it became clear that the availability of tree decompositions of low width can considerably speed up the process of solving many *NP*-complete problems on graphs. Unfortunately, constructing tree decompositions of low (let alone minimum) width is a task that is far from trivial.

The construction of a tree decomposition of width k is obviously at least as hard as deciding whether the treewidth of G is at most k, which is already an *NP*-complete problem. Computing treewidth and constructing tree decompositions of minimum width are *NP*-hard problems even if we restrict the input graphs to graphs of bounded degree (25), cocomparability graphs (7; 71), bipartite graphs (78) or the complements of bipartite graphs (7).

Despite these troubling facts, there are many cases for which treewidth can be efficiently determined or approximated.

1.7.1 exact methods

Over the last few decades, several graph properties have been shown to facilitate an efficient determination of the graph's treewidth. One of these properties is for the graph to be *chordal*. Chordal graphs are graphs that are triangulated. Determining the treewidth of a chordal graph allows for a polynomial-time algorithm, since finding the largest clique in a chordal graph is an easy problem. Other graph classes for which the treewidth can be determined in polynomial time include permutation graphs (22), circular-arc graphs (104), circle graphs (78) and distance hereditary graphs (46). In (41; 42), an algorithm was presented that determines the treewidth of a graph in time, polynomial in the number of its minimal separators. More examples of polynomial time algorithms for treewidth of graphs from various classes can be found in (28; 29; 47; 53; 77; 80).

Exact algorithms for treewidth that do not assume or exploit any properties of the input graph require at least exponential time. Based on the results from (41; 42), an exact algorithm for treewidth that runs in $O^*(1.9601^n)$ was obtained in (60), where n is the number of vertices in the input graph. For a survey of exact algorithms for *NP*-hard problems and for explanation on the $O^*()$ notation, we refer to (110). The result from (60) was improved in (61) and (62), decreasing the running time to $O^*(1.7549^n)$. The algorithm however requires

exponential space. The fastest exact algorithm for treewidth that takes only polynomial space is also from (62) and runs in $O^*(2.6151^n)$ time.

A well studied case is when the integer k is a fixed constant. Constructive algorithms for this case can be useful from a practical point of view, i.e., if the treewidth is at most k, a tree decomposition of width at most k must be returned. The first constructive polynomial time algorithm for the fixed parameter problem was found in (7). Although during later improvements upon it the running time of the algorithm was decreased to linear (see e.g., (20; 23; 24; 86; 87; 93)), the complexity of the algorithm still contains a hidden constant that is at least exponential in k, restricting its usability only to very small values of k (e.g., $k \leq 4$). Hence the quest remains for efficient algorithms for the fixed parameter case that are also practical from an implementation viewpoint.

For small graphs the treewidth can often be found in reasonable time using a branch-and-bound method. Experiments with constructive methods for treewidth were published in (64) and (102). By terminating the algorithm after a specific time bound and reporting the best solution so far, branch-and-bound algorithms can also be used as heuristics to obtain upper bounds on the treewidth of larger graphs. In Chapter 3 of this thesis, a branch-and-bound algorithm will be presented that builds triangulations of the input graph. In the branching step, potential fill-in edges are either added or denied in the triangulation. As will be explained in Chapter 2, a triangulation of width k can easily be transformed into a tree decomposition of width k.

1.7.2 approximation algorithms

Although constructing an optimal tree decomposition in reasonable time is often impossible, there are many algorithms that approximate the treewidth. Some of them are polynomial even when k is not bounded, others are exponential in k. In (21), a polynomial-time approximation algorithm was given that returns a tree decomposition with width at most $O(\log n)$ times optimal. This result was improved upon in (5) and (44), where polynomial-time approximation algorithms are presented with approximation ratio $O(\log k)$.

Constant performance ratio approximation algorithms exist for some special graph classes, see for example (43) where a polynomial 2-approximation algorithm was given for the treewidth of graphs that do not contain asteroidal triples. Treewidth can also be approximated within a constant factor for planar graphs. This is a consequence of the polynomial-time algorithm given by Seymour and Thomas (101) for computing the parameter branchwidth in planar graphs, whose value approximates treewidth within a factor of $\frac{3}{2}$. Whether constant ratio approximation algorithms exist for the treewidth problem on general graphs still remains a famous open problem.

Several approximation algorithms exist for treewidth that run in time exponential in k, the treewidth of the graph. These are also called fixed parameter approximation algorithms,

since they are polynomial only when k is fixed. They either give a tree decomposition of width at most ck for some constant $c \geq 1$, or tell that the treewidth is more than k. For some examples of them, we refer to (5; 12; 87).

1.7.3 heuristics

Fixed parameter approximation algorithms become less practical when k increases. Moreover, approximation algorithms that are polynomial even when k is unbounded tend to be very time consuming on big graph instances.

When a guaranteed performance ratio is not of primary importance, upper bound heuristics should be considered. They are often much quicker than approximation algorithms and although their performance can theoretically be exponentially bad, experiments have shown that these heuristics often perform very well in practice.

Many upper bound heuristics for treewidth share the same basic structure. A triangulation of the graph is constructed by visiting all vertices of the graph in some order and adding fill-in edges between the neighbors of the vertex at hand that are not yet visited. Different criteria for selecting the next vertex to visit lead to different heuristics. We refer to (83) for computational experiments on several upper bound heuristics.

Another type of upper bound heuristic is based on local search and was proposed in (81). The idea is to start with an arbitrary tree decomposition or triangulation and to reduce its width by making small local changes in the configuration of the bags in the tree decomposition (or cliques in the triangulation). In Chapter 4 of this thesis, we present a local search heuristic that splits the largest bag in a tree decomposition into smaller bags, while maintaining the validity of the tree decomposition. Our experiments show that this technique can lead to significant reduction in the width of tree decompositions resulting from other upper bound heuristics when they are used as a starting solution for the local search heuristic. Combining several types of upper bound heuristics may thus be beneficial.

1.8 Some applications

Tree decompositions of bounded width are interesting from a theoretical viewpoint, because all *NP*-complete problems that can be expressed in MSOL can theoretically be solved in linear time using them, see (51; 52). Considering that there are often huge constant factors concealed in their linear running times, algorithms exploiting the bounded width tree decompositions are not always practical. That being said, there are many examples in which tree decompositions have been successfully exploited. In this section, we will highlight a number of them.

In wireless communication networks, frequencies must be assigned to transmitters in

such a way that interference between the signals of different transmitters is avoided. In (81) and (82), optimal solutions were obtained to some hard frequency assignment problems in wireless networks. This was done by the use of dynamic programming techniques on a tree decomposition of the graph that models interference constraints in the network. In (84), a similar approach was used to solve small and medium-sized instances of the more general class of partial constraint satisfaction problems. The class PCSP contains a diversity of problems, such as generalized subgraph problems, MAX-SAT, Boolean quadratic programs and map coloring problems. For large instances of this class, the technique was shown to provide good lower bounds within reasonable time and memory limits.

A class that can be seen as a generalization of many hard problems (like Independent (Dominating) Set, Induced Bounded Degree Subgraph, Induced p-Regular Subgraph, Perfect Matching Cut and k-Colorability), is the class of Vertex Partitioning Problems. Given a graph, a Vertex Partitioning problem queries whether its vertices can be partitioned into a number of sets such that the sets comply to some particular conditions. In (105), it was shown that a large class of vertex partitioning problems is fixed parameter tractable when parameterized by treewidth. The FPT-algorithms exploit a bounded width tree decomposition of the input graph.

A dominating set in a graph is a subset D of the vertices such that every vertex not in D is connected via an edge to at least one vertex in D. A vertex cover of a graph is a subset C of the vertices such that each edge in the graph is incident to at least one element of C. The problem of finding the smallest vertex cover or dominating set in a graph has applications for example in sensor networks and social networks. A tree decomposition based approach enabled the authors in (4) to solve the Vertex Cover problem on planar graphs to optimality. In (2) and (3), tree decomposition based FPT-algorithms are suggested for the Dominating Set problem on planar graphs as well as for several other domination-like problems.

Practical applications of tree decompositions can also be found in network reliability. The reliability of a network measures the probability that communication in the network is possible, given probabilities for each of the elements in the network (e.g., links, routers) to break down. Many problems that arise in this context are *NP*-hard on general graphs. However, information and communication networks like Internet and telephone networks typically have low treewidth, simply because a tree is the cheapest way to connect a set of sites. In (91) and (111), tree decomposition based linear-time algorithms are proposed for several network reliability problems on graphs of bounded treewidth.

Another application that we bring up here concerns probabilistic networks, also called (Bayesian) belief networks. These networks are directed acyclic graphs that represent a set of random variables and their conditional (in)dependencies. They are used in decision support systems and expert systems. The value of a vertex v in the network depends on the value of all vertices that have an arc to v. Computing the conditional probabilities of all

vertices is called probabilistic inference, which is an *NP*-hard problem in general. In (88), an algorithm is presented that solves the problem in $O(n2^w)$ time by dynamic programming on a tree decomposition of an auxiliary graph of the network, where w is the width of the tree decomposition.

Recent applications also emerge from the field of Bioinformatics. Consider for example the problem of searching for some queried sequence in a Ribonucleic Acid (RNA) family with pseudoknots. Pseudoknots are secondary structures in RNA containing two stem-loop structures in which half of one stem is intercalated between the two halves of another stem. Searching for queried sequences of RNA in a structure with pseudoknots turns out to be a hard problem when using methods as Hidden Markov Models and Covariance Models. In (103), graphs are used to model both the RNA family with pseudoknots as well as the queried sequence of RNA. The problem of finding the queried sequence segment then boils down to the subgraph isomorphism problem, which is *NP*-hard in itself. However, since a graph modeling an RNA family with pseudoknots typically has treewidth that is only slightly higher than 2, the problem of recognizing queried sequences of RNA can be efficiently solved using dynamic programming on a tree decomposition of the graph. Searches that required several months with other methods can now be accomplished in days. In (112), some key problems in protein structure prediction were efficiently solved with the help of tree decompositions.

1.9 Thesis outline

The basic terminology that will be utilized throughout the thesis is clarified in Chapter 2. Next to formal definitions of treewidth, tree decomposition and some more notions that will be used throughout the thesis, Chapter 2 also covers some fundamental results that will be assumed as pre-knowledge in the remainder of the thesis.

In Chapter 3, we present a branch-and-bound algorithm that determines the treewidth of a graph and returns a tree decomposition of optimal (minimum) width. Since the algorithm is exact and therefore not efficient, its practical usability is limited to relatively small graph instances only. However, by terminating the algorithm after a fixed time bound, it can serve as a way to obtain lower and upper bounds on the treewidth of larger graphs.

When exact methods fail, it is necessary to resort to approximation algorithms or heuristics. We develop a treewidth upper bound heuristic that will be the subject of Chapter 4 in this thesis. The heuristic utilizes a local search technique and takes as input an arbitrary tree decomposition. It then manipulates the tree decomposition by making a series of local changes to it, while preserving its defining properties. The ultimate goal of the manipulation steps is to obtain a reduction in the width. Implementation of the local search heuristic reveals that the width of tree decompositions (even for those resulting from other upper

bound heuristics) can often be significantly decreased in an acceptable amount of time.

Subsequently, we turn our attention to the class of planar graphs. This class consists of all graphs that can be embedded in the plane without crossing edges. Planar graphs do not have bounded treewidth and the question whether or not computing the treewidth of planar graphs is *NP*-hard is still unanswered. The treewidth of a planar graph is however related to some other parameters of planar graphs, namely its branchwidth (also holds for general graphs) and the size of its largest square grid-minor. The nature of these relationships is examined more closely in Chapter 5 of this thesis. In this chapter, we also introduce a methodology that can be used to determine the size of a largest square grid-minor for some planar graphs.

In the final chapter of the thesis, Chapter 6, we investigate the problem of excluding a set of graphs as subgraph of input graph G by deleting a (minimum) number of edges from G. This problem is *NP*-hard for general graph instances. We show that the problem (depending on the type of subgraphs) is fixed parameter tractable when parameterized by the treewidth. Moreover, we show that an optimal solution to the problem can be efficiently approximated if the input graph is planar. The algorithms in this last chapter employ dynamic programming techniques that are very illustrative for the value of tree decompositions in dealing with *NP*-hard problems.

1.10 Published material

Part of the material presented in this thesis has been published or has been accepted for publication in refereed journals or conference proceedings. The following publication is based on Chapter 4 of this thesis.

> Stan P. M. van Hoesel and Bert Marchal. *Finding good tree decompositions by local search*. Electronic Notes in Discrete Mathematics, **32** (2009), pp. 43–50.

An extended version of the work described in Chapter 5 of this thesis has been published in the following publication.

> Alexander Grigoriev, Bert Marchal and Natalya Usotskaya. *On Planar Graphs with Large Treewidth and Small Grid-Minors*. Electronic Notes in Discrete Mathematics, **32** (2009), pp. 35–43.

Finally, Chapter 6 of this thesis forms the source of the following two publications.

> Alexander Grigoriev, Bert Marchal and Natalya Usotskaya. *Algorithms for the Minimum Edge Cover of H-Subgraphs of a Graph*. In Proc. of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010), Lecture Notes in Computer Science, **5901**, Springer (2010), pp. 352–464.
>
> Alexander Grigoriev, Bert Marchal, Ioan Todinca and Natalya Usotskaya. *A note on planar graphs with large width-parameters and small grid-minors*. Accepted in Discrete Applied Mathematics, 2012.

Chapter 2

Preliminaries

In this chapter we clarify the majority of the terminology that will be used throughout this thesis. Some terms that are utilized only occasionally will be introduced later, at the point where they are needed. We commence with some terms and notions that regard standard graph theory. After that some terminology will be stated concerning the subject of planar graphs and subsequently the notion graph minor will be explained. We round this chapter off with formal definitions of treewidth, tree decompositions and some related terms.

2.1 Graph terminology

A *graph* G is a pair $(V(G),E(G))$, where $V(G)$ is a finite set of *vertices* and $E(G)$ is a (multi)-set of *edges*. Figure 2.1 displays a graph on ten vertices and fifteen edges, known as the Petersen graph. If there is no ambiguity about which graph G we deal with, we will refer to

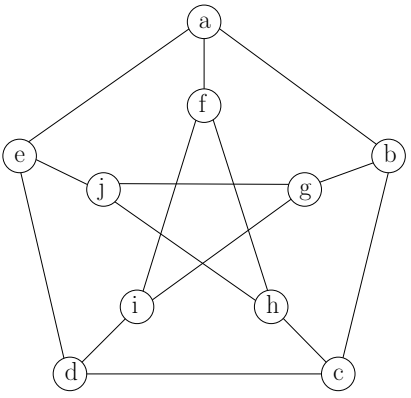


Figure 2.1: the Petersen graph

$V(G)$ and $E(G)$ simply as V and E. Following the convention in graph theory, we use $n(G)$ (or simply n) to denote the number of vertices in G and $m(G)$ (or simply m) to denote the number of edges in G, i.e., $n = |V|$ and $m = |E|$. An edge $e \in E$ is a two-element (multi)-set of vertices from V, i.e., $e \in V \times V$. If an edge e joins two vertices, then these vertices are called the *end vertices* of e. If u and v are the end vertices of the edge e, then e is formally denoted by $\{u,v\}$. For the sake of readability, we will refer to the edge $\{u,v\}$ simply as uv in this thesis. Two edges are *disjoint* if they have no end vertex in common. An edge e and an end vertex of e are said to be *incident* to one another. Two vertices that are joined by an edge are called *adjacent* vertices. In the Petersen graph as depicted in Figure 2.1, vertices a and b are adjacent and they are the end vertices of edge ab. In this example, vertex f is incident to edges af, fh and fi. The *open neighborhood* $N(v)$ of a vertex $v \in V$ in G is the set of vertices adjacent to v in G. In the Petersen graph, e.g., $N(f) = \{a,i,h\}$.

The *degree* $d(v)$ of a vertex $v \in V$ is the number of edges incident to v in G. The minimum degree over all vertices in G is denoted by $\delta(G)$, i.e., $\delta(G) = \min_{v \in V} d(v)$. Similarly, $\Delta(G)$ denotes the maximum degree in G, i.e., $\Delta(G) = \max_{v \in V} d(v)$. For the graph in Figure 2.1, $\delta(G) = \Delta(G) = 3$. An edge $e \in E$ is called *directed* if the pair of vertices denoting the edge is ordered and e is said to be *undirected* if it is represented by an unordered vertex pair. A graph G is undirected if all of its edges are undirected. An edge is called a *self-loop* if its two end vertices coincide. When $E(G)$ is a set rather than a multi-set and does not contain self-loops, G is said to be a *simple* graph. The graph in Figure 2.1 is simple and undirected. Unless stated otherwise, the graphs in this thesis are both simple and undirected.

A *path* in a graph G is a sequence of vertices from V such that each vertex in the sequence is adjacent to the next vertex in the sequence. For finite paths, the first vertex is called the *start vertex* and the last vertex is called the *end vertex*. All other vertices in a path are called *internal* vertices. A *cycle* is a path for which the start vertex and the end vertex coincide. A path with no repeated vertices is called a *simple* path, and a cycle with no repeated vertices or edges aside from the necessary repetition of the start and end vertex is a *simple* cycle. A graph is *acyclic* if it does not contain any cycle. The *length* of a path is the number of edges in the path, counting multiple edges multiple times. The sequence $afhjgb$ is a simple path of length 5 in the Petersen graph from Figure 2.1. Two paths in a graph are *independent* if they have no internal vertices in common. The *distance* between two vertices in a graph G is the length of the shortest path in G connecting the two vertices. The maximum distance between two vertices in the Petersen graph equals 2.

If $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$, then $G' = (V', E')$ is a *subgraph* of $G = (V, E)$, written as $G' \subseteq G$. If G' is a subgraph of G, then G is a *supergraph* of G'. a subgraph G' of G is said to be induced by V' if E' contains all edges from E for which both end vertices are in V'. The subgraph of $G = (V, E)$ that is induced by $V' \subseteq V$ is denoted by $G[V']$. For $v \in V$, we denote by $G \setminus v$ the graph that is obtained from G by deleting v and all edges incident to v,

i.e., $G \setminus v = G[V \setminus \{v\}]$. For $e \in E$, by $G \setminus e$ we denote the graph that results from G when we delete edge e.

A *clique* in G is a subset of V for which all vertices are pairwise adjacent. A clique S in G is called a *maximal* clique if no strict superset of S is a clique. A clique S in G is called a *maximum* clique if there is no clique in G that has bigger size. The size of a maximum clique in G is denoted by $\omega(G)$. A graph on n vertices that forms a clique on n vertices is called a *complete* graph and is denoted by K_n. An independent set in G is a subset of V in which the vertices are pairwise non-adjacent. Maximal and maximum independent sets can be defined analogously to maximal and maximum cliques. The set $\{a, f\}$ is a maximal (and maximum) clique and set $\{b, e, h, i\}$ is a maximal (and maximum) independent set in the Petersen graph from Figure 2.1.

A graph is said to be *connected* if there is a path between each pair of distinct vertices of the graph. The Petersen graph is an example of a connected graph. A *connected component* S of G is a maximal subgraph of G that is connected, i.e., each supergraph of S that is a subgraph of G is disconnected. Most problems on graphs can be cracked by solving them separately on each connected component of the graph. It is for this reason that we only consider input graphs in this thesis that have exactly one connected component, i.e., we assume that our input graphs are connected. Connected graphs that are acyclic are called *trees*.

2.2 Planar graphs

A graph G is called a *planar graph* if it can be drawn in the plane in such a way that its edges intersect only at shared end vertices. A drawing of a planar graph G in the plane is called a *planar embedding* of G. The graph from Figure 2.1 is non-planar, since it can not be embedded in the plane without crossing edges. The regions that are bounded by the edges in a planar embedding of a planar graph are called *faces*. If we embed a planar graph on the plane, there is always one face that is unbounded. This is called the *outer face*. All other faces are called *inner faces*. The number of faces in a planar embedding is denoted by the letter f. The next theorem implies that for a connected planar graph G, the parameter f does not depend on the embedding of G.

Theorem 2.2.1. ***(Euler's formula)*** *Let G be a connected planar graph with n vertices and m edges. Then for every planar embedding of G it holds that*

$$n - m + f = 2.$$

A graph G is called *outerplanar* (or 1-outerplanar) if there is a planar embedding of G in which each vertex is incident to the outer face. For $k > 1$ a planar embedding is *k-outerplanar* if removing the vertices incident to the outer face results in a $(k-1)$-outerplanar

embedding. A graph G is *k-outerplanar* if there exists a k-outerplanar embedding of G. The notion of k-outerplanar graphs was introduced by Baker, see (11). The smallest integer k for which G is k-outerplanar is called the *outerplanarity index* of G. As an example, the graph in Figure 2.2 has outerplanarity index 2, since there is no 1-outerplanar embedding of G.

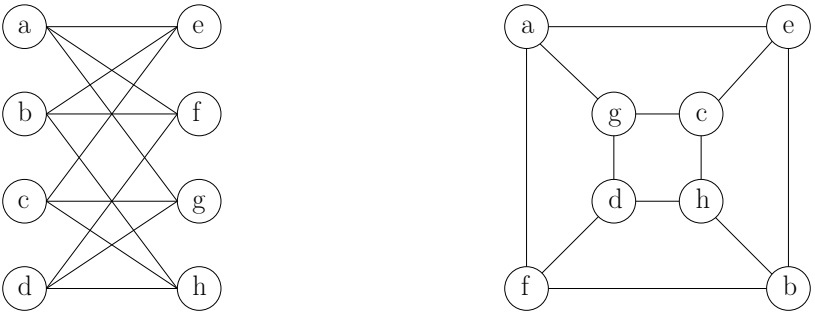


Figure 2.2: a planar graph G and a 2-outerplanar embedding of G

In (15), it is shown that the outerplanarity index of an arbitrary planar graph can be determined in polynomial time. The following theorem is from (75):

Theorem 2.2.2. *Given a planar graph G, the outerplanarity index k of G and a k-outerplanar embedding of G can be found in $O(n^2)$ time.*

The following theorem from (27) relates the outerplanarity index of a graph to its treewidth:

Theorem 2.2.3. *The treewidth of a k-outerplanar graph is at most $3k - 1$.*

2.3 Graph minors

We first explain the notion of an edge contraction in a graph. While doing so, we follow the terminology from (57). Subsequently, we give a formal definition of a graph minor. Minors can be obtained from a graph by a series of vertex deletions, edge deletions and edge contractions in any order. They play an important role in the characterization of many families of graphs.

Informally, contraction of edge $e = uv$ replaces vertices u and v by a new vertex that is adjacent to all neighbors of both u and v. In this thesis, we restrict ourselves to simple graphs and hence we enforce that after contraction, the new vertex is connected to its neighbors via a single edge and not via a multi-edge. This is illustrated in Figure 2.3.

As is formalized in the next definition, we use G/e to denote the graph that is obtained from G by contracting edge e.

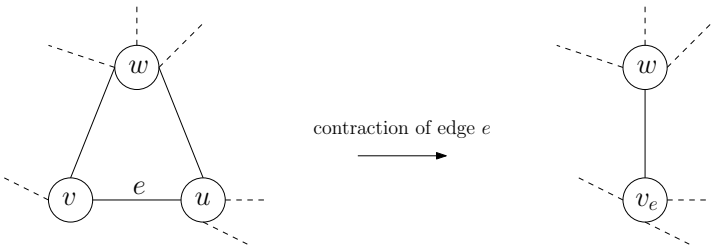


Figure 2.3: Before contraction both u and v are adjacent to w. After contraction of e, the new vertex v_e is connected to w only via a single edge.

Definition 2.3.1. *Let be given a graph $G = (V, E)$ and let $e = uv \in E$. Then G/e is the graph (V', E') that results from G after contracting edge e, where $V' = (V \setminus \{u, v\}) \cup \{v_e\}$ (with v_e the new vertex) and*

$$E' = \{\, xy \in E \mid \{u, v\} \cap \{x, y\} = \emptyset \,\} \bigcup \{\, v_e x \mid ux \in E \setminus \{e\} \text{ or } vx \in E \setminus \{e\} \,\}.$$

Two graphs G and H are said to be *isomorphic* if there exists a bijection $f := V(G) \to V(H)$ such that any two vertices u and v are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.

Definition 2.3.2. *A graph M is a* minor *of G, written as $M \preceq G$, if M is isomorphic to a graph that can be obtained from a subgraph of G by doing a number (zero or more) of edge contractions.*

Figure 2.4 shows a graph G, a subgraph H of G and a minor M of G. M can be obtained from H by contracting the dashed edges.

A family $\mathscr{F}$ of graphs is said to be closed under taking graph minors if for every graph G in $\mathscr{F}$, all minors of G are also element of $\mathscr{F}$. Planar graphs and the family of bounded treewidth graphs are examples of graph-minor-closed families.

We say that we *subdivide* an edge $e = uv$ in G if we add a new vertex w to G and replace the edge uv by edges uw and vw. A *subdivision* of G can be obtained from G by a series of subdivisions of the edges of G. If a graph M has a subdivision that is isomorphic to a subgraph of G, then M is called a *topological minor* of G. Since in Figure 2.4, graph $H \subseteq G$ is a subdivision of M, M is a topological minor of G.

In 1930, Kuratowski (85) provided a characterization of planar graphs in terms of two forbidden topological minors; the complete graph K_5 and the complete bipartite graph $K_{3,3}$. Seven years later in (108), Wagner proved that a graph is planar if and only if it does not have K_5 or $K_{3,3}$ as a regular minor. This led him to the bold conjecture that the set of forbidden minimal minors of any infinite graph-minor-closed family of graphs is finite. A proof of the theorem by Robertson and Seymour was completed in 2004 with the publication

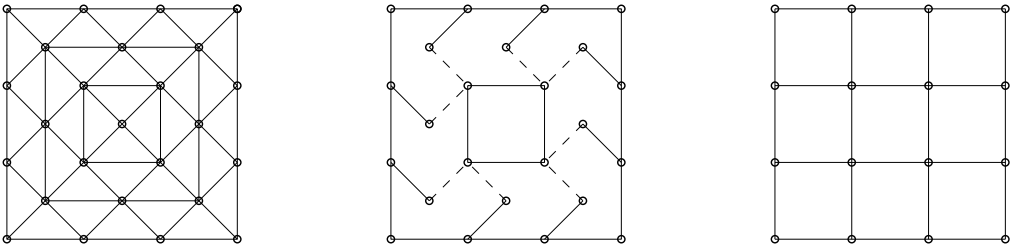


Figure 2.4: graph G, subgraph $H \subseteq G$ and minor $M \preceq G$.

of the last in a series of twenty papers (see, among others, (94; 95; 96)) running to over 500 pages and spanning almost 20 years.

2.4 Treewidth

In this section, we formally define treewidth and a few related terms. The notions of treewidth and tree decomposition were introduced by Robertson and Seymour in (94). Apart from their definitions, we introduce some fundamental lemma's regarding treewidth that are essential for some lemma's and theorems in later chapters of the thesis.

A *tree decomposition* of a graph G is a pair (T,X), where T is a tree and $X = (X_t : t \in V(T))$ is a family of subsets of $V(G)$, with the following properties:

- $\bigcup_{t \in V(T)} X_t = V(G)$.
- $\forall uv \in E(G)$, $\exists t \in V(T)$ such that $\{u,v\} \subseteq X_t$.
- For $t,t',t'' \in V(T)$, if t' is on the unique path in T between t and t'' then $X_t \cap X_{t''} \subseteq X_{t'}$.

For $t \in V(T)$, the set X_t is also referred to as a *bag* of the tree decomposition (T,X). The *width* of the tree decomposition (T,X) is $\max_{t \in V(t)}(|X_t| - 1)$.

Definition 2.4.1. *The* treewidth $tw(G)$ *of graph G is the minimum width over all tree decompositions of G.*

The following useful fact about tree decompositions originates from (19).

Lemma 2.4.2. *Let (T,X) be a tree decomposition of graph G. Then for any clique S in G, there is a bag X_t in (T,X) for which $S \subseteq X_t$.*

The relation between treewidth of G and the treewidth of a minor of G can be captured in the following lemma, which is due to Bodlaender (27).

Lemma 2.4.3. *If $M \preceq G$, then $tw(M) \leq tw(G)$.*

A lot of research has been dedicated to the fixed parameter case for treewidth, i.e., check whether the treewidth of a graph is at most some constant w and if so, return a tree decomposition of width at most w. For an overview of this work we refer to (30). Finally, in (23) the following result was obtained.

Theorem 2.4.4. *Given a graph of treewidth at most w, a tree decomposition of width at most w can be obtained in linear time.*

From a practical viewpoint the algorithm is only useful for low values of w because of a big hidden constant in the O()-notation in the running time of the linear algorithm.

An edge which joins two vertices of a cycle but is not itself an edge of the cycle is called a *chord* of that cycle. A *chordless* cycle in G is an induced cycle in G, i.e., a cycle that forms an induced subgraph of G. A graph is called *chordal* (or *triangulated*) if it does not contain chordless cycles of length greater than 3, i.e., if all induced cycles in the graph are 3-cycles (also called triangles). A graph can be transformed into a chordal graph by adding edges to it up to the point where every cycle contains a chord. Edges that are added to achieve chordality are called *fill-in edges*. In this thesis, we denote the set of fill-in edges by F.

