Computational Aspects of Treewidth

Lower Bounds and Network Reliability

Computational Aspects of Treewidth

Lower Bounds and Network Reliability

Computationele Aspecten van Boombreedte
Ondergrenzen en Betrouwbaarheid van Netwerken
(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. Dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op maandag 13 juni 2005 des middags te 12.45 uur door

Thomas Wolle

geboren op 29 oktober 1975 te Gera

promotor:	Prof. Dr. Jan van Leeuwen Faculty of Science Utrecht University
copromotor:	Dr. Hans L. Bodlaender Faculty of Science Utrecht University

The work in this thesis (IPA Dissertation series number 2005-09) has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was partially supported by EC contract IST-1999-14186: Project ALCOM-FT (Algorithms and Complexity - Future Technologies) and partially by the Netherlands Organisation for Scientific Research NWO (project *Treewidth and Combinatorial Optimisation*).

Printed by Ridderprint offsetdrukkerij B.V.

ISBN 90-393-3972-4

Contents

Preface

If I remember correctly, it was on Thursday, 4 January 2001, when the supervisor of my diploma thesis told me about a PhD-student position in Utrecht. One and a half months later, I applied for this position, and in March 2001 I was invited for an interview, which resulted in a positive outcome. After that, it still took me a couple of weeks to make up my mind whether or not I should go to Utrecht. It would be a big step for me, after growing up in a one-horse-village with around six hundred inhabitants, to move to such an incredibly world-famous metropolis as Utrecht. Now four years later, you know about my decision, and you hold a proof in your hands that these years have been successful at least to some extent. Of course there were days when I was thinking that the whole world turned against me, but looking back there was not a single day when I seriously regretted my decision to come to The Netherlands. I can say that during my time in Utrecht, I learned more about life in general than about science. However, this does not mean that I did not learn anything scientific.

Already before I started writing this thesis, I thought that writing the section where I say 'thank you' for personal matters would be one of the most difficult ones. Not because saying 'thank you' is so difficult, but because of the difficulties of adequately distributing my big 'THANK YOU', and because of the well-known 'thank you' paradox, which is: the more persons and details one mentions, the higher the probability that one forgets someone. Therefore, I would like to thank everyone who made the last four years possible as an enjoyable and successful time. Thanks to all those who accompanied, supported and encouraged me through these interesting years. However, to make it a bit more personal after all, I mention some groups of people rather than individuals (multiple occurrences of persons are possible). I would like to thank my colleagues (this also includes former colleagues, office-mates and student-assistants), all the girls and guys from UFC ('Sticky Fingers') Utrecht and the Neutel-voetballers (or Neutel-hoopsters). A very big thank you goes to my foreign friends, my Dutch friends, my friends in Germany and of course my family.

Utrecht, April 2005 Thomas Wolle

1 Introduction

Graphs are an essential part of many systems. They are used for representing and modelling the structure of data, systems, scenarios, projects and much more. On these abstract models, scientists try to solve problems relevant for the underlying real-world situation. Solving problems often means finding a strategy to manipulate the data that leads to an efficient algorithm. It can also mean constructing an algorithm to optimise some value based on the input. See e.g. [35] and [53] for an introduction to algorithms and to graph theory. An algorithm is considered to be efficient if its running time for solving a problem grows polynomially in the size of the input (the number of bits needed to denote the input). However, there are many practically relevant problems (see e.g. [49, 74]) for which such efficient algorithms are not known – they might not even exist. These problems appear to possess an inherent hardness. One form of computational hardness is derived from the theory of nondeterministic polynomial-time computability and is called NP-hardness, see [49] for more details on this theory. Currently, there are no NP-hard problems for which a polynomial-time algorithm is known, and many scientists expect that no NP-hard problem has such a polynomial-time algorithm (see e.g. [56, Remark 10.2]).

In the real world it is not possible to restrict the classes of problems to be solved to those having an efficient algorithm. We have to face NP-hardness. In the course of time, several approaches were developed to cope with NP-hard problems. One way of solving an NP-hard problem is to use an algorithm (e.g. based on backtracking or branch-and-bound) that enumerates and evaluates all possible solution-candidates. Such methods are usually considered to be inefficient, because they have exponential or super-exponential running times. Another way to cope with NP-hardness is to use approximation algorithms and/or heuristics. Approximation algorithms are by definition efficient algorithms, for which it is possible to prove that the results they give are always close to the optimum solutions. For heuristics, we do not have such a proof. Experiments can be used to estimate the quality (and running times) of heuristics and approximation algorithms. It is also possible to tackle NP-hard problems from yet a different angle. If we can enforce or ensure that our instances have a certain property, i.e. if the input graph belongs to a special graph class (see [26] for a survey on graph classes), then it might be possible to take advantage of that property when attempting to design an efficient algorithm for solving a hard problem. The property of a graph to be a tree helps very much algorithmically, in the sense that many hard problems are easy on trees. However, often enough, we have to deal with more complicated graphs. One possibility to generalise or to extend the property of a graph to be a tree is captured by the notion of ‘treewidth’. The treewidth of a graph is a parameter that, vaguely spoken, indicates the similarity of the graph to a tree. The smaller the treewidth of a graph the more tree-like it is. It turns out that the easiness of many problems on trees (that are hard on general graphs) carries over to graphs with small treewidth. These graphs will be the central theme in this thesis.