Definition 2.4.5. *A graph $H = (V, E \cup F)$ is called a* chordalization *(or* triangulation*) of $G = (V, E)$ if H is chordal.*

The *width* of a chordal graph H is equal to $\omega(H) - 1$, i.e., the size of the largest clique in H minus one. Using the notion of chordalization, treewidth can be alternatively defined in the following way.

Definition 2.4.6. *The treewidth $tw(G)$ of graph G is the minimum width over all chordalizations of G.*

The following result concerning chordal graphs comes from (98).

Lemma 2.4.7. *Given a chordal graph G, the size of a largest clique in G and hence $tw(G)$ can be determined in polynomial time.*

An *ordering* of the vertex set V is a bijection $\alpha : \{1, 2, \ldots, n\} \longleftrightarrow V$. If α is an ordering on n vertices, then we will also refer to α as $\alpha(1)\alpha(2)\ldots\alpha(n)$. A chordalization of $G = (V, E)$ can be obtained using vertex ordering α of V by application of Algorithm 2.1. In words, we run through the vertex ordering and turn the higher ordered neighbors of the vertex at hand into a clique by adding fill-in edges to G. Note that the added fill-in edges also define neighbor relations in subsequent steps of the algorithm.

The process of turning the higher ordered neighbors of vertex $\alpha(i)$ into a clique is often called the *elimination* of vertex $\alpha(i)$. In the context of chordalizations, vertex orderings

Algorithm 2.1: chordalization

Input: graph $G = (V, E)$ and vertex ordering α
Output: chordalization of G

for $i = 1$ *to* $n - 2$ **do**
  make $S_i = \{ v \in V \mid v \in N(\alpha(i)) \;\&\&\; \alpha(v) > i \}$ a clique by adding fill-in edges (if necessary);
end

are therefore often referred to as *elimination orderings*. An elimination ordering α on the vertices of G is called *perfect* if during Algorithm 2.1 no fill-in edges are added to G, i.e., if for all vertices $v \in V$ the set of higher ordered neighbors already forms a clique in G.

As an example, a graph is depicted in Figure 2.5 with a chordalization of the graph that is obtained using an elimination ordering on its vertices. Fill-in edges bd, bf, dh, fh, df are added during elimination of respectively the vertices a, c, i, g, b. While eliminating the last four vertices in the ordering, no fill-in edges need to be added.

Figure 2.5: graph G and chordalization H of G that is obtained via elimination ordering $\alpha = acgibdefh$.

A vertex v in a graph G is called *simplicial* if $N(v)$ induces a clique in G. It was shown in (89) that every non-empty chordal graph contains at least one simplicial vertex. Another classical fact is that chordal graphs are recursively simplicial, i.e., they contain a simplicial vertex and after removing this simplicial vertex, the subgraph is still simplicial. A perfect elimination ordering α of a chordal graph G can therefore be obtained by repeatedly selecting a simplicial vertex of G as the next vertex in α and deleting it from G.

2.5 Tree decompositions and chordalizations

In this thesis, we will use both tree decompositions and chordalizations as structures to bound and determine the treewidth of graphs. However, the two terms are closely related

in the sense that if we are given a graph G and a tree decomposition of G of width k, then it is easy to construct a chordalization of G of width k. Algorithm 2.2 shows how this can be done.

Algorithm 2.2: tree decomposition to chordalization

Input: graph G and tree decomposition (T, X) of G of width k
Output: chordalization of G of width k

for $t = 1$ *to* $|V(T)|$ **do**
    make X_t a clique in G by adding fill-in edges (if necessary);
end

Vice-versa, a chordalization of G can be used to construct a tree decomposition of G of the same width. One way to do this is by application of Algorithm 2.3. Note that this algorithm assumes a perfect elimination ordering (p.e.o.) of the chordalization H of G. Such a p.e.o. can be obtained using simplicial vertices as described above. The tree decomposition and

Algorithm 2.3: chordalization to tree decomposition

Input: graph G, chordalization H of G of width k and p.e.o. α of H
Output: tree decomposition of G of width k

Create a bag consisting of vertex $\alpha(n)$.;
for $i = n - 1$ *to* 1 **do**
    find bag X_i containing all higher ordered neighbors of $\alpha(i)$ in H.;
    if X_i *contains only the higher ordered neighbors of* $\alpha(i)$ **then**
        add $\alpha(i)$ to bag X_i.;
    end
    else
        make a new bag containing $\alpha(i)$ and its higher ordered neighbors in H and connect it to bag X_i.;
    end
end

chordalization of the Petersen graph as depicted in Figure 1.3 can be constructed using one another by application of Algorithms 2.2 and 2.3.

Chapter 3

Branch and Bound

This chapter deals with an exact method to determine the treewidth of graphs. The method that will be presented is a branch-and-bound algorithm that operates directly on the input graph and ultimately realizes chordalizations of the graph by adding fill-in edges to it. It is thus constructive in the sense that it not only determines the treewidth of a given graph instance, but also returns a chordalization of optimal (minimum) width. By application of Algorithm 2.3, such an optimal chordalization can be used to construct an optimal tree decomposition of the graph instance.

The content of this chapter is based on cooperation with Stan van Hoesel and Hans Ensinck.

3.1 Introduction

Finding chordalizations or tree decompositions of minimum width has been a central issue in algorithmic graph theory during the last decades. Its importance lies in the fact that many *NP*-hard problems can be solved efficiently using low-width tree decompositions with dynamic programming type methods. Examples of such problems are Vertex Coloring, Independent Set, and even Hamiltonian Cycle.

Courcelle (51) has shown that for each graph property that can be formulated in Monadic Second Order Logic (MSOL), there is a polynomial time algorithm that verifies if the property holds if a bounded width tree decomposition of G is available. Such algorithms operate in two steps. First a tree decomposition is constructed and then the problem is solved on this tree decomposition. The last step is usually done by some dynamic program for which both running time and memory space consumption are exponential in the width of the tree decomposition, but polynomial in the size of the graph.

Among the first authors applying this technique are Bern et al. (13). For other papers using this technique on a variety of problems, see for instance Wimer et al. (109) for some

of the pioneering papers with this type of algorithm, Arnborg and Proskurowski (8), Lauritzen and Spiegelhalter (88) for the inference problem in probabilistic networks, Telle and Proskurowski (105) for several vertex partitioning problems, Koster et al. (84) for partial constraint satisfaction problems (in particular frequency assignment problems) and Bodlaender and Koster (34) for the weighted independent set problem. See Arnborg (6) and Bodlaender (25) for surveys on treewidth algorithms and algorithms for intractable problems that are efficient when restricted to graphs of bounded treewidth.

Several graph classes admit polynomial time algorithms for determining their treewidth, e.g.chordal graphs, permutation graphs, circular arc graphs, circle graphs and distance hereditary graphs. The problem, however, when given an arbitrary graph G and an integer k, to determine whether the treewidth of G is at most k is NP-complete (7).

Most constructive methods focussing on upper bounding the treewidth of graphs are constant factor approximation algorithms or heuristics, see Section 1.7.2 and 1.7.3. The majority of exact algorithms for treewidth is non-constructive, see Section 1.7.1. There is however quite some recent literature focussing on constructive exact methods for determining the treewidth of a graph. These algorithms use either dynamic programming or branch-and-bound. Bodlaender et al. (32) give an overview on such algorithms and their theoretical and practical implications. Shoikhet and Geiger (102) constructed an algorithm that takes as input a graph G and an integer k. By building sets of minimal separators and so called fragments of G and by applying dynamic programming techniques, the algorithm returns an optimal chordalization of G or a valid statement that the treewidth of G is larger than k. Branch-and-bound is used by Gogage and Dechter (64) and in Bachoore and Bodlaender (10), among others. Their methods build perfect elimination orderings by adding vertices one by one in the branching process. Gogate and Dechter introduced a processing method to kill nodes using exchangeability of neighbors in the orderings. Bachoore and Bodlaender (10) added processing based on vertex disjoint paths between vertices. Their methods are practically suitable for graphs of up to 100 vertices and treewidth of no more than 10.

In this chapter, a new constructive algorithm for determining the treewidth of a graph is introduced. More specifically, a branch-and-bound algorithm is presented that uses a branching scheme in which fill-in edges are added or forbidden. By exploiting the knowledge about forbidden edges in any node of the branch-and-bound tree, new lower bound techniques for the treewidth are developed. Moreover, new processing rules are applied to limit the number of nodes that need to be visited in the branch-and-bound tree.

The remainder of this chapter is organized in the following way: in Section 3.2, some preliminaries and definitions are introduced, part of which will be used exclusively in this chapter. Section 3.3 attends to a special graph structure, the so called necklace, that forms the basis of several processing rules and lower bounds. Section 3.4 then describes the facets of branching and bounding in the algorithm. In Section 3.5, a number of processing rules

and some graph reduction techniques are introduced. A selection of the ideas presented in this chapter have been implemented and the resulting algorithm has been tested on several classes of graphs. The results of these experiments are presented in Section 3.6. Finally, conclusions and some directions for further research can be found in Section 3.7.

3.2 Preliminaries

Section 2.4 describes that a chordalization $H = (V, E \cup F)$ of $G = (V, E)$ can be obtained via an elimination ordering α on vertex set V by application of Algorithm 2.1. The following folklore fact provides a way to determine the width of such a chordalization H.

Lemma 3.2.1. *Let $H = (V, E \cup F)$ be the chordalization of G that is obtained via elimination ordering α by application of Algorithm 2.1. Then the width of H is equal to the maximum over all vertices $v \in V$ of the number of higher ordered neighbors of v in α at the moment that v is eliminated.*

Proof. The width of chordal graph H is equal to the size of its maximum clique minus one. Consider the higher ordered neighbors in α of any vertex v at the moment v is eliminated. Since v will form a clique in H together with these neighbors it follows that the maximum number of the higher ordered neighbors is a lower bound for the width of H. To see that the maximum clique in H is induced by some vertex v and its higher ordered neighbors in α at the moment of v's elimination, let C be a maximum clique in H and let w be the element from C that has the lowest order in α. After w's elimination, no edges are added to H that are incident to w. Also, during elimination of w, no edges incident to w are added to H. Therefore, all other elements of C (which are all higher ordered than w) are already incident to w at the moment w is eliminated. □

A perfect elimination ordering of a chordal graph can be found by recursively selecting simplicial vertices in the graph. By Lemma 3.2.1, the treewidth of a chordal graph thus equals the maximum degree of a simplicial vertex at the moment it is selected. This leads to the following trivial result:

Lemma 3.2.2. *Determining the treewidth of a chordal graph can be done in polynomial time.*

The algorithm described in this chapter builds chordalizations of some input graph $G = (V, E)$ by adding edges to it and by forbidding vertex pairs to be edges in the chordalization of G.

In the remainder of this chapter, the set $B = E \cup F$, the original set of edges in G plus the added edges, will be denoted as the set of *black edges*, and the set R, the vertex pairs forbidden in a chordalization of G, will be denoted as the set of *red edges*. The set of vertex

pairs that are neither in R nor in B, will be denoted as the set W of *white edges*. Given input graph $G = (V,E)$, nodes in the branch-and-bound tree thus correspond to graphs $S = (V,B,R)$, where $E \subseteq B$, and B, R, and W form a partition of the vertex pairs of G.

Definition 3.2.3. *$S' = (V,B',R')$ is called an* extension *of $S = (V,B,R)$ if $B \subseteq B'$ and $R \subseteq R'$.*

An extension of $S = (V,B,R)$ can thus be obtained from S by making some white edges black and/or red.

Definition 3.2.4. *$S' = (V,B',R')$ is called a* chordalization *of $S = (V,B,R)$ if S' is an extension of S and $H = (V,B')$ is chordal.*

Note that if S' is a chordalization of S and S is an extension of G, then S' is also a chordalization of G.

Definition 3.2.5. *The* treewidth *of $S = (V,B,R)$ is the minimum width over all chordalizations S' of S. If S has no chordalization, then $tw(S) = \infty$.*

In Figure 3.1, an example is given of a graph that has no chordalization.

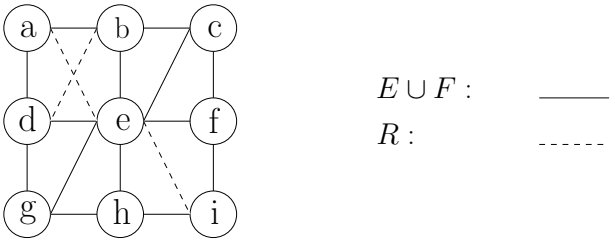


Figure 3.1: Depicted is the graph $S = (V,B,R)$. In all supergraphs of S, the cycle *abeda* is chordless. Therefore $tw(S) = \infty$.

Let it be clear that the red edges in $S = (V,B,R)$ are not real edges, so whenever a cycle, a path or the degree of a vertex in S are mentioned, they refer to the cycle, path or degree of the vertex in the corresponding graph $H = (V,B)$.

This section ends by introducing a lemma concerning chords in chordal graphs.

Lemma 3.2.6. *Let C be a cycle of length at least 4 in a chordal graph G and let v be a vertex on C. Then at least one of the following statements is true:*

- *The neighbors of v on C are connected by a chord.*
- *There is a chord of C incident to v.*

Proof. Proof by contradiction. Suppose there is a vertex v on C such that the neighbors u and w of v on C are not connected by a chord and no chords of C are incident to v. Then consider the shortest cycle C' in G containing path uvw and using only chords or edges of C. Since there are no chords in C incident to v and uw is not a chord of C, cycle C' must be a chordless cycle with length at least 4, contradicting the chordality of G. □

3.3 the Necklace structure

Both processing rules and bounding methods in this chapter depend heavily on the presence of a certain structure in the graph. This structure, which will be called a *necklace*, is introduced in this section and several lemmas are presented concerning chordalizations of graphs $S = (V,B,R)$ that contain necklaces.

Definition 3.3.1. *A path $P = v_1 \dots v_n$ of black edges in $S = (V,B,R)$ for which $v_i v_{i+2} \in R$ for $i = 1,\dots,n-2$ is called a* necklace *in S.*

If extension $S = (V,B,R)$ of $G = (V,E)$ contains a necklace, several conclusions can be drawn about possible chordalizations of S. They are summarized in the following set of lemmas that will later on be used to process graph S in a node of the branch-and-bound tree.

Lemma 3.3.2. *Let $P = v_1 \dots v_n$ be a necklace in $S = (V,B,R)$. Then in any chordalization of S, there are no black edges between vertices of P that are non-adjacent on P.*

Proof. Suppose that in some chordalization S' of S, there is a black edge between two vertices v_i and v_j that have distance of at least 3 on P. Select v_i and v_j as close as possible to each other on P. Then in S', $v_i \dots v_j v_i$ is a cycle of length at least 4, without any black chords. This contradicts the fact that S' is a chordalization of S. □

Lemma 3.3.2 is illustrated in Figure 3.2. The next lemma concerns a vertex in S that is connected to two vertices on a necklace in S.

Lemma 3.3.3. *Let $P = v_1 \dots v_{n-1}$ be a necklace in $S = (V,B,R)$ and let v_n be connected to both v_1 and v_{n-1}. Then in every chordalization of S, $v_i v_n$ will be a black edge for $i = 1,\dots,n-1$.*

Proof. By Lemma 3.3.2 the vertices of P will induce a path in the chordalization. Therefore, all chords of the cycle $C = v_1 \dots v_n v_1$ in the chordalization must be incident to v_n. This implies that in the chordalization, there are chords $v_i v_n$ for $i = 2,\dots,n-2$ in C. □

An illustration of Lemma 3.3.3 can be found in Figure 3.3. The last lemma of this chapter concerns a special type of cycle in $S = (V,B,R)$.

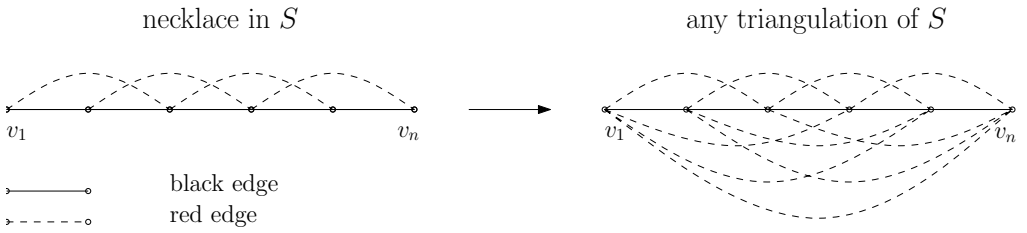


Figure 3.2: In any chordalization of S, there are no black edges between vertices of necklace P that are non-adjacent on P.

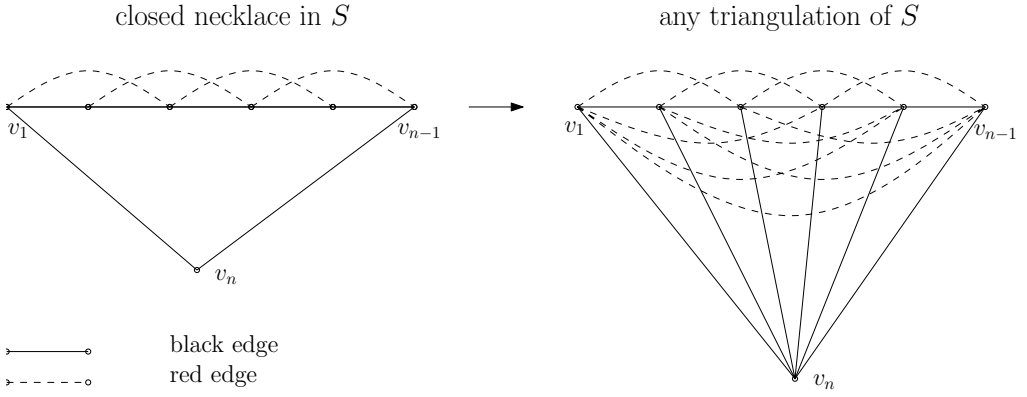


Figure 3.3: Let $P = v_1 \dots v_{n-1}$ be a necklace and $C = v_1 \dots v_n v_1$ be a cycle in S. Then in all chordalizations of S, cycle C has exactly $n-3$ chords, all incident to v_n.

Lemma 3.3.4. *Let $C = v_1 \dots v_n w v_1$ be a cycle of length at least 4 in $S = (V, B, R)$. If vertex w is connected with red edges to all vertices in C except its neighbors, then in all chordalizations of S, the edge $v_1 v_n$ will be a black edge.*

Proof. If in a chordalization of S, cycle C has no chords incident to w, then by Lemma 3.2.6, the neighbors of w on C must form a chord in the chordalization. □

3.4 Branch-and-Bound algorithm for treewidth

Branch-and-bound is the most widely used tool for solving large scale *NP*-hard combinatorial optimization problems. It is used here in a fairly standard way. Branch-and-bound algorithms rely on a systematic enumeration of all possible solution candidates. In this study, this implies that all potential chordalizations of the graph will, either explicitly or implicitly, be considered.

The branch-and-bound tree consists of nodes each representing a subproblem of the original problem. The tree is built by starting with one node (the root), which represents the original problem, with $S = (V, E, \emptyset)$. The subproblem in a node is to find the minimum width over all chordalizations of S, or equivalently, to determine the treewidth of S. In the process, nodes are split by extending graph S, whenever necessary. A node (the father) is split by selecting a white edge vw, and making two new subproblems or nodes (the children) in the tree, one in which vw is colored red, and one in which vw is colored black. Since, in any chordalization all vertex pairs are colored either red or black, it is clear that every feasible solution of the father node is present in one of the two children. At any stage of the tree, the leaf nodes, the nodes that have not been split, represent the whole set of feasible solutions, guaranteeing that the optimal solution to the root problem can also be found in one of these nodes. The splitting process is called *branching*. The selection of the edge on which to split, is called *branching variable/edge selection*.

For some of the leaf nodes, one might be able to conclude that no further examination is necessary. These nodes are labeled *inactive*, and no further splitting is necessary. The other leaf nodes are called *active*. Every time, after a node is split, a new active node has to be selected to be processed next. There are several orderings of the active nodes in which to select them. They are described below.

- Breadth First Search: here the subproblems are processed in the order of their creation. This strategy is seldom used.

- Depth First Search: here one always selects the last created subproblem. This is the easiest way of implementing the branching strategy. It also uses the least amount of memory, since the number of subproblems is relatively small.

- Best First Search: here the node is selected according to some criterion, to judge the potential of the node. Generally, in minimization problems the node with the smallest lower bound is selected. Note that both Depth First Search and Breadth First Search can be viewed as special cases of Best First Search using the tree depth within the Branch-and-Bound tree as criterion: taking highest depth and lowest depth as the selection criterion, respectively.

Before a node is split, it is processed with the techniques available, to find out whether it might still hold the optimal solution to the root problem. A standard way of processing is bounding. In a minimization problem such as determining treewidth, normally a good feasible solution (forming an upper bound) is known, i.e. a good chordalization of input graph G is known, with width val. If one can prove, with some kind of lower bounding technique, that for the subproblem at hand no chordalizations with value smaller than val can be found, then one does not need to examine this node any further; it is made inactive. This is the *bounding* procedure in the branch-and-bound algorithm. Besides bounding, other methods are used to draw conclusions about the subproblem at hand, such as the addition of red and black edges, and (almost) simplicial vertices. Together these ideas form the *processing* part in a node.

For all facets of the branch-and-bound technique that are mentioned here, ideas and methods will be described next, some of which are standard, others more problem specific.

3.4.1 branching rules

Branching in a node of the binary branch-and-bound tree is done by selecting a white edge in the graph corresponding to the node. Two children of the node at hand are then created by turning the selected white edge into a black edge in one child node respectively into a red edge in the other child node.

simple rules

The first two branching rules are pretty straightforward in the sense that they select a white edge e based on local properties of the graph in the subproblem at hand, and therefore require only little computation time.

The first rule selects a white edge vw where vertex v has the highest degree (wrt the black edges). w is then selected as the vertex with the highest degree (wrt the black edges) among the vertices for which vw is a white edge. Slightly different mechanisms are possible here: maximum total degree of v and w or including red edges in the selection process.

The second rule is based on extending some clique C, in which all edges are colored red or black. It starts off with a maximal clique C in the original graph. One can choose this initial C, for instance, by taking a good tree decomposition of G, and taking the largest bag. It starts then, hopefully, in the (dense) part of the graph which is the most complicated to chordalize and where it may be easiest to draw conclusions from coloring a white edge red or black. The vertex w outside C is chosen that has the fewest white edges to C. Then a white edge e between w and a vertex in C is chosen to branch on. Once all edges between w and C are defined, C can be extended by adding w to C.

probing based rules

Another way of branching is based on the processing rules that will be discussed in Section 3.5. While turning a white edge of S into a black or red edge, sometimes one can conclude that to obtain a chordalization of the resulting graph, other white edges have to be turned into black edges (or *cannot* be turned into black edges). By applying these processing rules recursively, the effects of fixing the color of one white edge can be far reaching. For edge $w \in W$, let w_b be the number of white edges in S whose color can be fixed after recursively applying the processing rules from Section 3.5 after w is colored black. Similarly, let w_r be the number of white edges in S whose color can be fixed after w is colored red. The probing based branching rule considers all white edges w in S and branches on the one for which $w_b + w_r$ is maximized. This rule greatly reduces the number of subproblems. However, the time won by pruning the subproblems will often be abolished by the time needed to compute $w_b + w_r$ for all white edges. Variants on this probing based rule are relatively easy to come up with. To save computation time, one might for example apply the processing rules only once instead of recursively.

3.4.2 bounding methods

As mentioned before, methods for bounding the treewidth in a node of the tree play a crucial role in our branch-and-bound algorithm. In this section, some novel techniques are presented for bounding the treewidth. They exploit information about the set of red edges in the extension $S = (V, B, R)$ of $G = (V, E)$ that corresponds to a node in the branch-and-bound tree.

upper bounding

To obtain an upper bound on $tw(G)$ that can be used from the very start of a run of the branch-and-bound algorithm, one might apply a quick upper bound heuristic, see Section 1.7.3. In case the results are not satisfying, one might consider to apply a more advanced heuristic like the local search heuristic that is the subject of Chapter 4 of this thesis.

Consider an extension $S = (V, B, R)$ of G and the graph S' that has the same set of black edges as S but an empty set of red edges. Any chordalization of S' is also a chordalization of G. If S' happens to be close to an optimal chordalization of G, an upper bound heuristic ran on S' will likely give a lower upper bound on $tw(G)$ than the same heuristic ran on G, simply because there is less room for deviating from an optimal solution. Therefore, in an attempt to improve upon the starting upper bound on $tw(G)$, it might be beneficial to run the upper bound heuristic from time to time on S' for some nodes deep in the branch-and-bound tree.

If one can verify that in some node the extension $S = (V, B, R)$ of $G = (V, E)$ is chordal, then obviously its width also provides an upper bound on $tw(G)$. Checking whether S is

chordal is straightforward and determining the treewidth of a chordal graph can be done in polynomial time, see Lemma 3.2.2.

lower bounding with R

The problem to decide whether an extension $S = (V, B, R)$ of $G = (V, E)$ has a chordalization can be described in the following way: decide whether a chordal graph S_C exists such that S is a proper subgraph of S_C and S_C is a proper subgraph of the graph obtained from S by making all white edges black. This problem is known as the chordal sandwich problem, which was proven to be *NP*-complete in (65). This destroys hope of effectively killing the subproblems for which no chordalization exist. Nonetheless, some techniques will be introduced here for bounding $tw(S)$ from below. If this lower bound on $tw(S)$ is equal to or larger than a known upper bound on $tw(G)$, the particular node can be set to inactive.

Any chordalization of $S = (V, B, R)$ can be obtained by applying some elimination ordering to S. A chordalization of S is an extension of S. Hence, at the moment of elimination of vertex v, there should not be any red edges between neighbors of v. Let V_b be the set of vertices v in S for which there are no red edges between neighbors of v in S. Based on the foregoing observations, the following lemma provides a lower bound on $tw(S)$:

Lemma 3.4.1. *The minimum degree in S of the vertices from V_b is a lower bound on $tw(S)$.*

To obtain a chordalization of S by use of an elimination ordering, one of the vertices from V_b must be eliminated first. This observation supports the following lemma:

Lemma 3.4.2. *Let C_b be the vertices in S that are adjacent to all vertices in V_b. Then C_b forms a clique in every chordalization of S and hence $|C_b| - 1$ is a lower bound on $tw(S)$.*

Next a second lower bound on $tw(S)$ will be described which is also based on the presence of red edges and can be seen as a generalization of a result from (49). Note again that a path in S refers to a path of which all edges are black.

Lemma 3.4.3. *Let $S = (V, B, R)$ and let $uv \in R$. Then the number of vertex disjoint paths between u and v in S forms a lower bound on $tw(S)$.*

Proof. If S has no chordalization, $tw(S) = \infty$ by definition and the claim is true. Otherwise, consider an arbitrary chordalization of S and a tree decomposition (T, X) of S of equal width that is derived from this chordalization by Algorithm 2.3. Since $uv \in R$, in (T, X) there is no bag that contains both u and v. Let X_u and X_v be the bags in (T, X) that contain respectively u and v such that all internal bags (if any) on the unique path from X_u to X_v in T contain u *nor* v. Then by the properties of a tree decomposition, bag X_u (the same holds for X_v) must contain an internal vertex of any vertex disjoint path between u and v in (the chordalization of) S. If there are w such vertex disjoint paths in S, then $|X_u| \geq w + 1$. The width of (T, X) and of the chordalization of S are thus at least w. □

To determine the maximum number of vertex disjoint paths between u and v in S, one can use a flow algorithm with capacities on the vertices.

lower bounding with necklaces

An improvement over the lower bound given by Lemma 3.4.3 can be obtained by using the result of Lemma 3.3.2.

Theorem 3.4.4. *Let $P = v_1 \dots v_n$ be a necklace in $S = (V,B,R)$. Then for any $i \in \{2,\dots,n-1\}$, the number of vertex disjoint paths between the vertex sets $V_1 = \{v_1,\dots,v_{i-1}\}$ and $V_2 = \{v_{i+1},\dots,v_n\}$ forms a lower bound for $tw(S)$.*

Proof. If S has no chordalization, $tw(S) = \infty$ by definition and the claim is true. Otherwise, consider an arbitrary chordalization of S and a tree decomposition (T,X) of S of equal width that is derived from this chordalization by Algorithm 2.3. By the definition of a necklace and by Lemma 3.3.2, all paths in the chordalization of S (and thus also in S itself) between a vertex from V_1 and a vertex from V_2 have at least one internal vertex. Let X_{i-1} and X_{i+1} be the bags in (T,X) that contain respectively $\{v_{i-1},v_i\}$ and $\{v_i,v_{i+1}\}$ such that all internal bags (if any) on the unique path from X_{i-1} to X_{i+1} in T do not contain v_{i-1} nor v_{i+1}. Note that any internal bag X_i on this path contains vertex v_i and that V_1 and V_2 are in different components of $T \setminus X_i$. By the properties of a tree decomposition, bag X_{i-1} (the same holds for X_{i+1}) must contain an internal vertex of any vertex disjoint path between a vertex from V_1 and a vertex from V_2 in (the chordalization of) S. If there are w such vertex disjoint paths in S, then $|X_{i-1}| \geq w+1$. The width of (T,X) and of the chordalization of S are thus at least w. □

Lemma 3.3.2 can also function as a way of bounding the treewidth from below; when a necklace is detected in S and two vertices at distance at least 3 on the necklace are connected by a black edge, then this implies that $tw(S) = \infty$ and the node can be made inactive. The same goes for Lemmas 3.3.3 en 3.3.4.

regular lower bound methods

Finally consider again the graph S' obtained from $S = (V,B,R)$ by deleting all red edges. Since the red edges in S are only restrictive for the number of possible chordalizations of S, any lower bound on $tw(S')$ is also a lower bound on $tw(S)$. By applying regular treewidth lower bound heuristics on S' one can thus also bound $tw(S)$ from below. Algorithm 3.1 which was proposed in (64) can therefore be used as a lower bounding procedure in the branch-and-bound method. Independently, the same algorithm was proposed in (33) under the name MMD+(min-d). In the latter paper, two more variants of the algorithm are proposed, called MMD+(max-d) and MMD+(least-c). In step 2a of the algorithm, MMD+(max-d)

contracts minimum degree vertex v to a maximum degree vertex u in $N(v)$ and algorithm MMD+(least-c) contracts v to a vertex u in $N(v)$ for which v and u have the least number of common neighbors. Experimental results in (33) show that MMD+(least-c) performs slightly better than MMD+(min-d) on several instances. In our implementation MMD+(min-d) is used, for the reason that it takes less computation time.

Algorithm 3.1: Algorithm Minor-Min-Width or MMD+(min-d)

Input: graph G
Output: a lower bound on $tw(G)$

1. $lb = 0$;
2. Repeat
   (a) Contract the edge between a minimum degree vertex v and $u \in N(v)$ such that the degree of u is minimum in $N(v)$ to form a new graph G'.
   (b) $lb = \max\{lb, degree_v(G)\}$.
   (c) Set G to G'.
3. until no vertices remain in G.
4. return lb.

For extensions S of G with only a low number of red edges, Algorithm 3.1 will provide better lower bounds on $tw(S)$ than the R-based lower bound algorithms, which exploit information about the set R. However, when $|R|$ increases, the R-based lower bound methods might be more competitive.

3.5 Rules for processing

The problem to determine $tw(S)$ in a node of the branch-and-bound tree becomes easier when more white edges are colored red or black. In this section, several rules are described for processing a graph S. The three lemmas from section 3.3 state that because of the presence of necklace structures, in a chordalization of S some white edges in S need to be black or cannot be black. In analogy with these three lemmas, three rules for processing a graph S are introduced below.

According to Lemma 3.3.2, a necklace in S will induce a path in any chordalization of S. Hence whenever a necklace is encountered in a subproblem, the branch-and-bound algorithm adds red edges to S between non-adjacent vertices of a necklace in S.

Assume that in graph S a cycle $C = v_1 \dots v_n v_1$ is found in which $P = v_1 \dots v_{n-1}$ forms a necklace. Then by Lemma 3.3.3, the algorithm simplifies the subproblem of determining $tw(S)$ in the node by connecting all vertices of P to v_n via a black edge.

Assume finally that in graph S, a cycle $C = v_1 \dots v_n w v_1$ exists of length at least 4 in which vertex w is connected with red edges to all vertices in C except its neighbors. Then by Lemma 3.3.4, in all chordalizations of S, edge $v_1 v_n$ will be black. Hence the algorithm will process the subproblem by coloring the edge black.

The three rules described above together form the set of so called necklace rules.

3.5.1 simplicial vertices

Another way of simplifying the subproblem of determining $tw(S)$ is by reducing the size of the graph S itself. In this section, some graph reduction rules are presented, based on simplicial vertices and almost simplicial vertices. The techniques described are an adaption of methods that are used for regular graphs (without set R), see e.g., (31).

Lemma 3.5.1. *Let v be a simplicial vertex in $S = (V,B,R)$ of degree $d(v)$ and let $S' = S \setminus \{v\}$. Then $tw(S) = \max\{d(v),\ tw(S')\}$.*

Proof. Since v and its neighbors form a clique in S of size $d(v)+1$, it follows that $tw(S) \geq d(v)$. S' is a minor of S, so by Lemma 2.4.3, $tw(S) \geq tw(S')$. Combining the two observations, if follows that $tw(S) \geq \max\{d(v), tw(S')\}$. To show that $tw(S) \leq \max\{d(v), tw(S')\}$, consider an optimal tree decomposition of S'. Since the neighbors of v form a clique in S', there will be a bag containing all neighbors of v. By adding a neighbor bag to this bag that contains v and all neighbors of v, one obtains a tree decomposition of S of width $\max\{d(v), tw(S')\}$. Hence $tw(S) \leq \max\{d(v), tw(S')\}$. □

The next lemma is the basis of a graph reduction rule with respect to almost simplicial vertices. First let us define contraction in the extended graph $S = (V,B,R)$ of G.

Definition 3.5.2. *A black edge uv in $S = (V,B,R)$ is said to be* contracted *to v if for all vertices w for which vw is a white edge:*

- *vw is added to B if $uw \in B$,*
- *vw is added to R if $uw \in R$,*

and u is deleted from S.

For an illustration of this definition, see Figure 3.4. In the following lemma, R_v denotes the set of vertices w for which $vw \in R$ in S and lb is some lower bound for $tw(S)$.

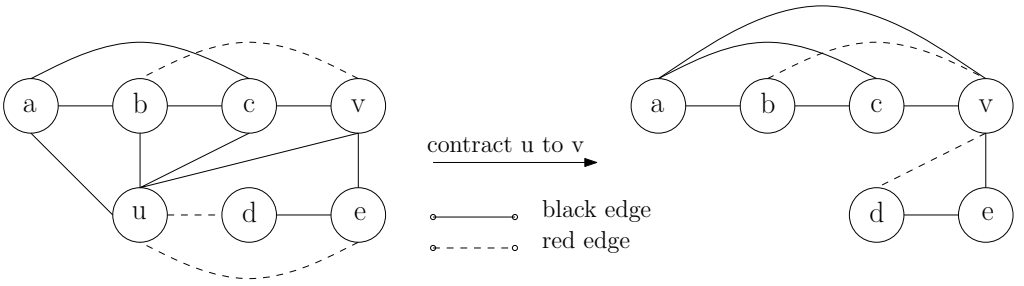


Figure 3.4: contraction of black edge uv to vertex v in graph $S = (V, B, R)$.

Lemma 3.5.3. *Let v be an almost simplicial vertex in S with $d(v) < lb$. Let w be the non-clique neighbor of v and let S' be the graph that is obtained from S by contracting vw to w. If $R_w \subseteq R_v$, then $tw(S) = tw(S')$.*

Proof. First let us show that $tw(S) \geq tw(S')$. Consider an optimal tree decomposition of S. Replace v by w in the bags of this tree decomposition and delete one w from each bag that contains two w's. The result is a tree decomposition of S' that has lower or equal width. Indeed, the subtree corresponding to w will still be connected, since there was a bag containing both v and w and all edge relations (from E and F) are still covered by a bag; w will be connected in S' to all vertices (except v) that were either a neighbor of w or v in S and in the new tree decomposition, for each such vertex there is a bag containing w and this vertex. Note that w will form a red edge in S' with all vertices that formed a red edge in S with either v or w. Since $R_w \subseteq R_v$ in S, neither of those vertices shares a bag with v in the tree decomposition for S, and therefore neither of those vertices will share a bag with w in the tree decomposition for S'. It follows that $tw(S) \geq tw(S')$.
Now let us show that $tw(S) \leq tw(S')$. Consider an optimal tree decomposition of S'. Since $R_w \subseteq R_v$ in S, all neighbors of v in S form a clique in S' and therefore there is a bag X in the tree decomposition of S' containing all neighbors of v in S. Make a bag Y containing vertex v and all its neighbors in S and attach it to bag X. The result is a tree decomposition of S. Indeed, there is no bag containing both end vertices of a red edge in S. The width of this tree decomposition is equal to the maximum of $tw(S')$ and the degree of vertex v in S, i.e., $tw(S) \leq \max\{d(v), tw(S')\}$. Since the degree of v in S is smaller than lb, the conclusion is that $tw(S) \leq tw(S')$. □

3.5.2 conflict graph

Finally, we propose the idea of keeping track of a so called *conflict graph*. A conflict graph should be constructed in such a way that it visualizes the conflicts in certain colorings of the white edges of the graph in the subproblem. The conflict graph should have two vertices for each white edge in the graph, one for the case that the edge is colored black and one for the case the edge is colored red. Suppose that e_1 and e_2 are potential edges in the input graph, then the conflict graph has vertices c_{1b}, c_{1r}, c_{2b} and c_{2r}. Vertex c_{1b} corresponds with the situation where e_1 is a black edge and c_{1r} to the situation where e_1 is a red edge. Suppose that as a result of the necklace lemmas, e_1 can not be colored black if e_2 is red, then the conflict graph has an edge $c_{1b}c_{2r}$. Conflicts that are detected by the processing rules in a node of the branch-and-bound tree remain valid for the children of this node. By apply the processing rules again in the child node, one has to determine the same conflicts in the child node all over again. The advantage of keeping track of a conflict graph is that from a node to its child node, the conflict graph has to be updated only in the local region where the branch-and-bound algorithm is operating at that moment. For this reason we believe that keeping track of a conflict graph would speed up the branch-and-bound algorithm.

3.6 Experimental results

The branch-and-bound algorithm has been tested on a number of graph classes. Implementation of the algorithm was done in C++ and experiments were run on a 2.4GHz Intel Core 2 CPU having 2 GB of RAM. The computational experiments focussed on grid graphs, queen graphs, and graphs derived with the procedure used for Mycielski graphs. These graphs can easily be extended to larger graphs with similar properties. In this way the behavior of the algorithm has be studied on similar instances of increasing size.

For all results in this section, Depth First Search has been used as the traversal strategy in the branch-and-bound tree. Furthermore, branching in the tree was done by the first branching rule as described in Section 3.4.1. Lower bounds were obtained by Algorithm 3.1. Upper bounds were found by checking for chordality in every subproblem. For some graph classes, initial upper bounds were passed as input to the algorithm. Finally, the graphs corresponding to the subproblems were processed according to the rules that are described in Section 3.5. At the moment of writing this chapter, the remaining ideas from this chapter have not yet been fully implemented. For this reason, no running times are displayed in the tables. Instead, the number of nodes visited in the branch-and-bound tree has been used as a performance criterion for our branch-and-bound algorithm.

The tables in this section use the following terminology. $|V|$ and $|E|$ denote respectively the number of vertices and edges in the graph, whereas ub and tw denote an upper bound

on the treewidth respectively the treewidth of the graph. Finally, columns #1, #2, and #3 display the number of processed subproblems for several variants of the algorithm:

- #1 denotes the number of processed subproblems when the algorithm is used in its most basic form; except for removal of simplicial vertices (which is needed to determine whether the graph is chordal), no processing is used,
- #2 denotes the number of processed subproblems when the necklace rules for processing are enabled,
- #3 denotes the number of processed subproblems when in addition to the necklace rules also the (almost) simplicial vertices rules are enabled.

3.6.1 Grid graphs

For $n, m \geq 2$, the $(n \times m)$-*grid graph* (see (96)) is the simple graph with vertices v_{ij} $(1 \leq i \leq n,\ 1 \leq j \leq m)$ where v_{ij} and $v_{i'j'}$ are adjacent if $|i - i'| + |j - j'| = 1$. The $(n \times m)$-grid graph will be denoted in this section as grid n_m. Grid graphs are relatively sparse and hence the graph corresponding to the root problem has a lot of white edges. This makes it relatively difficult for the branch-and-bound algorithm to find initial chordalizations that can be used as an upper bound. The treewidth of grid n_m, known to be $\min(n, m)$, is therefore passed as an upper bound to the algorithm. Experimental results of the branch-and-bound algorithm on grid graphs can be found in Table 3.1.

graph	$\|V\|$	$\|E\|$	ub	tw	#1	#2	#3
grid 3_3	9	12	3	3	401	39	39
grid 3_4	12	17	3	3	1835	67	67
grid 3_5	15	22	3	3	6109	105	105
grid 3_6	18	27	3	3	35455	293	293
grid 3_7	21	32	3	3	171711	743	743
grid 3_8	24	37	3	3	929393	2019	2019
grid 3_9	27	42	3	3	4754479	5501	5501
grid 3_10	30	47	3	3	$> 10^7$	15001	15001
grid 4_4	16	24	4	4	$> 10^7$	638739	298325
grid 4_5	20	31	4	4	$> 10^7$	9652057	4821991

Table 3.1: Experiments of the algorithm on Grid graphs

3.6.2 Queen graphs

The n by m queen graph, denoted by queen n_m, is a graph that is based on a chess board of n by m squares. Each square corresponds to one vertex in the graph. Vertices v and w are connected by edges if a queen is allowed to move from the square corresponding to v to the square corresponding to w, i.e. one or more positions horizontally, vertically, or diagonally on the board.

While testing the branch-and-bound algorithm on the queen graphs, no initial upper bound was used. Experimental results of the algorithm on queen graphs can be found in Table 3.2. The (small instances of) queen graphs are dense and hence the graphs corresponding to the subproblems have relatively few white edges. This makes processing rules more effective, but at the same time the high treewidth of the queen graphs makes finding an optimal chordalization hard.

graph	$\|V\|$	$\|E\|$	tw	#1	#2	#3
queen 3_3	9	28	6	5	5	5
queen 3_4	12	46	8	381	59	49
queen 3_5	15	67	10	20823	561	433
queen 3_6	18	91	12	1484735	11589	9383
queen 3_7	21	118	14	$> 10^7$	472621	392947
queen 4_4	16	76	11	37909	767	551
queen 4_5	20	110	14	$> 10^7$	93487	69055
queen 4_6	24	148	16	$> 10^7$	1301913	1159711

Table 3.2: Experiments of the algorithm on Queen graphs

3.6.3 Mycielski like graphs

The algorithm was tested on Mycielski extensions of paths P_n, cycles C_n, and cliques K_n of increasing size. The Mycielski extension of a graph can be found by adding copies of all vertices, connecting them to the neighbors of their original and finally adding one extra vertex that is connected to all the copies. Results can be found in Tables 3.3, 3.4 and 3.5. The subscripts $m1$ and $m2$ in these tables mean that respectively 1 and 2 rounds of the Mycielski construction were applied to the cycles, paths or cliques. Upper bounds that were obtained from a heuristic were passed as input to the algorithm.

graph	$\|V\|$	$\|E\|$	ub	tw	# 3
C_3_m1	7	12	3	3	1
C_4_m1	9	16	4	4	23
C_5_m1	11	20	5	5	259
C_6_m1	13	24	5	5	1457
C_7_m1	15	28	5	5	11537
C_8_m1	17	32	5	5	100317
C_9_m1	19	36	5	5	854125
C_{10}_m1	21	40	5	5	7627301
C_3_m2	15	43	7	7	727
C_4_m2	19	57	8	8	17823
C_5_m2	23	71	11	10	$> 10^7$
C_6_m2	27	85	11	10	$> 10^7$

Table 3.3: Experiments of the algorithm on Mycielski-cycles

graph	$\|V\|$	$\|E\|$	ub	tw	# 3
P_3_m1	7	9	2	2	1
P_4_m1	9	13	3	3	9
P_5_m1	11	17	3	3	9
P_6_m1	13	21	3	3	9
P_7_m1	15	25	3	3	9
P_8_m1	17	29	3	3	9
P_9_m1	19	33	3	3	9
P_{10}_m1	21	37	3	3	9
P_3_m2	15	34	6	5	55
P_4_m2	19	48	7	7	640091
P_5_m2	23	62	8	7	530707
P_6_m2	27	76	8	8	$> 10^7$

Table 3.4: Experiments of the algorithm on Mycielski-paths

3.7 Conclusions

In this chapter, an exact (branch-and-bound) algorithm is described for determining the treewidth of a graph $G = (V, E)$. The algorithm constructs all chordalizations $H = (V, E \cup F)$ of G (explicitly or implicitly) by splitting the solution space on the basis of adding/forbidding edges in the chordalization. This in contrast to other exact algorithms that use a solution space based on e.g. elimination orderings to determine the treewidth.

graph	$\|V\|$	$\|E\|$	ub	tw	# 3
K_4_m1	9	22	4	4	1
K_5_m1	11	35	5	5	1
K_6_m1	13	51	6	6	1
K_7_m1	15	70	7	7	1
K_8_m1	17	92	8	8	1
K_9_m1	19	117	9	9	1
K_{10}_m1	21	145	10	10	1
K_4_m2	19	75	9	9	1831
K_5_m2	23	116	11	11	6033
K_6_m2	27	166	13	13	20553
K_7_m2	31	225	15	15	168981
K_8_m2	35	293	17	17	942903
K_9_m2	39	370	19	19	$> 10^7$
K_4_m3	39	244	18	18	$> 10^7$

Table 3.5: Experiments of the algorithm on Mycielski-cliques

The current version of the algorithm uses a selection of the ideas that are put forward in this chapter and has been tested on a number of graph classes. The results show that the algorithm in its current state can solve graph instances up to some 30 vertices, if the treewidth is not too high. The processing rules as far as they are implemented now, significantly reduce the number of subproblems that need to be processed. However, the results are not yet comparable with the branch-and-bound methods of Gogage and Dechter (64) and Bachoore and Bodlaender (10). Some additional ideas have been discussed in this chapter. Section 3.4.2 describes several lower bound procedures that can be incorporated. These might be improvements over the current, more simple lower bounding procedure. In Section 3.5.2 the idea is proposed to keep track of a conflict graph. We do have confidence in the quality of these ideas, but we realize that the proof of concept still has to be done.

Chapter 4

Local Search

In the search for tree decompositions of large graph instances, the sheer complexity of the treewidth problem forces one to resort to non-exact methods. The spectrum of such non-exact, constructive treewidth algorithms shows a trade-off between quality of the solution (width of the returned tree decomposition) and computation time that is needed. For large graphs, it often happens that neither the popular upper bound heuristics nor the approximation algorithms are able to find a good balance between these two measures. The first type may return solutions that are exponentially far from optimal and they may do so in far less time than might be available. On the other hand, the second type may take unreasonable amounts of time to find a solution that approximates an optimal solution.

In this chapter, we introduce an upper bound heuristic for treewidth that partly fills this gap. More specifically, we construct a local search heuristic for treewidth that starts with a solution obtained by one of the quick upper bound heuristics and subsequently tries to improve upon this solution by performing a neighborhood search.

The heuristic exploits a new neighborhood structure that operates directly on a tree decomposition of the input graph. This in contrast to earlier local search heuristics for treewidth that generally used derived notions such as elimination orders of the vertices, see for example (50). As a side result, we find an alternative proof for a claim from (17) that chordalizations of a graph can be made minimal with respect to fill-in edges in $O(f(e+f))$ time, where f denotes the number of fill-in edges in a chordalization.

In Section 4.1, some yet undefined notions and preliminaries will be introduced. Section 4.2 describes in detail the two main building blocks of the neighborhood structure in our local search heuristic. Section 4.3 treats the local search heuristic itself, together with a way to evaluate different solutions and some ideas for restricting the size of the neighborhood. Finally, in Section 4.4, experimental results of an implementation of the heuristic are presented.

The content of this chapter is based on cooperation with Stan van Hoesel and Arie Koster.

4.1 Preliminaries

Definition 4.1.1. *A pair of vertices* $\{u,v\} \subset V$ *is called a* fill-in pair *in a tree decomposition* (T,X) *of* $G = (V,E)$ *if* $uv \notin E$ *and* $\exists X_i \in X$ *with* $u \in X_i$ *and* $v \in X_i$*. The set of fill-in pairs of* (T,X) *will be denoted by* F*.*

Note that in Chapter 3 the letter F was used for the set of fill-in edges in a chordalization of a graph G. We like to point out here that the set F of fill-in pairs in a tree decomposition (T,X) equals the set F of fill-in edges in the chordalization of G that can be obtained from (T,X) using Algorithm 2.2. Given a tree decomposition (T,X), we denote by F_i the set of fill-in pairs that are present in exactly i bags of (T,X). Thus for a tree decomposition on l bags, the corresponding set F can be partitioned into $F_1,\ldots,F_l$. Later in this chapter we will show that elements from F_1 can simply be 'removed' from (T,X) without increasing its width. For this reason, we call $f \in F_1$ a *redundant* fill-in pair.

Definition 4.1.2. *A tree decomposition* (T,X) *is* fill-in-pair minimal *if it contains* no *redundant fill-in pairs, i.e. if* $F_1 = \emptyset$*.*

A bag X_A of a tree decomposition is included in bag X_B (or $X_A \subseteq X_B$) if all vertices of X_A are contained in X_B.

Definition 4.1.3. *A tree decomposition* (T,X) *is* inclusion minimal *if* no *bag of* (T,X) *is included in another bag of* (T,X)*.*

Note that for an inclusion minimal tree decomposition it holds that for any two neighbor bags X_A and X_B, X_A contains a vertex that is not present in X_B, and vice versa. The inclusion minimality property can easily be obtained by contracting bags to bags which they are included in.

Definition 4.1.4. *A tree decomposition* (T,X) *is* minimal *if it is both inclusion minimal and fill-in-pair minimal.*

Given a tree decomposition (T,X) of $G = (V,E)$ and $S \subset V$, by T_S we denote the subgraph of T induced by the bags that contain all vertices in S. By the properties of a tree decomposition, T_S is always a subtree of T. Indeed, if two bags X_A and X_B of (T,X) contain all vertices from S, then the bags on the unique path in T between X_A and X_B do so as well.

4.2 Neighborhood structure

In this section, we define two basic operations on a tree decomposition that we will call *merge* and *split*. The merge of two neighbor bags X_A and X_B in tree T is a contraction of

bags X_A and X_B into a new bag X_C, where X_C contains the union of the vertices of X_A and X_B. All bags (except X_A and X_B themselves) that were neighbor to either X_A or X_B before the merge are neighbor to X_C after merging X_A and X_B. Note that the validity of a tree decomposition is invariant under a merge operation: the three properties of a tree decomposition are not violated by a merge. A subtree T_S is said to be merged in (T,X) if the bags of T_S are pairwise merged until T_S is a single bag in (T,X). Though a merge operation may create redundant fill-in pairs (as we will see later), it cannot destroy inclusion minimality of a tree decomposition.

A split of a bag is in some sense the opposite operation of a merge. During a split, one bag X_C in (T,X) is replaced by two new neighboring bags X_A and X_B, such that the union of the vertex sets of X_A and X_B is equal to the vertex set of X_C and $|X_A| = |X_B| = |X_C| - 1$. When bag X_C is split into the bags X_A and X_B, there will be a vertex pair $\{u,v\}$ in X_C that is present in neither X_A nor X_B. We say that such a vertex pair is separated during the split. Assume w.l.o.g. that bag X_A contains vertex u and bag X_B contains v. Now suppose that before the split, vertex pair $\{u,v\}$ was not only present in bag X_C, but also in some bag X_D. Then in order to maintain connectivity of T_u and T_v, in the tree decomposition after the split there must be a path from X_D to X_A not containing X_B and a path from X_D to X_B not containing X_A. Because X_A and X_B are neighbors in the new tree, this would imply that T contains a cycle, a contradiction. Hence a vertex pair $\{u,v\}$ in bag X_C can only be separated by the split operation if it is exclusively present in X_C. In addition, $\{u,v\}$ can only be separated when it is a fill-in pair, i.e., when it does not correspond to an element in E, since else the resulting tree decomposition is not valid. A bag X_C can thus only be split into neighbor bags X_A and X_B when X_C contains a redundant fill-in pair, i.e. an element from F_1. A split operation thus essentially removes a redundant fill-in pair from the tree decomposition.

Definition 4.2.1. *Given a bag X_C in the tree decomposition (T,X) and $\{u,v\} \in F_1$ in X_C, we say that $\{u,v\}$ is* removed *from (T,X) when:*

1. *X_C is split into neighbor bags $X_A = X_C \setminus \{u\}$ and $X_B = X_C \setminus \{v\}$ and*
2. *The former neighbor bags of X_C that contain vertex v are reconnected to X_A and the other former neighbor bags of X_C to X_B.*

Note that in step 2 neighbor bags of X_C containing neither v nor u can either be connected to X_A or to X_B. It is not difficult to check that validity of the tree decomposition is maintained under *removal* of a redundant fill-in pair. Removing a redundant fill-in pair does not necessarily maintain the inclusion minimality property of a tree decomposition. It can be shown however that removing a redundant fill-in pair does not create any new redundant fill-in pairs. When removing a redundant fill-in pair $\{u,v\}$ by splitting bag X_C,

other redundant fill-in pairs in X_C that are incident to neither u nor v will be present both in bag X_A and X_B afterwards, i.e. they move from F_1 to F_2. For an example of operations merge and split, see Figure 4.1. This figure illustrates that a merge operation can create new redundant fill-in pairs and that a split operation can result in one bag being included in another bag.

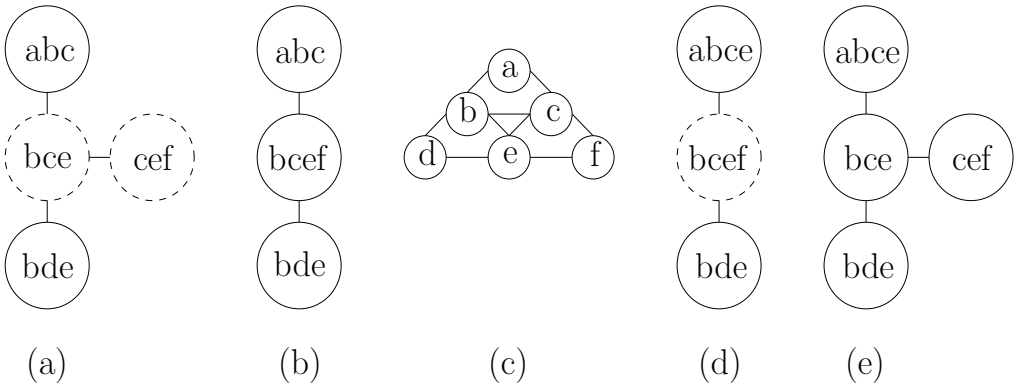


Figure 4.1: Picture (a) shows a fill-in pair minimal tree decomposition of the graph in (c). By merging the two dashed bags, pair $\{b,f\}$ becomes redundant, see (b). Picture (d) shows an inclusion minimal tree decomposition of the graph in (c). When removing redundant fill-in pair $\{b,f\}$ by splitting the dashed bag, the inclusion minimality property will be lost, see (e).

The neighborhood structure that will be employed in the local search heuristic can be described in terms of two procedures. The first one removes a fill-in pair $\{u,v\}$ from some initial minimal tree decomposition. This is done by merging $T_{\{u,v\}}$ into a single bag such that $\{u,v\}$ becomes a redundant fill-in pair and subsequently removing $\{u,v\}$ by splitting the merged bag. During the second procedure, minimality of the resulting tree decomposition is reattained. We construct an $O(f(e+f))$ algorithm that, given a tree decomposition with fill-in pair set F and width k, returns a minimal tree decomposition with fill-in pair set $F' \subseteq F$ and width $k' \leq k$. The neighborhood of a minimal tree decomposition (T,X) can now be defined by the set of all minimal tree decompositions that can be obtained from (T,X) by sequentially applying the two procedures. In the following subsections, the two procedures will be explained in more detail.

4.2.1 procedure 1: removing a fill-in pair

An important observation is that for any pair $\{u,v\} \notin E$, there exists a tree decomposition (T,X) of $G=(V,E)$ such that $\{u,v\} \notin F$. Given (T,X) with $\{u,v\} \in F$, the following algorithm shows how to remove an arbitrary fill-in pair $\{u,v\}$ from F using merge and split operations

as they are defined in the previous section.

Algorithm 4.1: remove a fill-in pair from (T,X)

Input: Tree decomposition (T,X) with $\{u,v\} \in F$
Output: Tree decomposition (T,X) with $\{u,v\} \notin F$

1. Merge $T_{\{u,v\}}$ into a single bag to make $\{u,v\}$ redundant.
2. Remove redundant fill-in pair $\{u,v\}$.

It is easy to see that inclusion minimality is maintained under the merge step in Algorithm 4.1. Also, it is not difficult to verify that a tree decomposition remains valid under application of Algorithm 4.1. Removing the redundant fill-in pair in the second step of Algorithm 4.1 is done by application of the steps in Definition 4.2.1. Note that when Algorithm 4.1 is applied to a *redundant* fill-in pair, the first step can be skipped. When Algorithm 4.1 is applied to remove fill-in pair $\{u,v\}$, we will from now on refer to the single bag that $T_{\{u,v\}}$ is merged into as bag X_M and to the bags $X_M \setminus \{u\}$ and $X_M \setminus \{v\}$ after step 2 as respectively X_{M_v} and X_{M_u}. Figure 4.2 provides an illustration of Algorithm 4.1.

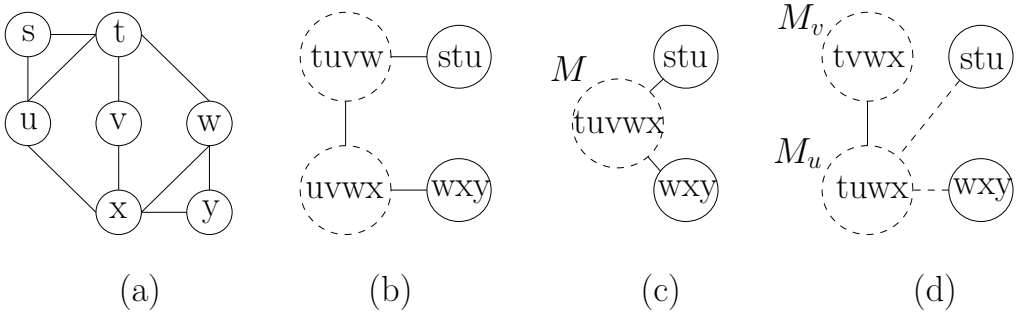


Figure 4.2: Picture (b) shows a tree decomposition of the graph in (a) with $\{u,v\} \in F_2$. In (c), subtree $T_{\{u,v\}}$ is merged into a single bag X_M and hence $\{u,v\}$ moves to F_1. Subsequently in (d), redundant fill-in pair $\{u,v\}$ is removed by splitting X_M into neighbor bags X_{M_u} and X_{M_v}. The two neighbor bags of X_M in (c) do not contain vertex v so both of them are connected to X_{M_u}.

In the second procedure that will be described in the next section, all elements from F_1 will be removed in order to regain minimality of the tree decomposition. It is therefore important to know what happens to the set F_1 during the first procedure. The following two lemmas specify for each of the two steps in Algorithm 4.1 what happens to the set of redundant fill-in pairs F_1.

Lemma 4.2.2. *Let (T,X) be a tree decomposition of a graph G with fill-in pair $\{u,v\}$. When $T_{\{u,v\}}$ is merged into a single bag X_M, no elements disappear from F_1 and only the following two types of fill-in pairs will move to F_1:*

(1) Fill-in pairs $\{i,j\}$ for which $|T_{\{i,j\}}| \geq 2$ and $T_{\{i,j\}}$ is a subtree of $T_{\{u,v\}}$.

(2) Pairs $\{i,j\}$ for which $T_{\{i,j\}} = \emptyset$ and both T_i and T_j intersect with $T_{\{u,v\}}$.

Proof. $T_{\{u,v\}}$ is merged into X_M by repeatedly merging pairs of neighbor bags in $T_{\{u,v\}}$. Any redundant fill-in pair that was present in one of the bags of $T_{\{u,v\}}$ will afterwards be present only in X_M, so it will still be redundant. Now consider a redundant fill-in pair $\{a,b\}$ from a bag not in $T_{\{u,v\}}$. Then at least one of the trees T_a or T_b does not intersect with $T_{\{u,v\}}$. Indeed if they did, then the bag $T_{\{a,b\}}$ should also intersect with $T_{\{u,v\}}$, a contradiction. Thus fill-in pair $\{a,b\}$ will not be present in X_M and hence it will remain redundant after the merge. While merging $T_{\{u,v\}}$ into X_M, all redundant fill-in pairs will thus remain redundant. Any fill-in pair that was present in at least 2 bags of $T_{\{u,v\}}$ and in no bag outside $T_{\{u,v\}}$ will only be present in X_M after the merge, so it moves from F_i (for some $i \geq 2$) to F_1. Finally, consider two vertices that are present in $T_{\{u,v\}}$, but not together in one bag. After merging $T_{\{u,v\}}$ into X_M, they form a fill-in pair that is present only in X_M, so they are redundant. □

Lemma 4.2.3. *While removing $\{u,v\}$ from (T,X) according to Definition 4.2.1, element $\{u,v\}$ drops out of F_1 and all redundant fill-in pairs that are not incident to u nor v and have the same subtree as $\{u,v\}$ will move from F_1 to F_2. All other redundant fill-in pairs remain redundant.*

Proof. Let $T_{\{u,v\}} = M$ before removing redundant fill-in pair $\{u,v\}$. First we show that while splitting X_M in X_{M_u} and X_{M_v}, no new redundant fill-in pairs are created. Each fill-in pair from X_M (except $\{u,v\}$) is present in at least one of the bags X_{M_u} or X_{M_v} after the split, so no fill-in pairs move from F_i (for some $i \geq 2$) to F_1. Furthermore, while splitting X_M, no new fill-in pairs are created, i.e. any pair in X_{M_u} or X_{M_v} was already present in bag X_M. Now consider a redundant fill-in pair in a bag other than X_M before the split. After splitting X_M, this pair won't be present in X_{M_u} or X_{M_v} either, so it remains redundant. Clearly, $\{u,v\}$ itself drops out of F_1 since it was only present in X_M and it is not present in X_{M_u} nor X_{M_v}. Finally consider the other elements of F_1 from X_M. If they are incident to u (v), then after the split they are only present in X_{M_u} (X_{M_v}), so they remain redundant. If they are not incident to u nor to v, then after the split they appear in both X_{M_u} and X_{M_v}, so they move from F_1 to F_2. □

Before proceeding with the second procedure of regaining minimality of the tree decomposition, we point out that application of the first procedure to a fill-in pair $\{u,v\}$ in a minimal tree decomposition will always lead to a tree decomposition of higher or at best

equal width. Indeed, $T_{\{u,v\}}$ consists of at least two bags. Given two bags from $T_{\{u,v\}}$, by the inclusion minimality property of (T,X) one of them contains a vertex that is not present in the other one. Thus by merging $T_{\{u,v\}}$ into a single bag X_M, bag X_M will be strictly larger than the largest bag in $T_{\{u,v\}}$. When X_M is subsequently split into bag X_{M_u} and X_{M_v} of size $|X_M|-1$, the size of these two bags is still equal to or larger than the largest bag of $T_{\{u,v\}}$.

The aim of our local search heuristic is obviously to find tree decomposition that have a lower width than some minimal starting solution. The contribution of the first procedure can thus not be found in the width of the resulting tree decomposition. However, as can be concluded from Lemmas 4.2.2 and 4.2.3, Algorithm 4.1 is capable of turning a minimal tree decomposition into a non-minimal one by creating redundant fill-in pairs in the first step. In Figure 4.2 the algorithm turns the minimal tree decomposition in (b) into the non-minimal tree decomposition in (d), where $\{u,w\},\{v,w\} \in F_1$. This property will be exploited by the second procedure, which basically removes the newly created redundant fill-in pairs, until minimality is obtained. Clearly, this can potentially decrease the width of the tree decomposition. Whenever the increase in width caused by the first procedure is smaller than the decrease in width caused by the second procedure, the overall result is positive and a better tree decomposition is detected in the neighborhood.

4.2.2 procedure 2: regaining minimality

In the previous section it was shown that tree decompositions obtained from a minimal tree decomposition by applying Algorithm 4.1 are likely to lack fill-in-pair minimality. In this section we present the procedure to regain minimality. The procedure is an algorithm called MTDA (Minimal Tree Decomposition Algorithm) that regains minimality by removing redundant fill-in pairs. It takes as input a graph G and an arbitrary tree decomposition (T,X) of G and it returns a minimal tree decomposition of lower or equal width. Moreover, it will not create any new fill-in pairs, so the set of fill-in pairs in the resulting minimal tree decomposition is a subset of the set of fill-in pairs in the input tree decomposition. The complexity of MTDA will be shown to be equal to $O(f(e+f))$, where $f = |F|$ and $e = |E|$.

inclusion minimality

Minimality of a tree decomposition is defined in terms of inclusion minimality and fill-in-pair minimality. We will assume that the input tree decomposition is already inclusion minimal. If this is not the case, we can easily obtain inclusion minimality by the following algorithm. Observe that if in a tree decomposition (T,X) of G bag $X_A \subseteq X_C$, then bag $X_A \subseteq X_B$, for all bags X_B on the unique path in T from X_A to X_C. Thus, if a bag in (T,X) is not included in any of its neighbor bags, then it will not be included in any other bag of (T,X) either.

Definition 4.2.4. *If bag X_A is included in neighbor bag X_B, we say that we contract X_A to X_B if we delete X_A from the tree decomposition and make all neighbor bags of X_A (except X_B) neighbor of X_B.*

With this in mind, we present Algorithm 4.2 to make a tree decomposition inclusion minimal.

Algorithm 4.2: inclusion minimal algorithm

Input: Tree decomposition (T, X)
Output: Inclusion minimal tree decomposition (T, X)

1. Define an ordering σ on the bags of (T, X).
2. Initialize bag X_A as the first bag in σ.
3. For each neighbor bag X_B of X_A, check whether $X_A \subseteq X_B$.
   If so, contract X_A to X_B.
4. If X_A is the last bag in σ, stop.
   Else, let X_A be the next bag in σ and goto 3.

Although we assume inclusion minimality at the start, during a run of MTDA we may still encounter situations where bags are included in neighboring bags. Consider a situation where bag X_A is included in X_B and there is a fill-in pair in F_2 that is present in bag X_A and in bag X_B. Then by contracting X_A to X_B, this fill-in pair becomes redundant, i.e. contraction of bags does not maintain fill-in pair minimality. The following lemma shows that new redundant fill-in pairs after a contraction step will only occur in one bag.

Lemma 4.2.5. *Let $X_A \subseteq X_B$ be a bag in tree decomposition (T, X). Then after contracting X_A to X_B, any new elements of F_1 are present in bag X_B.*

Proof. No new fill-in pairs are generated while contracting bag X_A to bag X_B, so $F_1 \subseteq F$. Furthermore, the only fill-in pairs for which the number of bags containing them decreases through the contraction were present in X_A and thus also in X_B. Hence, all fill-in pairs that become redundant during the contraction were present only in X_A and X_B before the contraction. After the contraction, these new redundant fill-in pairs are present only in X_B. □

The following lemma bounds the number of bags in an inclusion minimal tree decomposition from above.

Lemma 4.2.6. *Let G be a graph and let (T,X) be an inclusion minimal tree decomposition of G of width k. Then the number of bags in (T,X) is bounded from above by $n-k$.*

Proof. Since $w(T,X)=k$, there is a bag X_A in (T,X) containing $k+1$ vertices. By inclusion minimality of (T,X), the neighbor bags of X_A contain at least one vertex that is not present in X_A. Clearly, these are different vertices for all neighbors of X_A because the path connecting two such neighbors in T crosses X_A and X_A does not contain the vertices. Likewise, bags at distance 2 from X_A contain at least one vertex that is not present in any bag at distance 1 from X_A and thus neither in X_A itself. Again, these vertices are different from each other. Building on this idea, we conclude that it is possible to choose one vertex from each bag (except bag X_A) that is not present in X_A, in such a way that all the chosen vertices are pairwise different. Since X_A contains $k+1$ vertices, there are only $n-k-1$ different vertices that are not part of X_A, so there could be at most $n-k-1$ bags besides bag X_A in (T,X). □

Finally, we make the following easy observation:

Observation 4.2.7. *If $X_A \subseteq X_B$, then the width of (T,X) is invariant under contraction of bag X_A to X_B.*

fill-in-pair minimality

To obtain fill-in-pair minimality in a tree decomposition (T,X), we have to remove the redundant fill-in pairs. It is easy to check the correctness of the following observation about the width of a tree decomposition under removing a redundant fill-in pair.

Observation 4.2.8. *The width of tree decomposition (T,X) will not increase under removal of a redundant fill-in pair.*

Indeed, the only thing that happens is that one bag is replaced by two bags of strictly smaller size. By Lemma 4.2.3, removal of a redundant fill-in pair will not create new redundant fill-in pairs and even can cause other fill-in pairs to lose their redundancy. Inclusion minimality is not necessarily maintained under removal of a redundant fill-in pair. The following lemma however states that at most two bags violate this property after removing a redundant fill-in pair.

Lemma 4.2.9. *After removing a redundant fill-in pair $\{u,v\}$ from an inclusion minimal tree decomposition (T,X) by splitting $X_M = T_{\{u,v\}}$ into X_{M_u} and X_{M_v}, bags X_{M_u} and X_{M_v} are the only bags that may violate the inclusion minimal property.*

Proof. Assume that after splitting, bag X_A ($\neq X_{M_u}, X_{M_v}$) is included in some bag X_B. We show that bag X_A was already included then in another bag before the split. If $X_B = X_{M_u}$ or $X_B = X_{M_v}$, then X_A was already included in X_M. Otherwise X_A was already included

in X_B. To show that X_{M_u} (and X_{M_v}) can indeed be included in another bag after the split, let the tree decomposition before the split be such that X_M has a neighbor X_C such that $X_M \setminus X_C = \{v\}$. Then X_C contains u, so X_C will be a neighbor of X_{M_u} after the split. Since v was the only element of X_M that was not in X_C, X_{M_u} will be included in X_C. A similar argument can be used for X_{M_v}. □

minimality

We showed that the inclusion minimal property might get lost when we try to obtain fill-in pair minimality and vice versa. In this section, we show how to make a tree decomposition minimal. Algorithm MTDA that will be described next assumes as input an inclusion minimal tree decomposition and removes redundant fill-in pairs. After each such removal step, the set of redundant fill-in pairs is updated using Lemma 4.2.3 and inclusion minimality is repaired using Lemma 4.2.9. After repairing inclusion minimality, the set F_1 of redundant fill-in pairs is again updated using Lemma 4.2.5. By Observations 4.2.7 and 4.2.8, we conclude that application of MTDA to a tree decomposition will *not* increase its width.

Algorithm 4.3: MTDA: minimal tree decomposition algorithm

Input: inclusion minimal tree decomposition (T, X)
Output: minimal tree decomposition (T, X)

Determine set F_1.;
while $F_1 \neq \emptyset$ **do**
  Select $\{u, v\} \in F_1$ and let $X_M = T_{\{u,v\}}$.;
  Remove $\{u, v\}$ by splitting X_M into $X_{M_u} = X_M \setminus \{v\}$ and $X_{M_v} = X_M \setminus \{u\}$.;
  Update F_1 using Lemma 4.2.3.;
  if X_{M_u} *is included in some neighbor bag* X_A **then**
    contract X_{M_u} to X_A.;
    update F_1 using Lemma 4.2.5.;
  end
  if X_{M_v} *is included in some neighbor bag* X_A **then**
    contract X_{M_v} to X_A.;
    update F_1 using Lemma 4.2.5.;
  end
end

proof of correctness

Since no new fill-in pairs are created during either removal of a redundant fill-in pair or

contraction of a bag to a neighboring bag, the set F of fill-in pairs is strictly decreasing in size during a run of Algorithm 4.3. In each run of the while-loop, exactly one fill-in pair leaves the set F, namely the redundant fill-in pair that is removed, so after at most $|F|$ iterations of the while-loop, the set $F_1 \subseteq F$ will be empty and MTDA will stop. At the start and the end of the while-loop, the tree decomposition is inclusion minimal, so MTDA indeed returns a minimal tree decomposition. As a first step, the set of redundant fill-in pairs is determined. If F_1 is empty, the tree decomposition is minimal and MTDA stops. If F_1 is non-empty, the algorithm proceeds and selects one of the redundant fill-in pairs in F_1 that will be removed from the tree decomposition. One for example can choose an element from F_1 for which the bag containing it is maximal, since this is the bag that will be split into two smaller bags. Note that after removing the selected redundant fill-in pair, the tree decomposition is not necessarily inclusion minimal anymore. The set F_1 is updated using Lemma 4.2.3. According to Lemma 4.2.9, the only two bags that may violate inclusion minimality are X_{M_u} and X_{M_v}. If X_{M_u} is contained in one of its neighbors, it is therefore contracted to this neighbor. By performing this contraction, the set F_1 might change and we use Lemma 4.2.5 to update F_1. If X_{M_v} is included in one of its neighbors, it is also contracted to that neighbor and again Lemma 4.2.5 is used to update F_1. After this, the tree decomposition is inclusion minimal again and F_1 again contains all the redundant fill-in pairs. If F_1 is non-empty, MTDA continues and selects another element from F_1 to be removed.

Theorem 4.2.10. *Algorithm 4.3 runs in $O(f(e+f))$ time.*

Proof. As we already observed, the number of fill-in pairs will strictly decrease during a run of MTDA. At the start of MTDA, there are $f = |F|$ such fill-in pairs and only these f fill-in pairs are potential elements of F_1 during a run of MTDA. Therefore, the while-loop will be executed at most f times during a run of MTDA. It remains to show that one iteration of the while-loop takes $O(e+f)$ time. When removing a redundant fill-in pair $\{u,v\}$ by splitting bag X_M into X_{M_u} and X_{M_v}, each neighbor bag of X_M is attached to X_{M_u}, and then those containing v are reattached to X_{M_v}. Checking for the presence of vertex v in all neighbor bags of X_M can be done in $O(n)$ time. To update the set F_1 using Lemma 4.2.3, each element of F_1 that was present only in bag X_M has to be checked for incidence to both u and v, a process that takes complexity $O(f)$. Then the new bags X_{M_u} and X_{M_v} are checked for inclusion in one of their neighbors, which takes $O(n)$ time. Finally, the set F_1 is updated again by checking at most two bags for new redundant fill-in pairs. This can be done by checking for all vertex pairs in these bags that are not in E, whether they are present only in the bag under consideration, taking $O(e+f)$ time. Assuming that $n < e+f$ (which is true for any connected graph but a tree), we conclude that one iteration of the while-loop takes $O(e+f)$ time, making the time complexity of MTDA equal to $O(f(e+f))$. □

4.3 Local search

We showed how a regular fill-in pair can be removed from a minimal tree decomposition by Algorithm 4.1 and how this could introduce new redundant fill-in pairs. We also showed how Algorithm 4.3 can turn such a non-minimal tree decomposition back into a minimal one with lower or equal width. In this section, the two procedures will be combined into a local search heuristic for upper bounding the treewidth of the input graph.

4.3.1 starting solution

As a starting solution for the local search heuristic, any upper bound heuristic for treewidth can be used. We apply our local search heuristic on two different starting solutions for each considered graph instance. The first starting solution is obtained via a heuristic called *Greedy Fill-In* (GFI). GFI was first described in (97). It can be applied directly to graph G and returns a chordalization $H = (V, E \cup F)$ of G by repeatedly selecting the vertex for which the least fill-in edges have to be added when it is eliminated, then turning these neighbors into a clique by adding fill-in edges, and finally removing that vertex from consideration. The obtain a second starting solution, we employed an algorithm that is know as the *lexicographic breadth-first search* recognition algorithm, introduced in (98). Originally, this algorithm was constructed to recognize chordality of an input graph G and it terminated at the moment it recognized that no perfect elimination ordering can be found. Without the termination step however, it returns an elimination ordering on the vertices of G that can be used to construct a chordalization $H = (V, E \cup F)$ of G. We use a variant of this algorithm, known as LEX_M, that guarantees that the returned chordalization is minimal. For computational evaluations of GFI and LEX_M, we refer to respectively (36) and (83).

The chordalization that is returned by either GFI or LEX_M can be turned into an inclusion minimal tree decomposition (T, X) of the same width by means of Algorithm 2.3. Note that the set F of fill-in pairs in (T, X) equals the set F of fill-in edges in chordalization $H = (V, E \cup F)$. Tree decompositions that are constructed via LEX_M are therefore minimal. The ones that are obtained via GFI are not necessarily fill-in pair minimal, so we apply Algorithm 4.3 to them in order to obtain minimal starting solutions for our local search heuristic.

4.3.2 neighborhood

Definition 4.3.1. *Given a graph G and a minimal tree decomposition $(T, X)_0$, the* neighborhood *of $(T, X)_0$ is the set of minimal tree decompositions of G that can be obtained from $(T, X)_0$ by a single application of Algorithm 4.1 followed by a single application of Algorithm 4.3.*

Note that Algorithm 4.1 applied to a minimal tree decomposition does not necessarily return an inclusion minimal tree decomposition. To obtain the inclusion minimal tree decomposition that is requested as input for Algorithm 4.3, we apply Algorithm 4.2 once.

The neighborhood as defined here is quite large in the sense that each element of F is a candidate for removal in Algorithm 4.1. We present some ideas here for how to decrease the size of the neighborhood.

First of all, to reduce the size of the largest bag, MTDA should remove redundant fill-in pairs from the largest bag. In order to do so, there should be redundant fill-in pairs in the largest bag at the moment MTDA is applied. To achieve this, we apply Algorithm 4.1 to a fill-in pair that is present in a maximal bag of the minimal tree decomposition. This is done by merging its subtree into a single bag X_M and then splitting X_M into two bags, which are then maximal bags of the tree decomposition and moreover the only bags that possibly contain redundant fill-in pairs. Even when at the start of Algorithm 4.3 there are no redundant fill-in pairs in a maximal bag, there might still be redundant fill-in pairs in a maximal bag in a later stage of the algorithm. However these are more difficult to predict, so we apply Algorithm 4.1 only to fill-in pairs that are present in a maximal bag of the minimal tree decomposition.

Secondly, we realize that most of the minimal tree decompositions in the neighborhood of a minimal tree decomposition (T,X) will have a higher width than (T,X). Especially when $T_{\{u,v\}}$ contains a lot of vertices, application of Algorithm 4.1 to fill-in pair $\{u,v\}$ can dramatically increase the width of the tree decomposition. Most likely, the subsequent decrease in width caused by Algorithm 4.3 will not make up for this increase. Therefore it seems appropriate to apply Algorithm 4.1 only on fill-in pairs $\{u,v\}$ for which the number of different vertices in $T_{\{u,v\}}$ is reasonably small.

Combining the foregoing two considerations, we restrict the neighborhood of a minimal tree decomposition (T,X) to solutions that can be obtained from (T,X) by the following two operations in this order:

1. Single application of Algorithm 4.1 to a fill-in pair $\{u,v\}$ that is present in a maximal bag of (T,X) and for which $T_{\{u,v\}}$ contains at most $k*w$ vertices, where w is the width of (T,X) and $1<k\leq 2$ is a constant.

2. Single application of Algorithm 4.3 to restore minimality.

Finally, we mention that although different fill-in pairs may be removed in the first step, the application of MTDA could cause the resulting minimal tree decompositions to be equal. Being able to detect in advance whether two fill-in pairs being removed in the first step will lead to the same minimal tree decomposition would clearly be very helpful, since it would allow us to avoid exploring a single solution in the neighborhood more than once. In this chapter however, we did not further explore this consideration.

4.3.3 solution improvements

Clearly, we are interested in minimal tree decompositions in the (restricted) neighborhood of (T,X) that have a width strictly lower than (T,X). These are the tree decompositions for which the increase in width (caused by Algorithm 4.1) is strictly smaller than the subsequent decrease in width (caused by Algorithm 4.3). However, sometimes the local search heuristic is able to reduce the size of a maximal bag in one part of the tree decomposition, while in another part a maximal bag remains untouched. This would remain unnoticed if we only consider the width of the resulting tree decomposition. For this reason we consider solutions in the neighborhood of (T,X) that have the same width as (T,X) but a strictly smaller number of maximal bags, also as improvements over (T,X). A third measure for improvement can be stated in terms of the number of fill-in pairs. Suppose that a solution in the neighborhood of (T,X) has the same width and the same number of maximal bags as (T,X), but a larger number of fill-in pairs. Then we consider this solution to be an improvement over (T,X) as well, since a larger number of fill-in pairs basically means a larger neighborhood and thus more potential for better solutions in the neighborhood. In our heuristic, we apply a first improving strategy, i.e. as soon as a better solution is found, the current best solution will be updated and the search for further improvements is continued in the neighborhood of the new best solution.

4.4 Experimental results

In this section some experimental results of the local search heuristic are presented. Implementation was done in $C++$ and all experiments were run on an AMD Athlon 2400XP+ with 1 Gb of RAM. Graphs from Frequency Assignment networks and Second Dimacs graph coloring challenge are used as input graphs. Upper bound heuristics GFI and LEX-M are used to obtain starting solutions and we restrict the neighborhood to solutions that can be obtained by applying Algorithm 4.1 in the first step to fill-in pairs $\{u,v\}$ that are present in a maximal bag and for which the number of vertices in $T_{\{u,v\}}$ exceeds the size of this maximal bag by at most 10. To improve the chances of finding a better solution quickly, neighbors for which the number of vertices in $T_{\{u,v\}}$ are smallest are explored first. Our tables use the following terminology. N and E respectively denote the number of vertices and edges in the input graph. The columns lb and ub give the best upper and lower bounds on treewidth as reported by Koster et al. (see (83)) in their rigorous computational study on upper bound heuristics like greedy fill-in, min-fill, max-cardinality search and the minimum-separating-vertex-set heuristic and several lower bound heuristics. The columns gfi and $lex{-}m$ denote the width of the starting solution as obtained by upper bound heuristic GFI and LEX-M. Finally, the columns LS give the upper bound on treewidth that our local search heuristic

returned using these starting solutions in $time$ seconds.

4.4.1 Dimacs graph coloring

The statistics generated by our local search on the instances from the Dimacs Graph Coloring challenge can be found in Table 4.1. We were able to improve on the best known upper bound for the following graphs: games120, all of the le450-instances, queen6_6 to queen14_14, school1, zeroin.i.2 and zeroin.i.3, while for most other graphs we were able to match the best known upper bounds. For some instances (le450-5c, le450-25a, queen10_10), the portion of the gap between lower and upper bound that was closed is considerable. For the le450-graphs, upper bound heuristic LEX-M did not return an upper bound within an hour, so we decided to apply our local search heuristic for these graphs only with the starting solution provided by GFI.

4.4.2 frequency assignment

As a second set of graphs, we took the instance set of the CALMA project on frequency assignment problems. The results of our tests are displayed in Table 4.2. For graph celar03 we were able to close the gap between upper and lower bound. For all instances from graph02 to graph14, we were able to improve on the upper bound on its treewidth, often drastically. LEX-M could not find an upper bound on the treewidth of graph11 to graph14 within an hour, for which reason we only ran our local search heuristic on these graphs with a starting solution obtained by GFI.

4.4.3 Bayesian networks

Instances from the Bayesian Network Repository formed our final testing set. The results of the experiment can be found in Table 4.3.

Graph	N	E	lb	ub	gfi	LS	time	lex-m	LS	time
anna	138	493	12	12	12	12	0.17	12	12	0.31
david	87	406	12	13	13	13	0.10	13	13	0.10
fpsol2.i.1	269	11654	66	66	66	66	0.58	66	66	0.70
fpsol2.i.2	363	8691	31	31	31	31	2.05	52	31	0.71
fpsol2.i.3	363	8688	31	31	31	31	2.11	52	31	0.69
games120	120	638	24	36	40	35	141.76	37	32	561.25
homer	556	1628	26	31	31	31	99.55	36	31	45.22
huck	74	301	10	10	10	10	0.00	10	10	0.00
inithx.i.1	519	18707	56	56	56	56	27.41	223	56	1021.58
inithx.i.2	558	13979	31	31	31	31	94.17	227	31	1075.75
inithx.i.3	559	13969	31	31	31	31	93.58	227	31	1066.48
jean	77	254	9	9	9	9	0.00	9	9	0.01
le450-5a	450	5714	79	308	315	303	11929.50	-	-	-
le450-5b	450	5734	79	307	318	303	10519.11	-	-	-
le450-5c	450	9803	106	315	315	296	10076.76	-	-	-
le450-5d	450	9757	106	295	299	285	12352.52	-	-	-
le450-15a	450	8168	94	290	290	276	10314.78	-	-	-
le450-15b	450	8169	95	291	301	287	12337.49	-	-	-
le450-15c	450	16680	139	373	377	366	17365.87	-	-	-
le450-15d	450	16750	141	375	375	369	10474.78	-	-	-
le450-25a	450	8260	96	252	258	232	10348.08	-	-	-
le450-25b	450	8263	96	255	265	247	17352.31	-	-	-
le450-25c	450	17343	144	353	353	341	13817.04	-	-	-
le450-25d	450	17425	143	352	363	351	13063.89	-	-	-
miles250	125	387	9	9	9	9	0.06	10	9	0.09
miles500	128	1170	22	22	23	22	9.07	22	22	1.43
miles750	128	2113	35	36	39	36	9.44	36	36	15.63
miles1000	128	3216	49	49	50	49	7.78	49	49	6.84
miles1500	128	5198	77	77	77	77	3.58	77	77	2.83
mulsol.i.1	138	3925	50	50	50	50	0.33	66	50	1.43
mulsol.i.2	173	3885	32	32	32	32	0.17	69	32	1.82
mulsol.i.3	174	3916	32	32	32	32	0.19	69	32	1.87
mulsol.i.4	175	3946	32	32	32	32	0.19	69	32	1.82
mulsol.i.5	176	3973	31	31	32	31	0.88	69	31	1.97
myciel3	11	20	5	5	5	5	0.01	5	5	0.02
myciel4	23	71	9	10	11	10	0.26	12	10	0.32
myciel5	47	236	16	19	21	19	3.95	25	19	11.55
myciel6	95	755	29	35	35	35	36.52	55	35	246.39
myciel7	191	2360	52	66	66	66	232.89	103	78	2126.17
queen5_5	25	160	14	18	18	18	0.16	18	18	0.29
queen6_6	36	290	18	26	26	25	4.33	26	25	4.18
queen7_7	49	476	22	36	37	35	40.94	36	35	15.79
queen8_8	64	728	28	47	48	46	78.19	47	45	88.05
queen9_9	81	1056	35	59	65	58	403.95	61	60	269.99
queen10_10	100	1470	42	75	79	72	653.95	75	74	499.49
queen11_11	121	1980	48	90	95	89	1615.43	92	91	567.80
queen12_12	144	2596	55	109	117	108	2338.92	110	104	4019.00
queen13_13	169	3328	61	126	137	124	12013.44	126	126	1025.41
queen14_14	196	4186	67	149	163	146	5802.14	151	148	7340.12
queen15_15	225	5180	73	171	183	174	3399.04	172	166	5590.31
queen16_16	256	6320	79	194	218	196	11082.09	196	189	12334.27
school1	385	19095	149	221	225	213	6049.54	226	192	12131.81
zeroin.i.1	126	4100	50	50	50	50	0.20	50	50	0.21
zeroin.i.2	157	3541	32	33	33	32	2.65	43	32	5.21
zeroin.i.3	157	3540	32	33	33	32	1.89	43	32	5.14

Table 4.1: Dimacs Graph Coloring instances

Graph	N	E	lb	ub	gfi	LS	time	lex-m	LS	time
celar01	458	1449	15	15	16	15	12.72	18	15	13.08
celar02	100	311	10	10	10	10	0.04	10	10	0.04
celar03	200	721	14	15	15	14	12.18	16	14	3.45
celar04	340	1009	15	16	17	16	2.50	16	16	7.97
celar05	200	681	14	15	16	15	2.91	16	15	3.52
celar06	100	350	11	11	11	11	0.03	11	11	0.02
celar07	200	817	16	16	16	16	2.57	18	16	5.72
celar08	458	1655	16	16	16	16	9.01	20	17	3.40
celar09	340	1130	16	16	16	16	3.08	18	16	12.77
celar10	340	1130	16	16	16	16	3.11	18	16	12.66
celar11	340	975	14	15	15	15	1.10	16	15	2.32
graph01	100	358	16	24	25	24	28.26	27	23	417.46
graph02	200	709	24	50	51	41	305.03	57	43	1132.21
graph03	100	340	16	21	21	20	79.86	27	20	266.48
graph04	200	734	24	55	57	48	940.49	60	51	932.48
graph05	100	416	18	25	26	24	31.99	27	24	141.16
graph06	200	843	26	53	58	51	826.40	60	55	2429.09
graph07	200	843	26	53	58	51	826.63	60	55	2433.60
graph08	340	1234	32	91	95	84	1706.64	105	92	12077.96
graph09	458	1667	37	116	119	107	8415.20	132	121	12234.54
graph10	340	1275	31	93	97	84	12007.97	105	93	12123.45
graph11	340	1425	34	97	97	83	12079.65	-	-	-
graph12	340	1256	31	86	86	77	5235.54	-	-	-
graph13	458	1877	39	126	135	120	12130.80	-	-	-
graph14	458	1398	34	121	123	112	12041.12	-	-	-

Table 4.2: CALMA Project on Frequency Assignment instances

Graph	N	E	lb	ub	gfi	LS	time	lex-m	LS	time
alarm	37	65	4	4	4	4	0.00	4	4	0.00
barley	48	126	6	7	7	7	0.03	7	7	0.06
boblo	221	328	3	3	3	3	0.13	4	3	0.12
diabetes	413	819	4	4	4	4	13.27	35	4	128.07
fungiuk	15	36	4	4	4	4	0.00	4	4	0.00
link	724	1738	12	13	15	13	16.37	37	29	592.47
mainuk	48	198	7	7	7	7	0.02	7	7	0.07
mildew	35	80	4	4	4	4	0.03	4	4	0.03
munin1	189	366	10	11	11	11	0.91	15	11	11.08
munin2	1003	1662	6	7	7	7	22.73	16	7	178.24
munin3	1044	1745	7	7	7	7	6.23	15	8	95.07
munin4	1041	1843	8	8	8	8	10.01	28	8	76.93
munin-kgo	1066	1730	5	5	5	5	2.77	13	6	12.37
oesoca	39	67	3	3	3	3	0.00	3	3	0.00
oesoca42	42	72	3	3	3	3	0.00	3	3	0.00
oow-bas	27	54	4	4	4	4	0.02	4	4	0.02
oow-solo	40	87	5	6	6	6	0.16	6	6	0.42
oow-trad	33	72	5	6	6	6	0.04	6	6	0.08
pathfinder	109	211	6	6	6	6	0.07	7	6	0.05
pignet2	3032	7264	48	135	143	127	12391.09	-	-	-
pigs	441	806	8	10	10	9	7.02	18	10	20.94
ship-ship	50	114	6	8	8	8	0.14	9	8	0.74
vsd	38	62	4	4	4	4	0.00	4	4	0.01
water	32	123	9	9	10	9	0.19	10	10	0.06
weeduk	15	49	7	7	7	7	0.00	7	7	0.00
wilson	21	27	3	3	3	3	0.00	3	3	0.01

Table 4.3: Bayesian Network Repository instances

Chapter 5

Grid Minors

This chapter of the thesis is dedicated to treewidth in the class of planar graphs. More specifically, we examine the relation between treewidth and two other graph parameters, namely branchwidth and the side size of a largest square-grid minor in some special classes of planar graphs.

The complexity of finding the treewidth in planar graphs is not known, but it is widely believed that the problem is *NP*-hard. Approximating the treewidth of planar graphs is therefore an interesting research direction. To the best of our knowledge, the most successful approximations to the treewidth in planar graphs are based on two well known lower bounds, the branchwidth and the side size of the largest square-grid minor.

Branchwidth is a notion that is very closely related to treewidth and, just as for treewidth, it is an important algorithmic concept that is widely used in discrete mathematics and theoretical computer science; see, e.g., Bodlaender (30) and Hicks (74). Gu and Tamaki (69) constructed a $O(n^3)$ algorithm to compute optimal branch decompositions for planar graphs. For more work on planar branch decompositions, we refer to (14; 72; 73). As for the relation between treewidth (tw) and branchwidth (bw), Robertson and Seymour show in (96) that for any graph G with $bw(G) > 1$ it holds that $bw(G) \leq tw(G) + 1 \leq \lfloor \frac{3}{2} bw(G) \rfloor$. Seymour and Thomas developed an $O(n^3)$ time algorithm in (101) finding the minimum branchwidth of a given planar graph on n vertices. Therefore we can approximate treewidth efficiently in planar graphs to a factor $\frac{3}{2}$ from optimal.

Concerning the relation between treewidth and grid minors in planar graphs, independently in (67) and (106) it was shown that the treewidth of a planar graph is at most 5 times the side size of a largest square-grid minor in the graph. This upper bound on the treewidth was improved in (70) to 4.5 times the side size of a largest square-grid minor. The problem of finding the largest square-grid minor in a planar graph is interesting in itself. Although it is not known whether the problem can be solved in polynomial time, there is a $O(n^2 \log n)$ time algorithm for finding a square-grid minor with side size at least one fourth of the side

size of the largest square-grid minor, see (35).

The definition of branchwidth (see Section 5.1) straightforwardly implies that the side size of the largest square-grid minor (gm) is bounded from above by the branchwidth in any graph. In (56), a short proof can be found for the fact that graphs with high branchwidth contain large grid minors. In this chapter we further analyze relationships between treewidth, branchwidth and the side size of the largest square-grid minor in planar graphs. We present a class of planar graphs for which $tw \approx \frac{3gm}{2} - 1$ and $bw = gm$. We experienced that this seemingly trivial task is quite challenging because there are no simple techniques to accurately estimate treewidth in planar graphs. For the presented graph family, the branchwidth is easily verifiable, while arguing why the treewidth in the presented graph class is large is quite technical and requires methods from graph minor theory.

We present two different ways to determine the parameter gm in our graph class. We see the contribution of this research therefore not only in the construction of lower bounds for the treewidth approximation, but also in the proof methodologies for determining the parameter gm in a planar graph.

Furthermore we introduce two classes of planar graphs for which we conjecture that both branchwidth and treewidth are roughly equal to $2gm$. We do believe that these classes are worst cases for branchwidth and treewidth approximation in terms of gm in planar graphs.

The chapter is organized as follows. In Section 5.1, some new notions and definitions will be put forward. Subsequently in Section 5.2 we introduce a class of planar graphs for which we were able to determine bw, tw and gm. Two more classes of planar graphs are introduced in Section 5.3 for which several upper and lower bounds on bw, tw and gm are presented. Finally, we summarize the results from this chapter in Section 5.4 in which we also pose some open questions.

The content of this chapter is based on cooperation with Alexander Grigoriev and Natalya Usotskaya.

5.1 Preliminaries

For $n, m \geq 2$, the $(n \times m)$-*grid graph* (see (96)) is the simple graph with vertices v_{ij} $(1 \leq i \leq n,\ 1 \leq j \leq m)$ where v_{ij} and $v_{i'j'}$ are adjacent if $|i - i'| + |j - j'| = 1$. In this chapter, we are interested only in square grids. For simplicity of notation, we refer to the square $(n \times n)$-grid graph simply as the n-grid. In an n-grid, n is referred to as the side size of the n-grid.

We recall that H is a *minor* of a graph G if H is obtainable from a subgraph of G by edge contractions. A minor of graph G that forms a square grid graph is called a square-grid minor of G. The side size of the largest square-grid minor of G will be denoted by $gm(G)$.

We continue with definitions of branchwidth and branch decompositions. Both notions were introduced in (96).

Definition 5.1.1. *A* branch decomposition *of a graph $G = (V, E)$ is a pair (T, τ), where T is a ternary tree (every vertex has degree 1 or 3) and τ is a bijection from the set of leaves of T to $E(G)$.*

The *order* of an edge e of tree T in a branch decomposition is the number of vertices v from $V(G)$ such that there are leaves t_1, t_2 of T in different components of $T \setminus e$ for which $\tau(t_1), \tau(t_2)$ are both incident to v. The *width* of branch decomposition (T, τ) is the maximum order over all edges of T.

Definition 5.1.2. *The* branchwidth $bw(G)$ *of graph G is the minimum width over all branch decompositions of G (or 0 if $|E(G)| \leq 1$, when G has no branch decompositions).*

Two subsets $V', V'' \subseteq V$ in $G = (V, E)$ are said to *touch* each other if either they have a vertex in common or E contains an edge uv with $u \in V'$ and $v \in V''$. Given a graph $G = (V, E)$, we say that $V' \subseteq V$ is connected if $G[V']$ is connected, see Section 2.1.

Definition 5.1.3. *A set $\mathscr{B}$ of mutually touching, connected subsets of V is called a* bramble *of G. A subset of V intersecting with every element of $\mathscr{B}$ is called a* hitting set *for bramble $\mathscr{B}$.*

The *order* of a bramble $\mathscr{B}$ is the minimum size over all hitting sets for $\mathscr{B}$.

Definition 5.1.4. *The* bramble number *of a graph G is the maximum order over all brambles $\mathscr{B}$ of G.*

Brambles are a useful tool in bounding treewidth from below as can be concluded from the following theorem that is due to Seymour and Thomas (100) and its corollary.

Theorem 5.1.5. *Let k be a non-negative integer. A graph has treewidth k if and only if it has bramble number $k + 1$.*

Corollary 5.1.6. *Given graph G and a bramble $\mathscr{B}$ of order k, $tw(G) \geq k - 1$.*

Given a planar graph G together with its planar embedding, two faces f' and f'' in the embedding are said to be *adjacent* if there is a vertex in $V(G)$ incident to both f' and f''. We call a sequence $f_1, \ldots, f_n$ of faces an $f_1 f_n$-*facepath* if each of the faces in the sequence is adjacent to the previous face and to the next face in the sequence. The *length* of a facepath is equal to the number of faces in the facepath minus one.

Given an embedding of a graph G, we denote the collection of faces that are incident to vertex v by F_v. The *face distance* between faces f' and f'' in the embedding, denoted by $d_G(f', f'')$ in this thesis, is equal to the length of the shortest $f'f''$-facepath. The distance between vertex v and face f', denoted by $d_G(v, f')$, is equal to the length of a shortest $f'f''$-facepath, over all faces $f'' \in F_v$. It can be shown that for the considered planar graph families

both distances are independent of the embedding of G, since they are almost 3-connected; see (39).

For two graphs G and H, the cartesian product $G \times H$ is a graph with $V(G \times H) = V(G) \times V(H)$, $E(G \times H) = \{\{(u,v_1),(u,v_2)\} \mid u \in V(G), \{v_1,v_2\} \in E(H)\}$
$\bigcup \{\{(u_1,v),(u_2,v)\} \mid \{u_1,u_2\} \in E(G), v \in V(H)\}$. The following definition of a *cylinder graph* will be used in Section 5.3 and originates from (94).

Definition 5.1.7. *For cycle C_k and path P_h, we call $C_k \times P_h$ a $k \times h$ cylinder and we denote it by $C_{k,h}$.*

5.2 X-grids

In this section we start by introducing a family X_i of planar graphs, which we will call X-grids. Graph X_n from this family can be constructed by taking an n-grid, two $(n \times \lceil n/2 \rceil)$-grids and two $(\lceil n/2 \rceil \times n)$-grids. The four rectangular grids are connected via their long sides to the four sides of the square grid, as is illustrated in Figure 5.1.

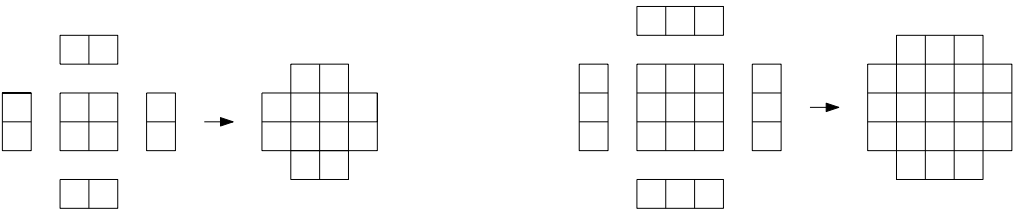


Figure 5.1: construction of X-grids X_3 and X_4

5.2.1 branchwidth of X-grids

In this section, we study the branchwidth of X-grids. Our findings are encapsulated in the following theorem.

Theorem 5.2.1. *For the X-grid X_n, $bw(X_n) = n$.*

Proof. It is well known that the branchwidth of an n-grid is equal to n. Furthermore, for any minor H of graph G, $bw(H) \leq bw(G)$. Since the n-grid is a minor of X_n we conclude from the two foregoing observations that $bw(X_n) \geq n$.

To show that $bw(X_n) \leq n$, we construct a branch decomposition of X_n of width n. To do so, we split the edge set E of X_n up in four symmetrical parts $E_1, \ldots, E_4$, as is done in Figure 5.2 for X_4. Note that for both even and odd values of n, it is easy to make a partition of E in X_n consisting of 4 symmetrical parts.

Ternary tree T of the branch-decomposition consists of one middle edge and four symmetrical subtrees T_i that contain the leaves of T corresponding to edges from set E_i, for $i = 1 \ldots 4$. The middle edge of T has order equal to n, the edges of T that are incident to a leaf have order 2 and all other edges in T have order at most n. The width of the branch decomposition for X_n is therefore equal to n.

Figure 5.3 illustrates how to construct such an optimal branch decomposition of X_4. Each edge of X_4 in Figure 5.2 is denoted by a number. In Figure 5.3, the bijection τ between the leaves of T and E is displayed by simply putting these numbers in the leaves of T. The labels on edges e of T denote the set of vertices v of X_4 for which there are leaves t_1, t_2 of T in different components of $T \setminus e$, with $\tau(t_1), \tau(t_2)$ both incident to v. Following the same idea as is illustrated in Figures 5.2 and 5.3, it is easy to construct a branch decomposition of width n for X_n for any value of $n \geq 2$. □

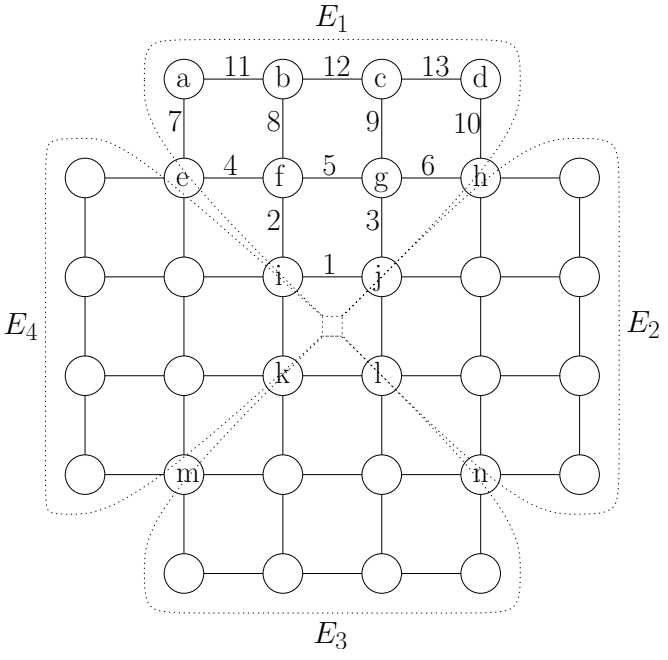


Figure 5.2: partition of E into sets $E_1, \ldots, E_4$ in planar graph X_4

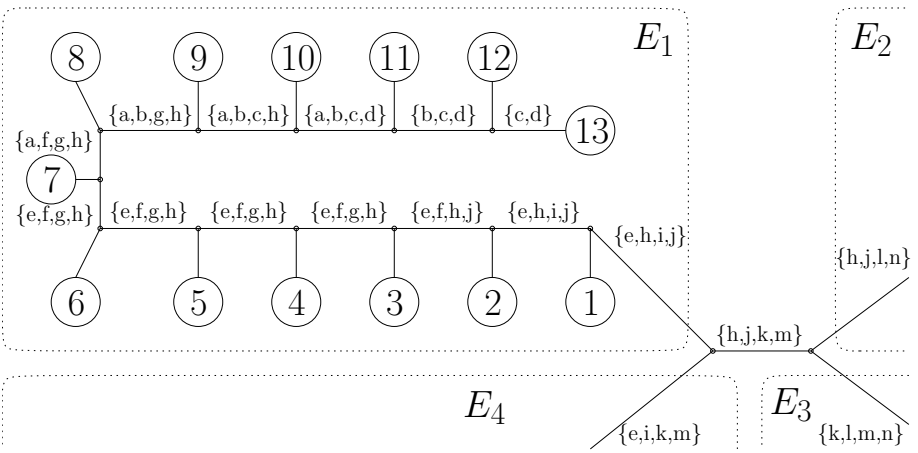


Figure 5.3: an optimal branch decomposition for the graph from Figure 5.2

5.2.2 treewidth of X-grids

In this section, we take a closer look at the treewidth of X-grids. The results of this study is condensed in the following theorem.

Theorem 5.2.2. *For X-grid X_n, $\lceil \frac{3n}{2} \rceil - 2 \leq tw(X_n) \leq \lfloor \frac{3n}{2} \rfloor - 1$.*

Proof. It is easy to construct a tree decomposition for X_n of width $\lfloor \frac{3n}{2} \rfloor - 1$. Another way to prove that $tw(X_n) \leq \lfloor \frac{3n}{2} \rfloor - 1$ is to combine the result from Theorem 5.2.1 with the fact that $tw \leq \lfloor \frac{3bw}{2} \rfloor - 1$.

To prove that $tw(X_n) \geq \lceil \frac{3n}{2} \rceil - 2$, we construct a bramble $\mathscr{B}$ of X_n of order $\lceil \frac{3n}{2} \rceil - 1$. From this bramble it follows that the bramble number of X_n is at least $\lceil \frac{3n}{2} \rceil - 1$. Then, we straightforwardly apply Theorem 5.1.5.

For an illustration of the bramble construction, we refer to Figure 5.4(a-b). We split the rows of X_n into sets R_1, R_2 and R_3 and the columns into sets C_1, C_2 and C_3. The bramble $\mathscr{B}$ now consists of all subsets that are of one of the following four types.

1. The vertices from one row and from one column intersecting this row.
2. The vertices from one column from C_1, one column from C_3 and a path between these two columns;
3. The vertices from one row in R_3, one column in C_1 and a path between this row and column;
4. The vertices from one row in R_3, one column in C_3 and a path between this row and column.

One subset of $\mathscr{B}$ of type 1 and one subset of type 2 are depicted in Figure 5.4(a) with respectively fat solid lines and fat dashed lines. Subsets of $\mathscr{B}$ of type 3 and 4 are depicted respectively by the fat solid lines and the fat dashed lines in Figure 5.4(b). It can be easily

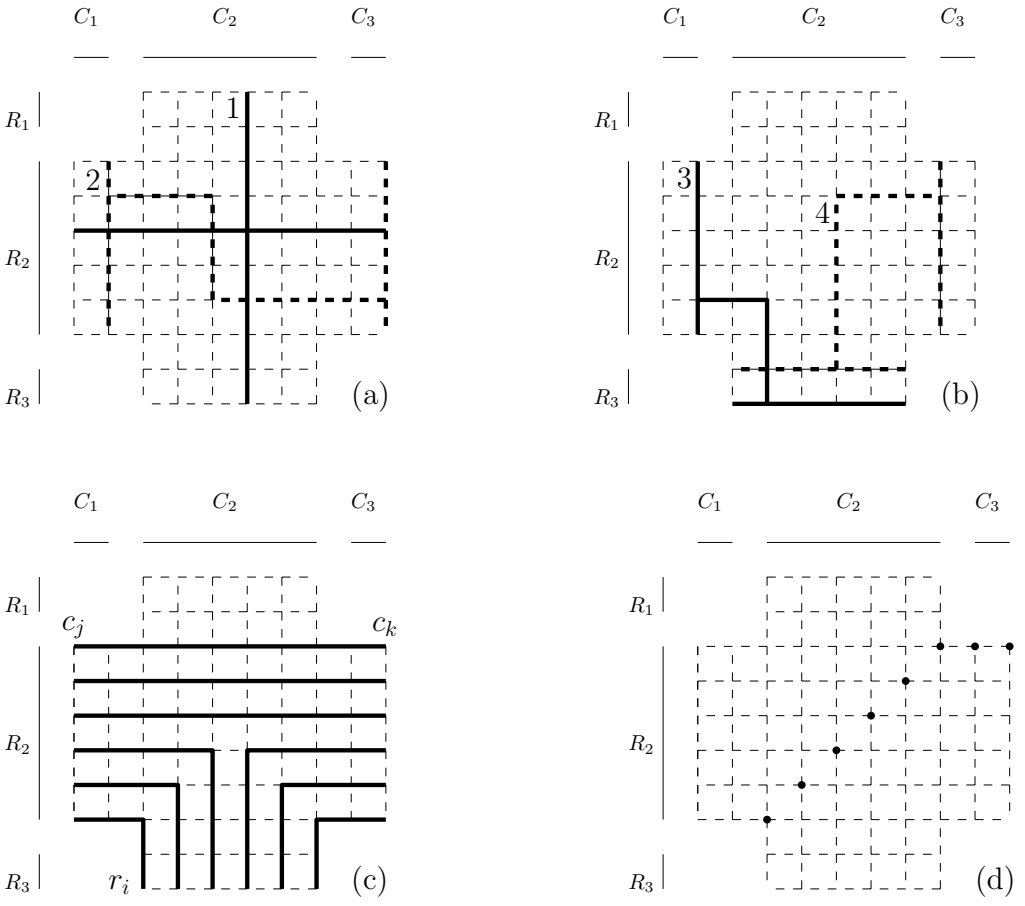


Figure 5.4: Examples of the four types of subsets of $\mathscr{B}$ in (a,b), a collection of $\lfloor \frac{3n}{2} \rfloor$ vertex disjoint paths in (c) and a hitting set for $\mathscr{B}$ of size $\lceil \frac{3n}{2} \rceil - 1$ in (d).

verified that the subsets of $\mathscr{B}$ are connected and mutually touching. We will now show that the order of $\mathscr{B}$ is equal to $\lceil \frac{3n}{2} \rceil - 1$ which will thus imply that $tw(X_n) \geq \lceil \frac{3n}{2} \rceil - 2$.

First we prove that the order of $\mathscr{B}$ is at least $\lceil \frac{3n}{2} \rceil - 1$ by showing that for every vertex set V' such that $|V'| < \lceil \frac{3n}{2} \rceil - 1$, there is a subset of $\mathscr{B}$ that is not hit by V'. Consider such a set V'. Note that R_2 and R_3 together have $\lceil \frac{3n}{2} \rceil - 1$ rows and C_1, C_2 and C_3 have more than $\lceil \frac{3n}{2} \rceil - 1$ columns together. Thus, if every row in R_2 is hit by V', then there is a row in R_3 that is not hit by V'. Similarly, if every column in C_2 is hit by V', then there is a column in

C_1 and a column in C_3 that are not hit by V'. Therefore, for any V' with $|V'| < \lceil \frac{3n}{2} \rceil - 1$, at least one of the following 4 situations occurs:

S1. Row r_i from R_2 and column c_j from C_2 are not hit by V'.

S2. Row r_i from R_2 and column c_j from C_1 are not hit by V'.

S3. Row r_i from R_3 and column c_j from C_2 are not hit by V'.

S4. Row r_i from R_3, column c_j from C_1 and column c_k from C_3 are not hit by V'.

We show that in each of these 4 situations, there is a subset of $\mathscr{B}$ that is not hit by V'. In situation S1, S2 and S3, r_i and c_j define a subset of $\mathscr{B}$ of type 1. Situation S4 needs somewhat more attention. First we point out that any vertex set in X_n that separates r_i from c_j and from c_k and simultaneously separates c_j from c_k contains at least $\lfloor \frac{3n}{2} \rfloor$ vertices. See Figure 5.4(c), where $\lfloor \frac{3n}{2} \rfloor$ vertex disjoint paths between r_i, c_j and c_k are depicted. Since V' contains strictly less than $\lceil \frac{3n}{2} \rceil - 1$ vertices and thus strictly less than $\lfloor \frac{3n}{2} \rfloor$ vertices, in situation S4 there exists thus a path P either between c_j and c_k or between r_i and c_j or between r_i and c_k such that P is not hit by V'. In the first case, $\{c_j, P, c_k\}$ forms a subset of $\mathscr{B}$ of type 2 that is not hit by V'. In the second case $\{r_i, P, c_j\}$ forms a subset of type 3 of $\mathscr{B}$ that is not hit by V' and in the last case $\{r_i, P, c_k\}$ forms a subset of $\mathscr{B}$ of type 4 that is not hit by V'. This shows that any set V' of size strictly smaller than $\lceil \frac{3n}{2} \rceil - 1$ can not be a hitting set of bramble $\mathscr{B}$. Hence the order of $\mathscr{B}$ is at least $\lceil \frac{3n}{2} \rceil - 1$.

To show that the order of $\mathscr{B}$ is equal to $\lceil \frac{3n}{2} \rceil - 1$, we note that the vertex set S as depicted in Figure 5.4(d) forms a hitting set of $\mathscr{B}$ of size $\lceil \frac{3n}{2} \rceil - 1$. It is easy to verify that all four types of subsets in $\mathscr{B}$ are indeed hit by this hitting set. □

5.2.3 largest square-grid minor of X-grids

By construction, X_n contains an n-grid minor as an induced subgraph. Intuitively, it might seem clear that the side size of the largest square-grid minor in X_n is equal to n. Next, we present two proofs that support this intuition. The first proof is an easy one and is based on results concerning branch decompositions. Without the use of branch decompositions however, finding a proof turned out to be less trivial than we thought. In our second proof, we do not resort to branch decompositions. Instead, we use arguments related to face distances in the graph.

Theorem 5.2.3. *Given X-grid X_n, $gm(X_n) = n$.*

Proof. When we combine the result of Theorem 5.2.1 with the fact that for any graph G it holds that $gm(G) \leq bw(G)$, we find that $gm(X_n) \leq n$. By construction, X_n contains an n-grid as an induced subgraph, from which we conclude that $gm(X_n) \geq n$. The two foregoing observations yield that $gm(X_n) = n$. □

Surprisingly enough, without the use of branchwidth it is rather difficult to prove Theorem 5.2.3. In the alternative proof that will be presented next, we use arguments in terms of face distances as they are defined in Section 5.1. This proof will provide a methodological technique in contrast to the branchwidth based proof.

Before we start with the alternative proof, we need several propositions. The propositions concern a mapping of the face set of a planar embedding of a graph G to the face set of a planar embedding of a minor M of G. We recall that a minor M can be obtained from G by a series of vertex deletions, edge deletions and edge contractions. Basically, if such operation does not change the number of faces in the embedding, then a face is mapped to the face in the embedding of the minor that naturally corresponds to it. If, by a vertex deletion or edge deletion, some faces are joined together to one face, then all these faces are mapped to the joined face. If, by an edge contraction, some face f disappears, then f is mapped to a face in the embedding of the minor that naturally corresponds to a neighbor face of f. We say that face f' in an embedding of minor M of G corresponds to face f in an embedding of G if f can be mapped to f'. We now introduce three propositions, for which correctness can be easily verified.

Proposition 5.2.4. *Using the mapping described above, each face in an embedding of G can be mapped to a face in an embedding of a minor M of G. Moreover, for any number i of different faces in the embedding of M, there are at least i different faces in the embedding of G that can respectively be mapped to them.*

Proposition 5.2.5. *Let f' and f'' be two faces in the embedding of G and let f'_M and f''_M be respectively the two faces corresponding to f' and f'' in the embedding of a minor M of G. Then, $d_M(f'_M, f''_M) \leq d_G(f', f'')$.*

Proposition 5.2.6. *Let f' be a face and v be a vertex in G and let M be a minor of G in which v is not deleted nor contracted to another vertex. Furthermore, let f'_M be a face in M corresponding to f'. Then, $d_M(v, f'_M) \leq d_G(v, f')$.*

Using these propositions, we are now ready to present an alternative proof of Theorem 5.2.3.

Proof. Again by construction of X_n, it is easy to see that X_n contains an n-grid as a minor. To show that $gm(X_n) = n$, we show that X_n has no $(n+1)$-grid minor. We will show that the outer face in an embedding of X_n cannot be mapped to any face in an embedding of the $(n+1)$-grid. Then by Proposition 5.2.4 we derive that the $(n+1)$-grid cannot be a minor of X_n. The proof is divided into three cases. For all three cases, we present arguments only for even values of n. For odd values of n the proof is similar. We consider the natural embedding of X_n and of the $(n+1)$-grid and from here on, we just talk about a face of X_n ($(n+1)$-grid) instead of about a face in the natural embedding of X_n ($(n+1)$-grid).

Case 1. The outer face of X_n cannot be mapped to the outer face of the $(n+1)$-grid. Because n is even, there is a vertex v in the $(n+1)$-grid that has distance $\frac{n}{2}$ to the outer face f', see the leftmost picture in Figure 5.5. If the outer face of X_n could be mapped to the outer face of the $(n+1)$-grid, then by Proposition 5.2.6 there should also be a vertex in X_n with distance at least $\frac{n}{2}$ to the outer face of X_n. However, there are no such vertices in X_n, see the rightmost picture in Figure 5.5.

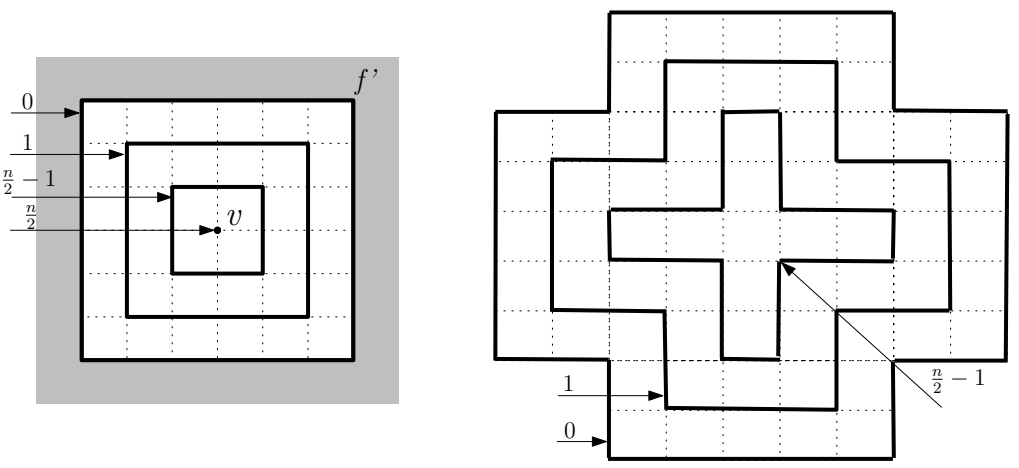


Figure 5.5: Distances of vertices to the outer face in the $(n+1)$-grid and in X_n

Case 2. We now show that the outer face of X_n cannot be mapped to any of the 4 middle faces of the $(n+1)$-grid. There are $2n$ faces in the $(n+1)$-grid having face distance $\frac{n}{2}$ to a middle face f', see Figure 5.6, leftmost picture. Suppose that the outer face of X_n can be mapped to any of the 4 middle faces of the $(n+1)$-grid. By Propositions 5.2.4 and 5.2.5 then there must be $2n$ different faces in X_n at face distance at least $\frac{n}{2}$ to the outer face. However, there are only $2n-3$ such faces in X_n, see the rightmost picture in Figure 5.6.

Case 3. Finally, let us show that the outer face of X_n cannot be mapped to any of the other inner faces of the $(n+1)$-grid. For each such inner face of the $(n+1)$-grid there is a vertex at distance $\frac{n}{2}$, see Figure 5.7. Suppose that the outer face of X_n can be mapped to one of the other inner faces of the $(n+1)$-grid. Then, by Proposition 5.2.6 there must be a vertex in X_n that has distance at least $\frac{n}{2}$ to the outer face. Again, there is no such vertex in X_n, see the rightmost picture in Figure 5.5. □

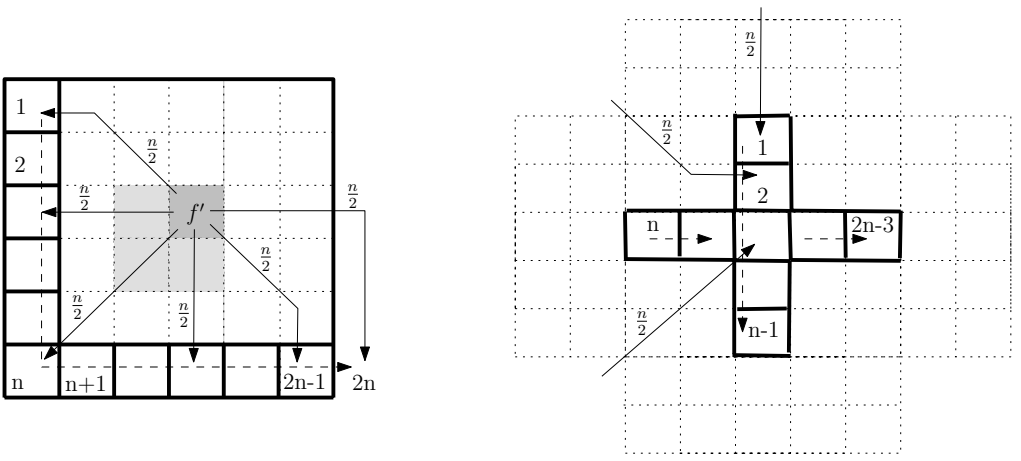


Figure 5.6: There are $2n$ faces at face distance $\frac{n}{2}$ to f' in the $(n+1)$-grid and $2n-3$ faces at face distance $\frac{n}{2}$ to the outer face in X_n

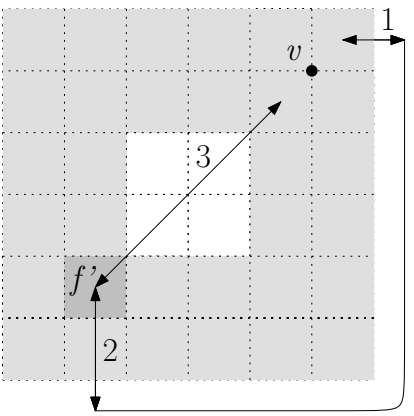


Figure 5.7: Vertex v at distance $\frac{n}{2}$ to a non-middle inner face f' in the $(n+1)$-grid

5.3 Sandwich grids and pyramids

In this section, we introduce two more classes of planar graphs, which we call *pyramids* and *sandwich grids*. The graphs in the pyramid family will be referred to as Λ_n, whereas S_n will be used to denote a sandwich grid. The pyramid Λ_n is a graph on $2n^2-2n+1$ vertices and $6n^2-10n+4$ edges. It can be constructed by building a pyramid with the side size of the base level equal to n, as shown in the leftmost picture in Figure 5.8. The sandwich-grid S_n is a graph on $2n^2$ vertices and $4n^2-4$ edges and can be constructed by taking two n-grids and

connecting the vertices on the outer face of one n-grid to the corresponding vertices on the outer face of the second n-grid, as is shown in the rightmost picture in Figure 5.8. Planar embeddings of the graphs Λ_3 and S_4 are given in Figure 5.9.

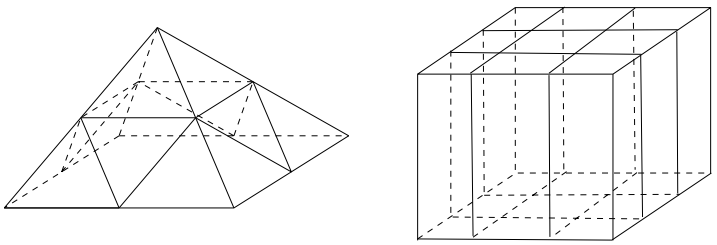


Figure 5.8: pyramid Λ_3 and sandwich grid S_4

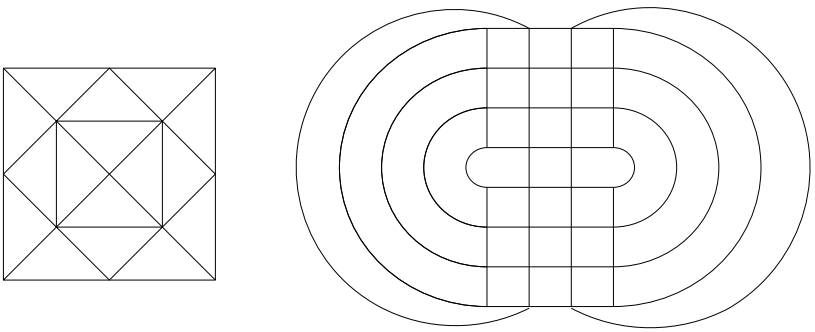


Figure 5.9: planar embeddings for pyramid Λ_3 and sandwich grid S_4

In the following we will determine the treewidth and branchwidth of sandwich grids. We will also present a lower bound on the side size of the largest square-grid minor for both families and an upper bound on the treewidth of pyramids. We conjecture that these bounds are tight.

5.3.1 branchwidth of sandwich grids

The branchwidth of sandwich grids can be roughly determined, but only indirectly, as is shown in the proof of the following theorem.

Theorem 5.3.1. *For a sandwich grid S_n, $2n \leq bw(S_n) \leq 2n+1$.*

Proof. To show that $bw(S_n) \leq 2n+1$, we use the result from the next section that $tw(S_n) \leq 2n$. When we combine this with the fact that for any graph G it holds that $bw(G) \leq tw(G)+1$, we obtain the desired result.

To show that $bw(S_n) \geq 2n$, we use a result on cylinder graphs. Gu and Tamaki show in (70) that for cylinder $C_{2n,n}$, which is a subgraph of S_n, $bw(C_{2n,n}) = 2n$. Since the branchwidth of a subgraph of S_n is smaller than or equal to $bw(S_n)$, we conclude that $bw(S_n) \geq 2n$. □

The question to determine the branchwidth of the pyramid class Λ_n still remains open. Later we will show that $tw(\Lambda_n) \leq 2n-1$. Using this result, we can bound $bw(\Lambda_n)$ from above by $2n$.

5.3.2 treewidth of sandwich grids and pyramids

The treewidth of sandwich grids can be approximated as follows.

Theorem 5.3.2. *For sandwich grid S_n, $2n-1 \leq tw(S_n) \leq 2n$.*

Proof. To prove that $tw(S_n) \leq 2n$, we show how to construct a tree decomposition (actually, a path decomposition) of S_n of width $2n$. One can start with a bag that contains $2n$ vertices from the first column of S_n (using the embedding of Figure 5.9) plus the top vertex of the second column. In the next bag, we eliminate the top vertex from the first column and introduce the second vertex from the second column, etc. The last bag contains the bottom vertex of the one-to-last column and the $2n$ vertices from the last column in S_n.

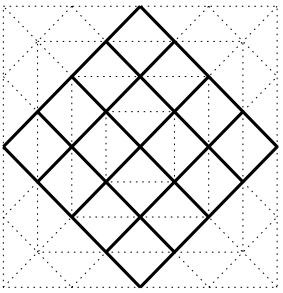
To show that $tw(S_n) \geq 2n-1$, we use a result from (70). A result from this study is that for subgraph $C_{2n,n}$ of S_n it holds that $bw(C_{2n,n}) = 2n$. Therefore $bw(S_n) \geq 2n$. Combined with the fact that $bw(S_n) \leq tw(S_n)+1$ this implies that $tw(S_n) \geq 2n-1$. □

For the pyramid Λ_n, we construct a tree decomposition showing that $tw(\Lambda_n) \leq 2n-1$. Consider one of the main diagonals in Λ_n (using the embedding from Figure 5.9) and all other paths in Λ_n that are parallel to this diagonal. Let the middle bag of the tree decomposition contain all vertices from the main diagonal. In both directions we add a bag containing the vertices from the main diagonal plus the top vertex from the next path. After that, we simply eliminate the vertices one by one. This will give a path decomposition of Λ_n of width $2n-1$, hence $tw(\Lambda_n) \leq 2n-1$. The question to prove tightness of this upper bound is still open at the moment of writing this thesis.

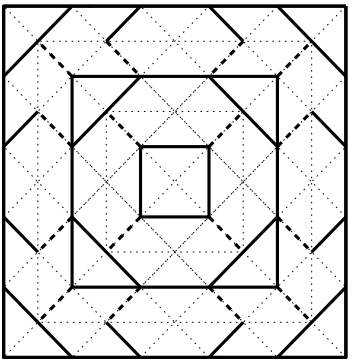
5.3.3 square-grid minor of sandwich grids and pyramids

It is easy to see that $gm(S_n) \geq n$ since S_n contains the n-grid as an induced subgraph. To show that $gm(\Lambda_n) \geq n$, we refer to Figure 5.10, which illustrates how to obtain an n-grid

minor in Λ_n for odd and even values of n. We do believe that both families S_n and Λ_n do not contain $(n+1)$-grid minors, but we are still looking for techniques to prove this.



odd n



even n

Figure 5.10: To obtain n-grid minors in Λ_n, delete dotted edges and vertices that are incident only to dotted edges and contract the dashed edges.

5.4 Summary and open questions

The results from this study are summarized in Table 5.1. As a direction for further research,

graph family	**gm**	**bw**	**tw**
X-grid (X_n)	n	n	$\approx \frac{3n}{2} - 1$
pyramid (Λ_n)	$\geq n$ $\leq 2n$	$\geq n$ $\leq 2n$	$\geq n$ $\leq 2n-1$
sandwich grid (S_n)	$\geq n$ $\leq 2n$	$\approx 2n$	$\approx 2n-1$

Table 5.1: summary of results

we recommend the interested reader to consider the following questions.

1. Can we find a technique to prove that $gm(\Lambda_n), gm(S_n) \leq n$?

2. Can we find a bramble for Λ_n of order $2n$, i.e., can we show that $tw(\Lambda_n) = 2n-1$?

3. Can we find a family of planar graphs satisfying $c_1 gm < bw < c_2 tw$, where $c_1 > 1$ and $c_2 < 1$ are some constants?

We end this chapter with the conjecture that the sandwich grid and the pyramid are worst cases for branchwidth and treewidth approximation in terms of the side size of the largest square-grid minor.

Conjecture 5.4.1. *For any planar graph G, both $bw(G)$ and $tw(G)$ are at most $2gm(G) + o(gm(G))$.*

Chapter 6

H-Subgraph Edge Deletion

In the final chapter of this thesis, we shift our focus to an application of treewidth and tree decompositions. We demonstrate how an otherwise intractable graph theoretical problem can be solved in linear time on graphs that have bounded treewidth, if a tree decomposition of the graph is available.

In the field of combinatorial graph theory, lately there has been an increased interest in algorithms for graphs not containing some specified subgraph or induced subgraph. An apparent reason for this interest is that many combinatorial problems on graphs have useful structural and/or algorithmic properties when restricted to graphs that exclude certain subgraphs. Just to give a few insightful examples: a famous result by Erdös, Kleitman and Rothschild (59) says that almost every triangle-free graph has chromatic number 2; Minty (92) and Sbihi (99) show that the maximum independent set problem is polynomially solvable on claw-free graphs; Dunbar and Frick (58) show that the path-partition conjecture is true for claw-free graphs; Borodin et al. (40) prove that planar graphs with no cycles of length from 4 to 7 are 3-colorable. There are, of course, many other structural and algorithmic results related to graphs without (a) specific subgraph(s). Motivated by this increased attention in the literature, we present algorithms in this chapter to turn a graph into a graph that excludes certain subgraphs by deleting a minimum number of edges from it.

The chapter is organized as follows. In Section 6.1 we introduce the problem that will be the subject of study in this chapter and we give some references to related studies. We continue by showing in Section 6.2 that by a general result of Courcelle, the decision version of the considered problem is theoretically solvable in linear time on graphs of bounded treewidth. We then introduce a dynamic program in Section 6.4 for the case where the input graph has bounded maximum degree that solves the problem in linear time on graphs of bounded treewidth. In Section 6.4 the subgraph that is excluded from the graph can be any fixed, connected graph. Subsequently in Section 6.5, we consider the case where the fixed subgraph forms a clique and we present a dynamic program that solves the problem

in linear time on graphs of bounded treewidth. In Section 6.6, we present Baker's style approximation schemes for the problems from Sections 6.4 and 6.5 on planar graphs. Finally, in Section 6.7, we show how our algorithms can be adopted to deal with the situation where more than one type of subgraph needs to be excluded from the input graph.

The content of this chapter is based on cooperation with Alexander Grigoriev and Natalya Usotskaya.

6.1 Problem definition

We consider the graph theoretical problem of turning a graph into a graph that excludes certain subgraphs. The transformation is executed by removing edges from the input graph. Given an input graph G and a finite set of graphs $\mathscr{H} = \{H_1, \ldots, H_t\}$, we call a subgraph of G an H-subgraph of G if it is isomorphic to one of the graphs in the set $\mathscr{H}$. In this context, a graph G is usually referred to as the *text* and $\mathscr{H}$ as the set of *patterns*. A graph G is called *H-free* if there are no H-subgraphs in G. For ease of notation, we use notions of H-subgraphs and H-free graphs instead of $\mathscr{H}$-subgraphs and $\mathscr{H}$-free graphs.

We restrict ourselves to the case in which the subgraphs that have to be excluded are fixed graphs that are connected. From here on, we will therefore assume that $\mathscr{H}$ forms a finite set of fixed, connected patterns H_i. The optimization problem that we consider in this chapter is the following.

PROBLEM:	MINIMUM H-SUBGRAPH EDGE DELETION
Input:	Text graph G and finite set of patterns $\mathscr{H} = \{H_1, \ldots, H_t\}$
Question:	What is the minimum number of edges that must be deleted from G to make it H-free?

We refer to the decision version of this problem as H-SUBGRAPH EDGE DELETION; i.e., given integer k, is it possible to make the input graph H-free by deleting at most k edges?

Related work. It is well-known that the TRIANGLE EDGE DELETION problem, a special case of H-SUBGRAPH EDGE DELETION with $\mathscr{H} = \{K_3\}$, is *NP*-complete, see Yannakakis (113). Therefore the problem H-SUBGRAPH EDGE DELETION is *NP*-complete as well. Recently, Brügmann et al. (48) proved that the problem TRIANGLE EDGE DELETION remains *NP*-complete even if the graph is planar and has maximum degree of 7. On the positive side, they construct polynomial time reduction rules to obtain linear problem kernels.

Closely related problems are considered extensively in the extremal graph theory; see, e.g., Bollobás (38) and Bohman (37). There, the general question is: Given a graph H and a number n, what is the maximum number of edges in a graph G on n vertices that does not

contain a subgraph isomorphic to H? In the context of this paper, the bounds obtained in the extremal graph theory can be seen as the source of generic lower bounds on the number of edges to be removed to make a given graph H-free.

To construct PTASs for MINIMUM H-SUBGRAPH EDGE DELETION on planar graphs, we employ the layerwise decomposition approach introduced by Baker (11). A generalization of this approach that can be applied also to problems with a non-local structure has recently been developed by Demaine and Hajiaghayi and is an extension of the bidimensionality theory. For a literature overview on bidimensionality and its connections to approximation schemes for planar graph problems, we refer to (55) and (54).

Our results. In this chapter, we first implicitly present a linear time algorithm for H-SUBGRAPH EDGE DELETION on graphs of bounded treewidth, based on a framework by Courcelle (51; 52). We then introduce constructive linear time algorithms for MINIMUM H-SUBGRAPH EDGE DELETION for the two cases for which respectively G has bounded maximum vertex degree and H is a clique. Both these algorithms are dynamic programs that assume as input a tree decomposition of the input graph. Finally, for the same cases we design PTASs for MINIMUM H-SUBGRAPH EDGE DELETION on planar graphs.

6.2 MSOL Formulations

Bounded treewidth and Monadic Second Order Logic (MSOL) have proven to be key concepts in establishing fixed-parameter tractability results. The general results of Courcelle and Arnborg et al (9) provide a host of powerful algorithmic tools for many combinatorial problems on graphs of bounded treewidth. In particular, they show that any property on graph G that can be expressed in MSOL, can be decided in linear time if G has bounded treewidth.

6.2.1 formulation for single pattern

Consider the H-SUBGRAPH EDGE DELETION problem where $\mathscr{H}$ consists of a single pattern H. By encoding this problem in MSOL, we implicitly construct a linear time algorithm solving H-SUBGRAPH EDGE DELETION on graphs of bounded treewidth. We end the section by generalizing the results to allow for a finite set $\mathscr{H}$ of patterns instead of just a single pattern.

Consider the following relational structure:

$$\mathbf{G} = \{V(G), E(G), V(H), E(H), R_G^2, R_H^2, Eq^2, f^1\},$$

where

- $R_G(x,e)$ and $R_H(x,e)$ are the vertex-edge incidence relation in G and H respectively;
- $Eq(x,y)$ is the equality predicate;
- $f : S \to V(H)$ is a function with domain $S \subseteq V(G)$ and image $V(H)$.

We now describe a formula $\Phi(k,G,H)$ such that $\mathbf{G} \models \Phi(k,G,H)$ if and only if there exists a set of edges $F \subseteq E(G)$ of cardinality at most k covering all H-subgraphs of the graph G.

Let G' be a subgraph of G. First, we define the formula $\Psi(f,G')$ to express that f is an isomorphism between G' and H. The following properties should be satisfied:

- Function f is injective:

$$Inj(f,G') = \forall\ u,v \in V(G'),\ Eq(f(u),f(v)) \Rightarrow Eq(u,v);$$

- Function f is surjective:

$$Surj(f,G') = \forall\ v \in V(H)\ \exists\ u \in V(G'),\ Eq(f(u),v);$$

- Function f preserves the edge relations from G' in H:

$$\overrightarrow{Edge}(f,G') = \forall\ u,v \in V(G'), e \in E(G'),\ R_G(u,e)\ \&\ R_G(v,e) \Rightarrow$$
$$\exists\ e' \in E(H),\ R_H(f(u),e')\ \&\ R_H(f(v),e');$$

- Function f preserves the edge relation from H in G':

$$\overleftarrow{Edge}(f,G') = \forall\ u,v \in V(H), e \in E(H),\ R_H(u,e)\ \&\ R_H(v,e) \Rightarrow$$
$$\exists\ u',v' \in V(G'),\ e' \in E(G'),\ R_G(u',e')\ \&\ R_G(v',e')\ \&\ Eq(f(u'),u)\ \&\ Eq(f(v'),v).$$

Let

$$\Psi(f,G') = Inj(f,G')\ \&\ Surj(f,G')\ \&\ \overrightarrow{Edge}(f,G')\ \&\ \overleftarrow{Edge}(f,G'),$$

and consequently

$$\Phi(k,G,H) = \exists F \subseteq E(G), |F| \leq k\ \&$$
$$(\forall V' \subseteq V(G), E' \subseteq E(G),\ \Psi(f,(V',E')) \Rightarrow |F \cap E'| \geq 1).$$

Hence, we expressed the problem H-SUBGRAPH EDGE DELETION in MSOL, which in combination with Courcelle's Theorem proves the following theorem.

Theorem 6.2.1. *Problem H-SUBGRAPH EDGE DELETION with $\mathscr{H}$ consisting of a single fixed, connected pattern H can be solved in $O(n \cdot g(w,h))$ time for some function g, where w is the treewidth of G and h is the size (number of vertices) of H.*

Using the result of Theorem 6.2.1 in combination with binary search, the corresponding optimization version of the problem can be solved in $\mathrm{O}(n\log(m) \cdot g(w,h))$ time, where m denotes the number of edges in G.

6.2.2 formulation for set of patterns

To generalize Theorem 6.2.1 to a finite set of patterns $\mathscr{H} = \{H_1, \ldots, H_t\}$, we simply need a wider relational structure $\mathbf{G}$:

$$\mathbf{G} = \{V(G), E(G), V(H_1), E(H_1), \ldots, V(H_t), E(H_t), R_G, R_1, \ldots, R_t, Eq, f_1, \ldots, f_t\},$$

where in addition to the definitions above we have to specify

- $R_i(x,e), i = 1 \ldots t$, are the vertex-edge incidence relations in H_i;
- $f_i : S \subseteq V(G) \to V(H_i), i = 1 \ldots t$, are functions with domain $S \subseteq V(G)$ and images $V(H_i)$.

Then, we can write down the general formula $\Phi(k, G, \mathscr{H})$ in the following way:

$$\Phi(k, G, \mathscr{H}) = \exists F \subseteq E(G), |F| \leq k \ \&$$

$$(\forall V' \subseteq V(G), E' \subseteq E(G), \exists 1 \leq i \leq t, \ \Psi(f_i, (V', E')) \Rightarrow |F \cap E'| \geq 1).$$

This MSOL formulation, in combination with Courcelle's Theorem, proves the following theorem.

Theorem 6.2.2. *Problem H-*SUBGRAPH EDGE DELETION *with a finite set $\mathscr{H} = \{H_1, \ldots, H_t\}$ of fixed, connected patterns can be solved in $O(n \cdot g(w,h))$ time for some function g, where w is the treewidth of G and h is the size of the largest pattern in $\mathscr{H}$.*

To solve the corresponding optimization version of the problem, one can combine Theorem 6.2.2 with binary search, which adds a factor $\log(m)$ to the complexity.

6.3 Nice tree decompositions

A tree decomposition (T, X) is called a *nice tree decomposition* if the following conditions are satisfied: Every node of the tree T has at most two children; if a node i has two children j and k, then $X_i = X_j = X_k$; and if a node i has one child j, then either $|X_i| = |X_j| + 1$ and $X_j \subset X_i$ or $|X_i| = |X_j| - 1$ and $X_i \subset X_j$. The following result is from (79):

Lemma 6.3.1. *A tree decomposition (T, X) of a graph G can be transformed without increasing its width into a nice tree decomposition of G in time polynomial in $|X|$ and the size of G. The size of the resulting nice tree decomposition is $O(w|X|)$, where w is the width of the tree decomposition.*

We point out that any tree decomposition of G can be transformed into a tree decomposition of the same width having at most n bags by simply removing all bags that are subset of another bag. Hence using Fact 6.3.1 we assume from here on that our algorithms run on nice tree decompositions with $O(wn)$ bags, rooted in some arbitrary bag X_r.

6.4 DP for text graphs with bounded degree

In this section, we consider problem MINIMUM H-SUBGRAPH EDGE DELETION in the setting where text graph G has bounded degree and bounded treewidth and $\mathscr{H}$ consists of a single fixed, connected pattern H. We construct a dynamic programming algorithm for this setting that solves MINIMUM H-SUBGRAPH EDGE DELETION in time that is exponential only in the maximum degree of G, in the width of a tree decomposition of G and in some fixed parameters of H. A generalization of this result to the setting where $\mathscr{H}$ is a finite set of fixed, connected patterns will be introduced in Section 6.7.

6.4.1 constant parameters and notation

By w, we denote the *width* of (T,X). The *maximum degree* in G will be denoted by $\Delta(G)$. The *diameter* of a graph $G=(V,E)$ is the maximum over all vertex pairs $\{u,v\}\subseteq V$ of the length of a shortest path between u and v in G. By h and d we denote respectively the *size* $|H|$ and the diameter of pattern H. Since G has bounded degree and bounded treewidth and H is a fixed connected pattern, the values $\Delta(G)$, w, h and d are all constants in the dynamic program.

Given a bag X in a nice tree decomposition (T,X), we let T_X be a subtree of T that is rooted in X and we let $G[T_X]$ be the subgraph of G that is induced by all vertices from the bags in T_X. By $\mathscr{E}_X$ we denote the set of all edges of G for which both end points are present in bag X and by E_X, we denote a subset of $\mathscr{E}_X$, i.e., $E_X\in 2^{\mathscr{E}_X}$. For a bag X, by V_X we denote the set of vertices of G that are in bag X. For a subtree T_X, by V_{T_X} we denote the set of vertices of G that are present in some bag of T_X. Finally for a vertex set S, by $\mathscr{H}_S$ we denote the set of different H-subgraphs in G that are incident to some vertex in S and by H_S, we denote a subset of $\mathscr{H}_S$.

Lemma 6.4.1. *Consider a graph G and a tree decomposition (T,X) of G of width w. Let $Q=h!\binom{\Delta(G)^{d+1}}{h}$, then*

- *for vertex v, $|\mathscr{H}_{\{v\}}|$ is bounded from above by Q, and*
- *for bag X in (T,X), the value $|\mathscr{H}_{V_X}|$ is bounded from above by $(w+1)Q$.*

Proof. Given that v corresponds to some vertex of an H-subgraph in G it is clear that all h vertices in this H-subgraph have distance at most d to v in this H-subgraph and thus also in G. Since the maximum degree in G is $\Delta(G)>1$, there are at most $\sum_{i=0}^{d}\Delta(G)^i=\frac{\Delta(G)^{d+1}-1}{\Delta(G)-1}\leq \Delta(G)^{d+1}$ vertices in G that have distance at most d to v. Since $\binom{\Delta(G)^{d+1}}{h}$ different sets of h vertices can be chosen among $\Delta(G)^{d+1}$ vertices and each such set can contain at most $h!$ different H-subgraphs, the result follows. The second statement is a straightforward corollary of the first one, since $|X|\leq w+1$. □

6.4.2 dynamic program and results

First we note that by deleting a set of edges from G that covers all H-subgraphs of G, G becomes H-free. Given a subtree T_X of T, we define:

- $F(E_X, T_X, H_{V_X})$ is the minimum cardinality of a set S of edges from $G[T_X]$ covering all H-subgraphs from $(\mathscr{H}_{V_{T_X}} \setminus \mathscr{H}_{V_X}) \cup H_{V_X}$ in G, given that $S \cap \mathscr{E}_X = E_X$

- $F(E_X, T_X, H_{V_X}) = \infty$ if H-subgraphs from $(\mathscr{H}_{V_{T_X}} \setminus \mathscr{H}_{V_X}) \cup H_{V_X}$ can *not* be covered by E_X plus some set of edges from $E(G[T_X]) \setminus \mathscr{E}_X$.

For each bag X in (T, X), we compute a table of such $F-$values for subtrees rooted in X, one value for each combination of a subset from $\mathscr{E}_X$ and a subset from $\mathscr{H}_{V_X}$. One such table thus contains $2^{|\mathscr{E}_X|+|\mathscr{H}_{V_X}|}$ values. To be more specific, we compute tables for the following three types of subtrees:

1. For each bag X in T, except for root bag X_r, we compute the table for the subtree consisting of the parent X^+ of X, X and all descendants of X in T. The root bag of this subtree is X^+. Such subtree will be denoted by T_{X^+}.

2. For each bag X with two children we compute a table for the subtree T_X, consisting of bag X and all descendants of X.

3. For each leaf bag X we compute a table for the subtree T_X.

Note that for a leaf bag X, subtree T_X equals bag X. Therefore the tables for the subtrees of type 3 are easy to compute.

Lemma 6.4.2. *A value in the table of the subtree T_X rooted in a leaf bag can be computed in constant time.*

Proof. To compute one value $F(E_X, T_X, H_{V_X})$ for the table of the subtree T_X that is rooted in leaf bag X, one just needs to check whether edge set E_X covers all H-subgraphs of H_{V_X}. If this is the case, then $F(E_X, T_X, H_{V_X}) = |E_X|$, else $F(E_X, T_X, H_{V_X}) = \infty$. Using the observation that $|E_X| \leq w^2$, Lemma 6.4.1 and the fact that both w and Q are constants, the result follows. □

The following lemmas give recursive formulas that show how to compute the tables for subtrees that are rooted in a non-leaf bag of tree decomposition (T, X). First we show how to update the table when we combine two subtrees that are rooted in the same bag and have only this bag in common.

Lemma 6.4.3. *Let T'_X and T''_X be two subtrees rooted in X that only share bag X. Let T_X be the subtree that is obtained by combining T'_X and T''_X and let $H_{V_X}, I_{V_X}, J_{V_X} \in 2^{\mathcal{H}_{V_X}}$. Then*

$$F(E_X, T_X, H_{V_X}) = \min_{I_{V_X}, J_{V_X} : H_{V_X} \subseteq I_{V_X} \cup J_{V_X}} F(E_X, T'_X, I_{V_X}) + F(E_X, T''_X, J_{V_X}) - |E_X|.$$

The next two lemmas show how to update the values for a subtree when we extend it by the parent bag of its root.

Lemma 6.4.4. *Let T_Y be a subtree rooted in Y and let $X = Y \cup \{v\}$ be the parent bag of Y. Furthermore let $T_X = T_{Y^+}$ and let $E_Y = E_X \cap \mathcal{E}_Y$. Then:*

$$F(E_X, T_X, H_{V_X}) = \min_{H_{V_Y} :\, E_X \setminus E_Y \text{ covers } H_{V_X} \setminus H_{V_Y} \text{ in } G} F(E_Y, T_Y, H_{V_Y}) + |E_X \setminus E_Y|$$

and $F(E_X, T_X, H_{V_X}) = \infty$ if there is no such H_{V_Y}.

Lemma 6.4.5. *Let T_Y be a subtree rooted in Y and let $X = Y \setminus \{v\}$ be the parent bag of Y. Furthermore, let $T_X = T_{Y^+}$. Then:*

$$F(E_X, T_X, H_{V_X}) = \min_{E_Y, H_{V_Y} :\, H_{V_X} \cup (\mathcal{H}_{\{v\}} \setminus \mathcal{H}_{V_X}) \subseteq H_{V_Y},\ E_Y \cap \mathcal{E}_X = E_X} F(E_Y, T_Y, H_{V_Y}).$$

Obviously, when bag Y and parent bag X of Y have the same vertex set, then the update of the table after an extension by the parent bag can be accomplished by simply copying the values. The following lemma gives an upper bound on the time to compute one F-value.

Lemma 6.4.6. *The time needed to determine an F-value by one of the Lemmas 6.4.3, 6.4.4 or 6.4.5 is bounded from above by $O(2^{w^2+2(w+1)Q} w^4 Q^2)$.*

Proof. To determine a value using Lemma 6.4.3 takes $O(4^{|\mathcal{H}_{V_X}|} |\mathcal{H}_{V_X}|^2)$ time. Using Lemma 6.4.1, this time is bounded from above by $O(4^{(w+1)Q} w^2 Q^2)$. To determine a value using Lemma 6.4.4, for all subsets H_{V_Y} we have to check whether all elements from $H_{V_X} \setminus H_{V_Y}$ contain an edge from $E_X \setminus E_Y$. This takes less than $2^{|\mathcal{H}_{V_Y}|} |H_{V_X}| |E_X|$ time and by using Lemma 6.4.1 and the observation that $|E_X| \leq w^2$, this is bounded from above by $O(2^{(w+1)Q} w^3 Q)$. To determine a value using Lemma 6.4.5, for all combinations of E_Y and H_{V_Y} we have to check whether $H_{V_X} \subseteq H_{V_Y}$, whether $\mathcal{H}_{\{v\}} \setminus \mathcal{H}_{V_X} \subseteq H_{V_Y}$ and whether $E_Y \cap \mathcal{E}_X = E_X$. This takes less than $2^{|\mathcal{E}_Y| + |\mathcal{H}_{V_Y}|} (|H_{V_X}| + |\mathcal{H}_{V_X}|^2 + |E_X|^2)$ time and as shown before this can be bounded by $O(2^{w^2+(w+1)Q} w^4 Q^2)$. Clearly, all these running times are smaller than $O(2^{w^2+2(w+1)Q} w^4 Q^2)$. □

Using Lemmas 6.4.2, 6.4.3, 6.4.4 and 6.4.5, we can compute all necessary tables. For all bags in the tree, in post order, we construct the tables for the subtree(s) rooted in this bag. The proof of Lemma 6.4.2 shows how to construct the table for a subtree rooted in leaf bag X. If X is not a leaf bag, suppose X has 1 child Y, then we may assume that earlier

already we computed the table for subtree T_Y. By using Lemma 6.4.4 or 6.4.5 we then can compute the table for subtree $T_X = T_{Y^+}$ from the table of T_Y. If X is not a leaf bag and it has 2 children X and Y, then by definition of a nice tree decomposition $X = Y = Z$ and we may assume that we already computed the tables for subtrees T_Y and T_Z. By simply copying the values we can construct the tables for subtrees T_{Y^+} and T_{Z^+} from the tables of T_Y respectively T_Z. Then we use Lemma 6.4.3 to compute the table for subtree T_X from the tables for subtrees T_{Y^+} and T_{Z^+}. The answer to MINIMUM H-SUBGRAPH EDGE DELETION can be found in the table of T_{X_r}. To be more precise, the solution is

$$\min_{E_{X_r}} F(E_{X_r}, T_{X_r}, \mathscr{H}_{V_{X_r}}).$$

Using the optimal set E_{X_r}, one can perform a backward search in the tree to determine a minimum set of edges that must be deleted from G to make it H-free. To estimate the running time of the dynamic program, in the following lemma we determine an upper bound on the number of individual F-values that must be computed:

Lemma 6.4.7. *In the dynamic program, at most $O(n2^{w^2+(w+1)Q}w)$ F-values need to be determined.*

Proof. A bag in the rooted tree T has at most 2 children. Thus for any bag X, we compute a table for at most 3 subtrees rooted in bag X, namely T_X and T_{Y^+}, T_{Z^+} for possible children Y, Z of X. We recall that the number of bags in T is bounded by $O(wn)$. Therefore the total number of tables to construct is bounded by $O(wn)$. Since the width of (T,X) is w, there are at most $w+1$ vertices in bag X and therefore $|\mathscr{E}_X|$ is bounded from above by $\frac{w^2+w}{2} \leq w^2$ for $w \geq 1$. Furthermore, by Lemma 6.4.1, $|\mathscr{H}_{V_X}|$ is bounded from above by $(w+1)Q$. The latter two observations imply that the number of values in one table is bounded from above by $2^{w^2+(w+1)Q}$, from which the result follows. □

The main result of this section is described in the following theorem:

Theorem 6.4.8. *Given a graph G of bounded maximum degree $\Delta(G)$, a fixed connected pattern H and a nice tree decomposition (T,X) of G of bounded width w. The problem* MINIMUM H-SUBGRAPH EDGE DELETION *on G with $\mathscr{H} = \{H\}$ can be solved in time $O(n2^{2w^2+3(w+1)Q}w^5Q^2)$, where $Q = h!\binom{\Delta(G)^{d+1}}{h}$.*

Proof. By Lemma 6.4.7, we need to compute at most $O(n2^{w^2+(w+1)Q}w)$ F-values in the dynamic program that solves the problem and by Lemma 6.4.6, it takes at most $O(2^{w^2+2(w+1)Q}w^4Q^2)$ time to compute one such value. Combining these results, we conclude that the dynamic program runs in $O(n2^{2w^2}w^5\binom{w}{h-1}h^2)$ time. □

6.5 Dynamic program for clique patterns

In this section, we consider MINIMUM H-SUBGRAPH EDGE DELETION for the special case where H is a clique and G has bounded treewidth. Compared to Section 6.5 we thus drop the constraint that G should have bounded vertex degree. We exploit the fact that every tree decomposition contains a bag with all vertices from the clique (see Lemma 2.4.2) and we find a polynomial time algorithm that again acts on a nice tree decomposition of the text graph G. It is important to state here that in this section we consider H-subgraphs of G that are induced by the vertex set of $G[T_X]$, not by the vertex set of G.

6.5.1 dynamic program and results

For subtree T_X of T rooted in bag X, we define:

- $F(E_X, T_X)$ is the minimum cardinality of a set S of edges from $G[T_X]$ that covers all H-subgraphs of $G[T_X]$, given that $S \cap \mathscr{E}_X = E_X$.
- $F(E_X, T_X) = \infty$, if **not** all H-subgraphs of $G[T_X]$ can be covered by E_X plus some set of edges from $E(G[T_X]) \setminus \mathscr{E}_X$.

We compute tables for the same subtrees as in the previous section. Note that one table for a subtree rooted in bag X now consists of $2^{|\mathscr{E}_X|}$ F-values. Using similar arguments as those used in the previous section, it is easy to prove the following:

Lemma 6.5.1. *During a run of the dynamic program, at most $O(n2^{w^2}w)$ individual F-values have to be determined.*

As in the previous section, a value for the table of a subtree that is rooted in a leaf bag can be computed in constant time. The following lemmas give recursive formulas that show how to compute the tables for subtrees that are rooted in a non-leaf bag of T. The first lemma shows how we can combine subtrees that are rooted in the same bag and have only this bag in common.

Lemma 6.5.2. *Let T_X be obtained by taking the union of subtrees T'_X and T''_X such that the root X of T'_X and T''_X is the only bag that belongs to both subtrees. Then*

$$F(E_X, T_X) \;=\; F(E_X, T'_X) + F(E_X, T''_X) - |E_X|.$$

The next two lemmas show how to update the values for a subtree when we extend it by the parent bag of its root.

Lemma 6.5.3. *Let T_Y be a subtree of T rooted in bag Y, let $X = Y \cup \{v\}$ be the parent bag of Y in T. Furthermore let $T_X = T_{Y^+}$ and let $E_Y = E_X \cap \mathscr{E}_Y$. Then:*

$$F(E_X, T_X) = \begin{cases} F(E_Y, T_Y) + |E_X \setminus E_Y|, & \text{if } E_X \text{ covers all } H\text{-subgraphs of} \\ & G[T_X] \text{ that are incident to vertex } v. \\ \infty, & \text{otherwise.} \end{cases}$$

Indeed, since X is the only bag containing v in T_X, it also contains all neighbors of v in $G[T_X]$. Therefore, all edges of an H-subgraph of $G[T_X]$ that are incident to v are part of $\mathscr{E}_X$ and thus if such an H-subgraph is not covered by E_X then it is not covered at all.

Lemma 6.5.4. *Let T_Y be a subtree of T rooted in bag Y, let $X = Y \setminus \{v\}$ be the parent bag of Y in T and let $T_X = T_{Y^+}$. Then*

$$F(E_X, T_X) = \min_{E_Y \,:\, E_Y \cap \mathscr{E}_X = E_X} F(E_Y, T_Y).$$

Again, when bag Y and parent bag X of Y have the same vertex set, then the update of the table after an extension by the parent bag can be accomplished by simply copying the values. In the previous section, it is explained how Lemmas 6.5.2, 6.5.3 and 6.5.4 can be used to determine the tables for all necessary subtrees. The minimum value in the table of subtree T_{X_r} is the solution to MINIMUM H-SUBGRAPH EDGE DELETION.

Lemma 6.5.5. *The time needed to determine an F-value in the algorithm is bounded from above by $O(2^{w^2} w^4 \binom{w}{h-1} h^2)$.*

Proof. Determining the value for a subtree rooted in a leaf bag or by using Lemma 6.5.2 takes constant time. When using Lemma 6.5.3, we check whether E_X covers all H-subgraphs of $G[T_X]$ that are incident to v. Vertex v has at most w neighbors in $G[T_X]$, so we have to check for each of the at most $\binom{w}{h-1}$ different combinations of $(h-1)$ such neighbors whether they form an h-clique with v in $G[T_X]$ for which all $\frac{h^2-h}{2}$ edges are in $\mathscr{E}_X \setminus E_X$. Since $|\mathscr{E}_X|$ is bounded by $O(w^2)$, this takes at most $O(w^2 \binom{w}{h-1} h^2)$ time. When using Lemma 6.5.4 to determine an F-value, for all E_Y we have to check whether $E_Y \cap \mathscr{E}_X = E_X$, which can be done in time $O(2^{w^2} w^4)$. Clearly, all these algorithmic time complexities are bounded from above by $O(2^{w^2} w^4 \binom{w}{h-1} h^2)$. □

The main result of this section is described in the following theorem:

Theorem 6.5.6. *Given an arbitrary text graph G, clique pattern H of size h and a nice tree decomposition (T, X) of G of bounded width w with N bags. Then* MINIMUM H-SUBGRAPH EDGE DELETION *on graph G with $\mathscr{H} = \{H\}$ can be solved in $O(n 2^{2w^2} w^5 \binom{w}{h-1} h^2)$ time.*

Proof. By Lemma 6.5.1, we need to compute at most $O(n 2^{w^2} w)$ F-values in the dynamic program that solves the problem and by Lemma 6.5.5, it takes at most $O(2^{w^2} w^4 \binom{w}{h-1} h^2)$ time to compute one such value. Combining these results, we conclude that the dynamic program runs in $O(n 2^{2w^2} w^5 \binom{w}{h-1} h^2)$ time. □

6.6 Baker's approximation scheme

In this section, we consider the problem MINIMUM H-SUBGRAPH EDGE DELETION for planar text graphs and patterns. We combine the dynamic programs from Sections 6.4 and 6.4 with a technique invented by Brenda Baker to construct PTASs for the two cases where respectively G has bounded vertex degree and where H is a 3-clique or 4-clique.

6.6.1 bounded outerplanarity index

The following two lemmas form the analogues to respectively Theorem 6.4.8 and Theorem 6.5.6. The only difference is that they assume planar input graphs with bounded outerplanarity index instead of regular input graphs with bounded treewidth.

Lemma 6.6.1. *Given a planar text graph G of bounded maximum degree $\Delta(G)$ and bounded outerplanarity index l and a fixed connected pattern H, an optimal solution to* MINIMUM H-SUBGRAPH EDGE DELETION *can be obtained in time $O(n2^{18l^2+9lQ}l^5Q^2)$, where $Q = h!\binom{\Delta(G)^{d+1}}{h}$.*

Proof. By Theorem 2.2.2, the outerplanarity index l of G can be determined in $O(n^2)$ time. By Theorem 2.2.3, $tw(G) \leq 3l-1$ and by Theorem 2.4.4, a tree decomposition of G of width $w \leq 3l-1$ can be obtained in linear time that can be turned into a nice tree decomposition in linear time. By Theorem 6.4.8 we can use this nice tree decomposition to solve MINIMUM H-SUBGRAPH EDGE DELETION on G in time $O(n2^{2w^2+3(w+1)Q}w^5Q^2)$. Since $w \leq 3l-1$, the result follows. □

A similar result can be obtained for MINIMUM H-SUBGRAPH EDGE DELETION on planar graphs when H is a 3-clique or 4-clique. Note that to cover all K_2-subgraphs of a graph by edges, one simply has to select as an Edge Cover the complete set of edges. Moreover, note that a planar graph G does not have K_k as a subgraph for $k \geq 5$. This is why we only consider K_3 and K_4 as subgraphs in this section that deals with planar graphs.

Lemma 6.6.2. *Given a planar text graph G of bounded outerplanarity index l and pattern H that is either a K_3 or K_4, an optimal solution to* MINIMUM H-SUBGRAPH EDGE DELETION *on G can be obtained in time $O(n2^{18l^2}l^5\binom{3l-1}{h-1}h^2)$.*

Proof. Similar as the proof of Lemma 6.6.1. By Theorem 6.5.6, MINIMUM H-SUBGRAPH EDGE DELETION on G can be solved in $O(n2^{2w^2}w^5\binom{w}{h-1}h^2)$ time. Since $w \leq 3l-1$, the result follows. □

The problem MINIMUM H-SUBGRAPH EDGE DELETION on planar graphs of bounded degree with fixed, connected pattern is thus fixed parameter tractable, since it is tractable

when parameterized by outerplanarity index l of G. The same holds for MINIMUM H-SUBGRAPH EDGE DELETION on planar graphs with clique pattern. We will use these properties to construct a Baker's approximation scheme in the next section that can be applied to both problems.

6.6.2 approximation schemes

Given a planar embedding of a planar graph $G = (V, E)$, we say that a vertex v is of *level* 1 if it is on the exterior face of the embedding. Let V_i be the set of all vertices of level i or lower than i, then vertex w is in level $i+1$ if it is on the exterior face of the embedding induced by $G[V \setminus V_i]$. We say that edge $e = (v,w) \in E$ is in level i of the embedding if both vertices v and w are in level i. We assume in this section that a planar embedding is represented by an appropriate data structure such that levels of vertices can be computed in linear time.

In the following two theorems and their proof, we denote by E_{opt} a minimum set of edges from G covering all H-subgraphs of G and by OPT we denote the size of E_{opt}. First we present an approximation scheme for MINIMUM H-SUBGRAPH EDGE DELETION on a planar graph G with bounded degree and arbitrary fixed, connected pattern H. A generalization of the results to finite sets of fixed, connected patterns will be introduced in Section 6.7.

Theorem 6.6.3. *For a planar text graph G of bounded degree, a fixed connected pattern H and any $s > 0$, there is a $O(n^2 2^{18l^2+9lQ} l^5 Q^2)$-time algorithm for* MINIMUM H-SUBGRAPH EDGE DELETION *that finds a solution of size at most $(\frac{s+1}{s})OPT$, where l is a constant depending on s and the outerplanarity index of H.*

Proof. First, we use Theorem 2.2.2 to determine G's outerplanarity index k and a k-outerplanar embedding of G in $O(n^2)$ time. I.e, each vertex of G belongs to one of the k levels. Similarly, we determine H's outerplanarity index k' in $O(h^2)$ time. Since H is a fixed subgraph, k' is a constant. If k was a constant, the problem could be solved in polynomial time by complete enumeration. Therefore it is reasonable to assume that $k > (s+2)k'$. Now for some fixed $2k' \leq l \leq k$ and for each $i \in I = \{mk'+1 \mid 0 \leq m \leq \lfloor \frac{l}{k'} - 2 \rfloor\}$ we construct a set G^i of induced subgraphs of G, consisting of the l-outerplanar subgraphs of G:

- induced by levels 1 to $i + k' - 1$.
- induced by levels $j(l-k') + i$ to $j(l-k') + i + l - 1$, $0 \leq j \leq \lfloor \frac{k-i-l+1}{l-k'} \rfloor$.
- induced by levels $\lfloor \frac{k-i-l+1}{l-k'} \rfloor (l-k') + i + l - k'$ to k.

See Figure 6.1 for an illustration. Note that $i + k' - 1 \leq \lfloor \frac{l}{k'} - 2 \rfloor k' + 1 + k' - 1 \leq l - k' < l$

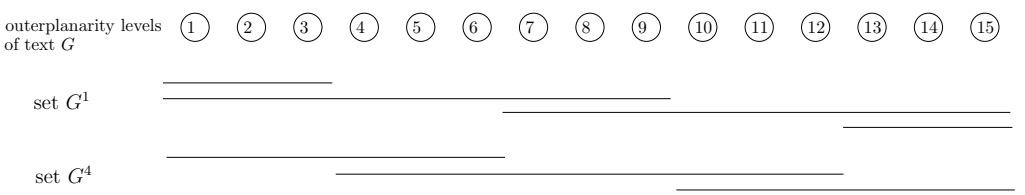


Figure 6.1: Suppose text G has 15 outerplanarity levels and pattern H is 3-outerplanar. For $l = 9$, we construct two sets (G^1 and G^4) of 9-outerplanar subgraphs of G. For example, the first graph in G^1 is induced by all vertices in G's first three levels.

and also that $k-(\lfloor\frac{k-i-l+1}{l-k'}\rfloor(l-k')+i+l-k')+1 \leq k-(k-i-2l+1+k'+i+l-k')+1 = l$, so both the subgraph induced by the first bullet and the one induced by the third bullet are l-outerplanar. Clearly, also the subgraphs under the second bullet are l-outerplanar. Since $|I| = \lfloor\frac{l}{k'}-1\rfloor$ and for each i we construct $\lfloor\frac{k-i-l+1}{l-k'}+3\rfloor$ subgraphs, in total we construct less than $(\frac{l-k'}{k'})(\frac{k+2l+1}{l-k'}) = \frac{k+2l+1}{k'} \leq 3k \leq 3n = O(n)$ induced subgraphs of G for fixed value of l.

Observation 6.6.4. *For every $i \in I$, the vertices in the following set of levels are the only vertices that are part of more than one graph from G^i:*

- *levels $j(l-k')+i, \ldots, j(l-k')+i+k'-1$, $0 \leq j \leq \lfloor\frac{k-i-l+1}{l-k'}\rfloor$ and*
- *levels $\lfloor\frac{k-i-l+1}{l-k'}\rfloor(l-k')+i+l-k'$ to $\lfloor\frac{k-i-l+1}{l-k'}\rfloor(l-k')+i+l-1$.*

Observation 6.6.5. *For at least one $i \in I$ the set of levels from Observation 6.6.4 contains at most $\frac{k'}{l-2k'}OPT$ edges from E_{opt}.*

Proof. For any two different values of i from I, the two sets of levels from Observation 6.6.4 are disjoint. Thus if for all $i \in I$, the sets of levels would contain strictly more than $\frac{k'}{l-2k'}OPT$ edges from E_{opt}, then all these levels would contain strictly more than

$$\sum_{m=0}^{\lfloor\frac{l}{k'}-2\rfloor}\frac{k'}{l-2k'}OPT = \lfloor\frac{l}{k'}-1\rfloor\frac{k'}{l-2k'}OPT \geq (\frac{l-k'}{k'}-1)\frac{k'}{l-2k'}OPT = OPT$$

edges from E_{opt}, a contradiction. □

Now for each $i \in I$, we use Lemma 6.6.1 to compute optimal solutions to MINIMUM H-SUBGRAPH EDGE DELETION for all l-outerplanar subgraphs. Since for all values of i together, the number of l-outerplanar subgraphs is bounded by $O(n)$, this can be done in time $O(n^2 2^{18l^2+9lQ} l^5 Q^2)$. We observe that for each i, any H-subgraph of G is present in at least one of the subgraphs of G induced by this i. Therefore, for each i, the union of the optimal solutions for its induced subgraphs is a set of edges that covers all H-subgraphs in G. The algorithm picks the best of these unions as an approximation to the optimal

solution. To see that this approximation is at most $(\frac{s+1}{s})OPT$, consider again an optimal solution E_{opt} for G. We pick the value i that by Observations 6.6.4 and 6.6.5 has not more than $\frac{k'}{l-2k'}OPT$ edges from E_{opt} in intersecting levels of graphs from G^i. For each element $S \in G^i$ we let E_S be the set of edges in E_{opt} in subgraph S. Furthermore, we let E'_S be the edges in an optimal solution of MINIMUM H-SUBGRAPH EDGE DELETION for S. For this choice of i, we thus have a solution of MINIMUM H-SUBGRAPH EDGE DELETION for G of size no larger than the sum of the $|E'_S|$'s. Clearly for each S it holds that $|E'_S| \leq |E_S|$. Moreover, since at most $\frac{k'}{l-2k'}OPT$ edges are counted twice while summing the $|E_S|$'s, we conclude that $\sum_{S\in G^i}|E'_S| \leq \sum_{S\in G^i}|E_S| \leq \frac{k'}{l-2k'}OPT + OPT = \frac{l-k'}{l-2k'}OPT$. Thus in total time $O(n^2)$+$O(h^2)$+$O(n^2 2^{18l^2+9lQ} l^5 Q^2)$ we constructed a solution of size at most $\frac{l-k'}{l-2k'}OPT$. By choosing $l = (s+2)k'$, we obtain the $(\frac{s+1}{s})$-approximation to OPT. Since s is strictly positive and we assume that $k > (s+2)k'$, we ensure that $2k' \leq l \leq k$. □

Next, we present an approximation scheme for MINIMUM H-SUBGRAPH EDGE DELETION on planar graph G and pattern H that is either a K_3 or a K_4.

Theorem 6.6.6. *For planar text graph G, pattern H that is either a K_3 or K_4 and any $s > 0$, there is a $O(n^2 2^{18l^2} l^5 \binom{3l-1}{h-1} h^2)$-time algorithm for* MINIMUM H-SUBGRAPH EDGE DELETION *on G that finds a solution of size at most $(\frac{s+1}{s})OPT$, where l is a constant depending on s and the outerplanarity index of H.*

Proof. Same as proof of Theorem 6.6.3, with the only difference that Lemma 6.6.2 is used instead of Lemma 6.6.1. □

6.7 Generalization to set $\mathscr{H}$ of patterns

In this final section of the chapter we generalize the results from the previous sections. We show how the algorithms can be adopted to deal with a finite set $\mathscr{H} = \{H_1, \ldots, H_t\}$, $t > 1$ of fixed, connected patterns instead of with a single fixed, connected pattern H. Consider such a set of patterns $\mathscr{H} = \{H_1, \ldots, H_t\}$, $t > 1$. First we note that if $H_i \in \mathscr{H}$ is a subgraph of $H_j \in \mathscr{H}$, $j \neq i$, then H_j is redundant. Indeed, a set of edges that covers all occurrences of H_i as a subgraph of G will also cover all occurrences of H_j as a subgraph of G. Therefore any graph G that is H_i-free is also H_j-free. For the set of cliques this means that only the smallest clique is significant and hence there is no need to generalize the result from Section 6.5. We thus generalize the results from Section 6.4. To that end, we consider text graph G of bounded maximum degree and a finite set $\mathscr{H}$ of fixed connected patterns such that for each $1 \leq i \neq j \leq t$, H_i is not isomorphic to a subgraph of H_j. By h_i, d_i and k_i we denote respectively the *size*, the *diameter* and the *outerplanarity index* of pattern H_i. The following theorem then generalizes Theorem 6.4.8.

Theorem 6.7.1. *Given a text graph G of bounded maximum degree $\Delta(G)$, a finite set of fixed, connected patterns $\mathcal{H}$ and a nice tree decomposition (T,X) of G of bounded width w, the problem* MINIMUM H-SUBGRAPH EDGE DELETION *on G can be solved in time $O(n2^{2w^2+3(w+1)Q}w^5Q^2)$, where $Q=\sum_{i=1}^{t}h_i!\binom{\Delta(G)^{d_i+1}}{h_i}$.*

Proof. We repeat the proof of Theorem 6.4.8, taking $Q=\sum_{i=1}^{t}h_i!\binom{\Delta(G)^{d_i+1}}{h_i}$ as a new upper bound on the number of considered subgraphs of G. □

Similarly, we generalize Theorem 6.6.3.

Theorem 6.7.2. *For a planar text graph G of bounded degree, finite set of fixed, connected patterns $\mathcal{H}$ and any $s>0$, there is a $O(n^2 2^{18l^2+9lQ}l^5Q^2)$-time algorithm for* MINIMUM H-SUBGRAPH EDGE DELETION *that finds a solution of size at most $(\frac{s+1}{s})OPT$, where $Q=\sum_{i=1}^{t}h_i!\binom{\Delta(G)^{d_i+1}}{h_i}$ and l is a constant depending on s and the largest outerplanarity index over all elements from $\mathcal{H}$.*

Proof. The proof is similar to the proof of Theorem 6.6.3, with the difference that $k'=\max_{1\leq i\leq t}k_i$ to guarantee that each occurrence of a pattern from $\mathcal{H}$ as a subgraph of G is present in at least one subgraph from $G^i, i\in I$. □

6.8 Conclusions

We analyzed the problem of excluding a set $\mathcal{H}$ of patterns as subgraphs of a graph G by deleting a (minimum) number of edges from G. By a general result from Courcelle and through a formulation of the problem in Monadic Second Order Logic, we implicitly showed that the decision version of the problem is solvable in linear time on graphs that have bounded treewidth. Subsequently, we presented a constructive algorithm solving the optimization version of the problem in linear time by using dynamic programming on a tree decomposition of the input graph. For this, however, we needed the additional assumption that the input graph has bounded maximum vertex degree, except for the case where $\mathcal{H}$ contains only cliques. Using Baker's layerwise decomposition approach, we created a polynomial time approximation scheme for the problem on planar graphs, using the same additional assumption. It should be noted that the constants in the time complexities of both the linear time algorithm as the PTAS are immense, severely limiting their practical usage. The question remains open whether a combinatorial linear time algorithm and a PTAS can be found that do not require the condition concerning maximum vertex degree in the input graph.

Bibliography

[1] Scott Aarsonson: *Complexity Zoo.* April 2010, `http://qwiki.stanford.edu/wiki/Complexity_Zoo`

[2] Jochen Alber, Hans L. Bodlaender, Henning Fernau, Ton Kloks, Rolf Niedermeier: *Fixed Parameter Algorithms for DOMINATING SET and Related Problems on Planar Graphs.* Algorithmica **33**(4): 461–493 (2002)

[3] Jochen Alber, Rolf Niedermeier: *Improved Tree Decomposition Based Algorithms for Domination-like Problems.* LATIN 2002: 613–628

[4] Jochen Alber, Frederic Dorn, Rolf Niedermeier: *Experimental evaluation of a tree decomposition-based algorithm for vertex cover on planar graphs.* Discrete Applied Mathematics **145**(2): 219–231 (2005)

[5] Eyal Amir: *Efficient Approximation for Triangulation of Minimum Treewidth.* UAI 2001: 7–15

[6] Stefan Arnborg: *Efficient Algorithms for Combinatorial Problems with Bounded Decomposability - A Survey.* BIT **25**(1): 2–23 (1985)

[7] Stefan Arnborg, Derek G. Corneil, Andrzej Proskurowski: *Complexity of finding embeddings in a k-tree.* SIAM Journal on Algebraic and Discrete Methods **8**(2): 277–284 (1987)

[8] Stefan Arnborg, Andrzej Proskurowski: *Linear time algorithms for NP-hard problems restricted to partial k-trees.* Discrete Applied Mathematics **23**(1): 11–24 (1989)

[9] Stefan Arnborg, Jens Lagergren, Detlef Seese: *Easy Problems for Tree-Decomposable Graphs.* J. Algorithms **12**(2): 308–340 (1991)

[10] Emgad H. Bachoore, Hans L. Bodlaender: *A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth.* AAIM 2006: 255–266

[11] Brenda S. Baker: *Approximation Algorithms for* NP*-Complete Problems on Planar Graphs.* J. ACM **41**(1): 153–180 (1994)

[12] Ann Becker, Dan Geiger: *A sufficiently fast algorithm for finding close to optimal clique trees.* Artif. Intell. **125**(1-2): 3–17 (2001)

[13] Marshall W. Bern, Eugene L. Lawler, A.L. Wong: *Linear-Time Computation of Optimal Subgraphs of Cecomposable Graphs.* J. Algorithms **8**(2): 216–235 (1987)

[14] Zhengbing Bian, Qian-Ping Gu: *Computing Branch Decomposition of Large Planar Graphs.* WEA 2008: 87–100

[15] Daniel Bienstock, Clyde L. Monma: *On the Complexity of Embedding Planar Graphs To Minimize Certain Distance Measures.* Algorithmica **5**(1): 93–109 (1990)

[16] Norman L. Biggs, E. K. Lloyd, Robin J. Wilson: *Graph Theory: 1736-1936.* Oxford: Clarendon Press (1976)

[17] Jean R.S. Blair, Pinar Heggernes, Jan Arne Telle: *A practical algorithm for making filled graphs minimal.* Theor. Comput. Sci. **250**(1-2): 125–141 (2001)

[18] Hans L. Bodlaender: *Dynamic Programming on Graphs with Bounded Treewidth.* ICALP 1988: 105–118

[19] Hans L. Bodlaender, Rolf H. Möhring: *The Pathwidth and Treewidth of Cographs.* SIAM J. Discrete Math. **6**(2): 181–188 (1993)

[20] Hans L. Bodlaender: *Improved Self-reduction Algorithms for Graphs with Bounded Treewidth.* Discrete Applied Mathematics **54**(2-3): 101–115 (1994)

[21] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, Ton Kloks: *Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree.* J. Algorithms **18**(2): 238–255 (1995)

[22] Hans L. Bodlaender, Ton Kloks, Dieter Kratsch: *Treewidth and Pathwidth of Permutation Graphs.* SIAM J.Discrete Math. **8**(4): 606–616 (1995)

[23] Hans L. Bodlaender: *A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth.* SIAM J. Comput. **25**(6): 1305–1317 (1996)

[24] Hans L. Bodlaender, Ton Kloks: *Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs.* J. Algorithms **21**(2): 358–402 (1996)

[25] Hans L. Bodlaender, Dimitrios M. Thilikos: *Treewidth for Graphs with Small Chordality.* Discrete Applied Mathematics **79**(1-3): 45–61 (1997)

[26] Hans L. Bodlaender: *Treewidth: Algorithmic Techniques and Results.* MFCS 1997: 19–36

[27] Hans L. Bodlaender: *A Partial k-Arboretum of Graphs with Bounded Treewidth.* Theor. Comput. Sci. **209**(1-2): 1–45 (1998)

[28] Hans L. Bodlaender, Ton Kloks, Dieter Kratsch, Haiko Müller: *Treewidth and Minimum Fill-in on d-Trapezoid Graphs.* J. Graph Algorithms Appl. **2**(2): 1–23 (1998)

[29] Hans L. Bodlaender, Udi Rotics: *Computing the Treewidth and the Minimum Fill-In with the Modular Decomposition.* Algorithmica **36**(4): 375–408 (2003)

[30] Hans L. Bodlaender: *Discovering Treewidth.* SOFSEM 2005: 1–16

[31] Hans L. Bodlaender, Arie M. C. A. Koster, Frank van den Eijkhof: *Preprocessing Rules for Triangulation of Probabilistic Networks.* Computational Intelligence **21**(3): 286–305 (2005)

[32] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, Dimitrios M. Thilikos: *On Exact Algorithms for Treewidth.* ESA 2006: 672–683

[33] Hans L. Bodlaender, Thomas Wolle, Arie M. C. A. Koster: *Contraction and Treewidth Lower Bounds.* J. Graph Algorithms Appl. **10**(1): 5–49 (2006)

[34] Hans L. Bodlaender, Arie M.C.A. Koster: *Combinatorial Optimization on Graphs of Bounded Treewidth.* Comput. J. **51**(3): 255–269 (2008)

[35] Hans L. Bodlaender, Alexander Grigoriev, Arie M.C.A. Koster: *Treewidth Lower Bounds with Brambles.* Algorithmica **51**(1): 81–98 (2008)

[36] Hans L. Bodlaender, Arie M. C. A. Koster: *Treewidth computations I. Upper bounds.* Inf. Comput. **208**(3): 259–275 (2010)

[37] Tom Bohman: *The triangle-free process.* Advances in Mathematics **221**(5): 1653–1677 (2009)

[38] Béla Bollobás: *Extremal Graph Theory.* London Mathematical Society Monographs, Vol. 11, Academic Press, London (1978)

[39] J.A. Bondy, U.S.R. Murty: *Graph Theory.* Springer Publishing Company, Incorporated (2008)

[40] Oleg V. Borodin, Alexei N. Glebov, André Raspaud, Mohammad R. Salavatipour: *Planar Graphs without Cycles of Length from 4 to 7 are 3-Colorable.* J. Comb. Theory, Ser. B **93**(2): 303–311 (2005)

[41] Vincent Bouchitté, Ioan Todinca: *Treewidth and Minimum Fill-in: Grouping the Minimal Separators.* SIAM J. Comput. **31**(1): 212–232 (2001)

[42] Vincent Bouchitté, Ioan Todinca: *Listing all potential maximal cliques of a graph.* Theor. Comput. Sci. **276**(1-2): 17–32 (2002)

[43] Vincent Bouchitté, Ioan Todinca: *Approximating the treewidth of AT-free graphs.* Discrete Applied Mathematics **131**(1): 11–37 (2003)

[44] Vincent Bouchitté, Dieter Kratsch, Haiko Müller, Ioan Todinca: *On treewidth approximations.* Discrete Applied Mathematics **136**(2-3): 183–196 (2004)

[45] Andreas Brandstädt, Van Bang Le, Jeremy P. Spinrad: *Graph classes: a survey.* Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)

[46] Hajo Broersma, Elias Dahlhaus, Ton Kloks: *A Linear Time Algorithm for Minimum Fill-in and Treewidth for Distance Hereditary Graphs.* Discrete Applied Mathematics **99**(1-3): 367–400 (2000)

[47] Hajo Broersma, Ton Kloks, Dieter Kratsch, Haiko Müller: *A Generalization of AT-Free Graphs and a Generic Algorithm for Solving Triangulation Problems.* Algorithmica **32**(4): 594–610 (2002)

[48] Daniel Brügmann, Christian Komusiewicz, Hannes Moser: *On Generating Triangle-Free Graphs.* Electronic Notes in Discrete Mathematics **32**: 51–58 (2009)

[49] François Clautiaux, Jacques Carlier, Aziz Moukrim, Stéphane Nègre: *New Lower and Upper Bounds for Graph Treewidth.* WEA 2003: 70–80. Lecture Notes in Computer Science **2647**, Springer-Verlag (2003)

[50] François Clautiaux, Aziz Moukrim, Stéphane Nègre, Jacques Carlier: *Heuristics and metaheuristic methods for computing graph treewidth.* RAIRO - Operations Research **38**(1): 13–26 (2004)

[51] Bruno Courcelle: *The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs.* Inf. and Comput. **85**(1): 12–75 (1990)

[52] Bruno Courcelle: *The monadic second-order logic of graphs III: tree-decompositions, minor and complexity issues.* ITA **26**: 257–286 (1992)

[53] Elias Dahlhaus: *Minimum Fill-in and Treewidth for Graphs Modularly Decomposable into Chordal Graphs.* WG 1998: 351–358

[54] Erik D. Demaine, MohammadTaghi Hajiaghayi: *Approximation Schemes for Planar Graph Problems.* Encyclopedia of Algorithms 2008

[55] Erik D. Demaine, MohammadTaghi Hajiaghayi: *Bidimensionality.* Encyclopedia of Algorithms 2008

[56] Reinhard Diestel, Tommy R. Jensen, Konstantin Yu. Gorbunov, Carsten Thomassen: *Highly Connected Sets and the Excluded Grid Theorem.* J. Comb. Theory, Ser. B **75**(1): 61–73 (1999)

[57] Reinhard Diestel: *Graph Theory.* Fourh Edition, Springer-Verlag, Heidelberg (2010)

[58] Jean E. Dunbar, Marietjie Frick: *The Path Partition Conjecture is true for claw-free graphs.* Discrete Mathematics **307**(11-12): 1285–1290 (2007)

[59] Paul Erdös, Daniel J. Kleitman, Bruce Rothschild: *Asymptotic Enumeration of K_n-free Graphs.* Atti Dei Convegni Lincei 17, Colloquio Internazionale sulle Teorie Combinatorie **20**: 19–27, American Mathematical Society, Rome (1976)

[60] Fedor V. Fomin, Dieter Kratsch, Ioan Todinca: *Exact (Exponential) Algorithms for Treewidth and Minimum Fill-In.* ICALP 2004: 568–580

[61] Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, Yngve Villanger: *Exact Algorithms for Treewidth and Minimum Fill-In.* SIAM J. Comput. **38**(3): 1058–1079 (2008)

[62] Fedor V. Fomin, Yngve Villanger: *Treewidth Computation and Extremal Combinatorics.* ICALP(1)2008: 210–221

[63] Michael R. Garey, David S. Johnson: *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W.H. Freeman & Co., New York, NY (1990)

[64] Vibhav Gogate, Rina Dechter: *A complete Anytime Algorithm for Treewidth.* UAI 2004: 201–208

[65] Martin Charles Golumbic, Haim Kaplan, Ron Shamir: *Graph Sandwich Problems.* J. Algorithms **19**(3): 449–473 (1995)

[66] Graph Classes,
`http://wwwteo.informatik.uni-rostock.de/isgci/classes.cgi`

[67] Alexander Grigoriev: *Treewidth and large grid minors in planar graphs.* unpublished manuscript (2007)

[68] Jonathan L. Gross, Jay Yellen: *Graph Theory and Its Applications.* Second Edition (Discrete Mathematics and Its Applications), Chapman & Hall/CRC (2005)

[69] Qian-Ping Gu, Hisao Tamaki: *Optimal Branch-Decomposition of Planar Graphs in $O(n^3)$ Time.* ACM Transactions on Algorithms **4**(3): (2008)

[70] Qian-Ping Gu, Hisao Tamaki: *Improved bounds on the planar branchwidth with respect to the largest grid minor size.* Technical Report **2009-17**: School of Computing Science, Simon Fraser University, Burnaby, BC, Canada (2009)

[71] Michel Habib, Rolf H. Möhring: *Treewidth of cocomparability graphs and a new order-theoretic parameter.* ORDER **11**(1): 47–60 (1994)

[72] Illya V. Hicks: *Planar Branch Decompositions I: The Ratcatcher.* INFORMS Journal on Computing (INFORMS) **17**(4): 402–412 (2005)

[73] Illya V. Hicks: *Planar Branch Decompositions II: The Cycle Method.* INFORMS Journal on Computing (INFORMS) **17**(4): 413–421 (2005)

[74] Illya V. Hicks: *Graphs, Branchwidth, and Tangles! Oh my!* Networks **45**(2): 55–60 (2005)

[75] Frank Kammer: *Determining the Smallest k Such That G Is k-Outerplanar.* ESA 2007: 359–370

[76] Pierluigi Crescenzi, Viggo Kann:
A compendium of NP optimization problems. July 2005,
`http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html`

[77] Ton Kloks, Dieter Kratsch: *Treewidth of Chordal Bipartite Graphs.* J. Algorithms **19**(2): 266–281 (1995)

[78] Ton Kloks: *Treewidth of Circle Graphs.* Int. J. Found. Comput. Sci. **7**(2): 111–120 (1996)

[79] Ton Kloks: *Treewidth: Computations and Approximations.* Lecture Notes in Computer Science **842** Springer-Verlag, Heidelberg, (1994)

[80] Ton Kloks, Dieter Kratsch, Jeremy Spinrad: *On Treewidth and Minimum Fill-In of Asteroidal Triple-Free Graphs.* Theor. Comput. Sci. **175**(2): 309–335 (1997)

[81] Arie M.C.A. Koster: *Frequency assignment - models and algorithms.* PhD thesis, University of Maastricht, Maastricht, The Netherlands, (1999)

[82] Arie M.C.A. Koster, Stan P.M. van Hoesel, Antoon W.J. Kolen: *Solving Frequency Assignment Problems via Tree-Decompositions.* Electronic Notes in Discrete Mathematics **3**: 102–105 (1999)

[83] Arie M.C.A. Koster, Hans L. Bodlaender, Stan P.M. van Hoesel: *Treewidth: Computational Experiments.* Electronic Notes in Discrete Mathematics **8**: 54–57 (2001)

[84] Arie M.C.A. Koster, Stan P.M. van Hoesel, Antoon W.J. Kolen: *Solving partial constraint satisfaction problems with tree decompositions.* Networks **40**(3): 170–180 (2002)

[85] K. Kuratowski: *Sur le problème des courbes gauches en topologie.* Fund. Math. **15**: 271–283 (1930)

[86] Jens Lagergren, Stefan Arnborg: *Finding Minimal Forbidden Minors Using a Finite Congruence.* ICALP 1991: 532–543

[87] Jens Lagergren: *Efficient Parallel Algorithms for Graphs of Bounded Tree-Width.* J. Algorithms **20**(1): 20–44 (1996)

[88] Steffen L. Lauritzen, David J. Spiegelhalter: *Local computations with probabilities on graphical structures and their application to expert systems.* Journal of the Royal Statistical Society, series B **50**: 253–258 (1988)

[89] C.G. Lekkerkerker, J.C. Boland: *Representation of finite graphs by a set of intervals on the real line.* Fund. Math. **51**: 45–64 (1962)

[90] Jan van Leeuwen: *Graph Algorithms.* Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, 522–631 (1990)

[91] Jean-Francois Manouvrier, Corinne Lucet: *Resolving the Network Reliability Problem with a Tree Decomposition of the Graph.* OPODIS 1997: 193–204

[92] George J. Minty: *On Maximal Independent Sets of Vertices in Claw-free Graphs.* J. Comb. Theory, Ser. B **28**(3): 284–304 (1980)

[93] Bruce A. Reed: *Finding Approximate Separators and Computing Tree Width Quickly.* STOC 1992: 221–228

[94] Neil Robertson, Paul D. Seymour: *Graph minors. III. Planar tree-width.* J. Comb. Theory, Ser. B **36**(1): 49–64 (1984)

[95] Neil Robertson, Paul D. Seymour: *Graph minors. II. Algorithmic Aspects of Tree-Width.* J. Algorithms **7**(3): 309–322 (1986)

[96] Neil Robertson, Paul D. Seymour: *Graph minors. X. Obstructions to tree-decomposition.* J. Comb. Theory, Ser. B **52**(2): 153–190 (1991)

[97] Donald J. Rose: *A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations.* Graph Theory and Computing, 183–217 (1972)

[98] Donald J. Rose, Robert E. Tarjan, George S. Lueker: *Algorithmic Aspects of Vertex Elimininations on Graphs.* SIAM J. Comput. **5**(2): 266–283 (1976)

[99] Najiba Sbihi: *Algorithme de Recherche d'un Stable de Cardinalité Maximum dans un Graphe sans Étoile.* Discrete Math. **29**(1): 53–76 (1980)

[100] Paul D. Seymour, Robin Thomas: *Graph Searching and a Min-Max Theorem for Tree-Width.* J. Comb. Theory, Ser. B **58**(1): 22–33 (1993)

[101] Paul D. Seymour, Robin Thomas: *Call Routing and the Ratcatcher.* Combinatorica **14**(2): 217–241 (1994)

[102] Kirill Shoikhet, Dan Geiger: *A Practical Algorithm for Finding Optimal Triangulations.* AAAI 1997: 185–190

[103] Yinglei Song, Chunmei Liu, Russell L. Malmberg, Fangfang Pan, Liming Cai: *Tree Decomposition Based Fast Search of RNA Structures Including Pseudoknots in Genomes.* CSB 2005: 223–234

[104] Ravi Sundaram, Karan Sher Singh, C. Pandu Rangan: *Treewidth of Circular-Arc Graphs.* SIAM J. Discrete Math. **7**(4): 647–655 (1994)

[105] Jan Arne Telle, Andrzej Proskurowski: *Algorithms for Vertex Partitioning Problems on Partial k-Trees.* SIAM J. Discrete Math. **10**(4): 529–550 (1997)

[106] Robin Thomas: *Tree-Decompositions of Graphs.*
http://people.math.gatech.edu/~thomas/SLIDE/CBMS/trdec.pdf

[107] K. Thulasiraman, M.N.S. Swamy: *Graphs: Theory and Algorithms.* John Wiley & Sons, Inc., New York, NY (1992)

[108] K. Wagner: *Über eine Eigenshaft der ebenen Complexe.* Math. Ann. **14**: 570–590 (1937)

[109] T.V. Wimer, S.T. Hedetniemi, R. Laskar: *A methodology for constructing linear graph algorithms.* Congressus Numerantium **50**: 43–60 (1985)

[110] Gerhard J. Woeginger: *Exact Algorithms for NP-hard Problems: A Survey.* Combinatorial Optimization 2001: 185–208

[111] Thomas Wolle: *A Framework for Network Reliability Problems on Graphs of Bounded Treewidth.* ISAAC 2002: 137–149

[112] Jinbo Xu, Feng Jiao, Bonnie Berger: *A Tree-Decomposition Approach to Protein Structure Prediction.* CSB 2005: 247–256

[113] Mihalis Yannakakis: *Edge-Deletion Problems.* SIAM J. Comput. **10**(2): 297–309 (1981)

Nederlandse Samenvatting

Een boom decompositie van een graaf is een weergave van de graaf door middel van een boom, waarbij de knooppunten van de decompositie deelverzamelingen van de punten van de graaf zijn. De breedte van een boom decompositie wordt gedefinieerd als de omvang van het grootste knooppunt, verminderd met één. De best mogelijke breedte (de kleinste) over alle boom decomposities van een graaf noemen we de boombreedte van de graaf. Een klasse van grafen heeft een begrensde boombreedte als de boombreedte begrensd wordt door een constante die niet afhankelijk is van het aantal punten in de graaf. De boombreedte is een belangrijke structuurparameter van een graaf. Intuïtief geeft ze aan hoe "dik" de boom is waarmee de graaf beschreven kan worden. Het belang van de (begrensde) boombreedte is te vinden in de oplosbaarheid van een groot aantal optimaliseringsproblemen op grafen, zoals minimale kleuring, grootste onafhankelijke verzameling, en diverse netwerkproblemen zoals capaciteitsplanning. Deze problemen kunnen met behulp van de beste boom decompositie, door middel van dynamische programmeringsalgoritmen opgelost worden in een tijd die polynomiaal afhangt van de grootte van de graaf en exponentieel van de boombreedte van de graaf. Met andere woorden: deze problemen kunnen efficiënt opgelost worden op grafen waarvoor de boombreedte begrensd is. Een uitgewerkt voorbeeld van een grafenprobleem dat met behulp van een dynamisch programmeeralgoritme op een boom decompositie wordt opgelost komt in dit proefschrift aan de orde. Helaas is de boombreedte van een graaf niet eenvoudig te bepalen: het probleem is *NP*-moeilijk. In dit proefschrift wordt een nieuw (exponentieel) algoritme beschreven om de boombreedte exact te bepalen met behulp van een boomzoek methode en een heuristiek voor de boombreedte waarbij met een polynomiaal algoritme goede (maar niet per se optimale) boom decomposities geconstrueerd worden. Daarnaast worden structuren beschreven waarvan bekend is dat ze gerelateerd zijn aan de boombreedte van een graaf, zoals roosters. De aard van de samenhang tussen de boombreedte en deze structuren wordt in dit proefschrift nader onderzocht.

In Hoofdstuk 3 wordt een boomzoek algoritme beschreven voor het exact bepalen van de

boombreedte. Dit algoritme is gebaseerd op een splitsingsregel die punten paren selecteert en hiervoor twee keuzes laat: het wel of niet samen voorkomen van de twee punten in een knooppunt van de uiteindelijke boom decompositie. Deze regel laat krachtige processing toe, zodat voor kleine grafen (tot ongeveer 30 punten) de boombreedte bepaald kan worden, met name als de graaf weinig kanten heeft in verhouding tot het aantal punten. Voor grotere grafen is dit exacte algoritme in de praktijk echter niet bruikbaar. In Hoofdstuk 4 wordt daarom een heuristische methode beschreven voor het vinden van boom decomposities met kleine, maar niet noodzakelijkerwijs optimale breedte. Deze methode maakt gebruik van een nieuwe buurruimte structuur, waarbij verbonden knooppunten van een decompositie eerst worden samengevoegd en daarna weer worden gescheiden zodanig dat daarmee een nieuwe boom decompositie verkregen wordt. De buurruimte structuur is ingebed in een lokaal zoek algoritme, met meerdere starts. Dit algoritme is competitief met bestaande heuristische methoden voor de boombreedte die onder andere gebaseerd zijn op "tabu search" en "simulated annealing".
In Hoofdstuk 5 wordt gekeken naar planaire grafen. Voor een rooster, een eenvoudig type planaire graaf, geldt dat de boombreedte wordt bepaald door de afmeting van het rooster. Hieruit volgt dat de klasse van planaire grafen geen begrensde boombreedte heeft. Het is nog altijd niet aangetoond dat het bepalen van de boombreedte van planaire grafen een *NP*-moeilijk probleem is. Wel is bekend dat de boombreedte van een planaire graaf van onderen wordt begrensd door zowel de grootte van de grootste rooster minor in de graaf als door de "branchwidth" van de graaf. Ook bestaan er bovengrenzen voor de boombreedte van planaire grafen in termen van de twee bovengenoemde parameters afzonderlijk. In dit proefschrift wordt een klasse van planaire grafen opgesteld waarmee wordt aangetoond dat de bestaande bovengrens voor boombreedte in termen van "branchwidth" scherp is en dat de bovengrens in termen van rooster minoren zeker niet verder aangescherpt kan worden dan tot anderhalf keer de grootte van de grootste rooster minor. Daarbij wordt een nieuwe methode gebruikt om de grootte van de grootste rooster minor te bepalen, gebaseerd op afstanden tussen de vlakken in een planaire inbedding van de graaf. Uit ander onderzoek is inmiddels gebleken dat er een klasse van planaire grafen bestaat waarvoor de boombreedte tweemaal zo groot is als de grootste rooster minor.
Hoofdstuk 6 richt zich tenslotte op de aan- of afwezigheid van subgrafen in een graaf en illustreert het gebruik van boom decomposities in algoritmen. Diverse belangrijke graafklassen kunnen worden gekarakteriseerd door uitsluiting van een verzameling subgrafen. Het grafenprobleem waarvoor in dit hoofdstuk algoritmen worden gepresenteerd luidt: hoeveel kanten dienen minimaal uit een graaf G te worden verwijderd om een verzameling H van grafen niet meer als subgraaf in G aanwezig te laten zijn? Dit probleem is *NP*-moeilijk voor algemene grafen. Voor speciale gevallen van grafen G met begrensde boombreedte en verzamelingen H presenteren we lineaire constructieve algoritmen, gebaseerd op dyna-

mische programmering. Verder wordt in dit hoofdstuk aangetoond dat met een laagsgewijze decompositie aanpak een optimale oplossing voor het subgraaf eliminatie probleem op planaire grafen efficiënt kan worden benaderd.

Curriculum Vitae

Lambertus Marchal was born on October 20, 1978 in Maarn, The Netherlands. In 1997, he received his Gymnasium diploma from the Van Lodenstein College in Amersfoort. In September of that same year he started studying Applied Mathematics at the University of Twente, where he graduated at the Chair of Discrete Mathematics and Mathematical Programming in January 2005. In September 2005, he started his doctoral research at the department of Quantitative Economics at the Faculty of Economics and Business Administration of Maastricht University, the results of which are presented in this thesis. Since November 2010, Bert has been employed at ABF Research in Delft.