1.1 Treewidth

Informal Description

About twenty years ago, Robertson and Seymour introduced the notions tree-decomposition, treewidth and graphs of bounded treewidth [79, 80]. Other notions had been developed, which are equivalent to graphs of bounded treewidth, e.g. partial k-trees (see [85]). Nowadays, the terminology introduced by Robertson and Seymour is most frequently used.

Since their introduction these notions have received a growing interest, both from the theoretical and from the practical point of view. To simplify the following informal description of the relevant terms, we assume that there is a path between any pair of vertices in our original graph G, i.e. we assume our graph is connected. Most of the times solving problems on disconnected graphs means solving a problem on each connected component, and hence, this can be interpreted as only considering connected graphs. The graph parameter *treewidth* is defined using the notion *tree-decomposition*. A tree-decomposition consists of a tree T and a set of subsets of vertices of the graph G, where one subset is associated to every node (or vertex) of T. A tree-decomposition represents the structure of the graph G. In particular, to construct a tree-decomposition we

- 'group' the vertices of G together to form 'bags' (these bags are sometimes called 'supernodes'; each bag is a subset of the vertices of G) and
- connect these bags to form a rooted tree T (the bags play the role of vertices of the tree T).

Note that the bags do not have to be disjoint, i.e. there might be vertices of the graph contained in more than one bag. This grouping and connecting has to be done in such a way that

- for all edges e of the graph: there exists a bag containing both endpoints of edge e, and
- for all vertices v of the graph: all bags containing v together with the edges between those bags in T form a connected subtree of T.

The width of a tree-decomposition is the number of graph-vertices of a largest bag minus 1. The treewidth of a graph G is the smallest possible width over all tree-decompositions of G. Section 2.4 provides a formal definition of tree-decomposition and treewidth, and Figure 2.4 depicts a tree-decomposition of the graph in Figure 2.1. Graphs of bounded treewidth are graphs whose treewidth is bounded from above by some constant k. They are also called partial k-trees (see e.g. [11] for more related notions).

Trees vs. Tree-decompositions

One reason why many problems that are hard in general are easy on trees is that trees do not contain cycles. When solving a problem on a tree it is often possible to make 'local' decisions for a vertex or for a subtree. These local decisions might influence the decisions that have to be made in another part of the tree. However, they will be 'propagated' to these other parts via a single unique path, since there are no cycles. Hence, the effect of local decisions to another part of the tree can be captured by information concerning only one path.

Another way of describing this issue is to use *separators*. A separator is a set of vertices whose removal from the graph (including the removal of the edges connected to the removed vertices) results in a disconnected graph (see e.g. [51, 53] for more formal definitions). The set $\{b, e\}$ is a separator of the graph in Figure 2.1. Thus, each internal vertex in a tree, i.e. a vertex connected to more than one edge in a tree, is a separator.

Very often it is easier to solve problems on a graph by decomposing the graph into smaller components, solving the corresponding (sub)problems on the smaller components and combining the solutions of the subproblems to a solution of the original problem. In this process separators play the role of the interface between two (or more) components of the graph. It turns out that the size of the separator often determines the number of solutions to the (sub)problems; and in turn this number determines the algorithmic running time when using this approach to solve the problem computationally. In many cases the running time is exponential in the size of the separator. However, as each internal vertex of a tree is a separator of size one, many problems are solvable very efficiently on trees.

One way of interpreting the notion of a tree-decomposition of width k is to understand it as an organised representation of separators of size at most $k + 1$, because the vertices of each bag can be shown to be a separator. Using these separators of size at most $k + 1$, we can iteratively decompose the graph into smaller components until we have components with at most $k + 1$ vertices.

Dynamic Programming with Tree-decompositions

A different approach to describe, how tree-decompositions are used to solve problems involves dynamic programming (see [36] for more details on this algorithm design paradigm). Let us assume we are given a tree-decomposition of graph G. Since this tree-decomposition is represented as a rooted tree T, the ancestor/descendant relation is well-defined. We can associate to each bag X the subgraph of G made up by the vertices in X and all its descendant bags, and all the edges between those vertices. Starting at the leaves (the nodes with no descendants), we can compute information typically stored in a table, in a bottom-up manner for each bag until we reached the root. This information is sufficient to solve the subproblem for the corresponding subgraph. To compute the table for a node of the tree-decomposition, we only need the information stored in the tables of the children (i.e. direct descendants) of this node. This is typical for dynamic programming. The problem for the entire graph can then be solved with the information stored in the table of the root of T. This approach is explained much more elaborately in [8, 10]. Often the size of the tables is exponential in the width of the tree-decomposition used, which in turn is crucial for the running times to compute these tables. We therefore would like to use tree-decompositions of width as small as possible. The smallest possible width is the treewidth of the graph and is NP-hard to compute in general.

Computing Tree-decompositions

Despite the fact that computing the treewidth is NP-hard [1], much work has been done on practical algorithms for determining the treewidth of graphs.

In [9], an algorithm is given that decides whether the treewidth of a graph G is at most k (if k is a constant and not a part of the input to the problem) and if so, outputs a corresponding tree-decomposition. The algorithm's running time is linear in the size of G, but it has a very large constant factor at least exponential in k. Therefore, this algorithm is not applicable in practice (unless k is known to be very small) for finding the treewidth (see also [84]). A more practical exact method is presented in e.g. [89]. Recently, an exact branch-and-bound algorithm was given in [50]. Another method to progress towards tree-decompositions of small width is based on preprocessing. Such methods apply certain rules for stepwise reducing the graph. If no rules can be applied anymore, exact methods have to be used on the (much) smaller graph, which give a tree-decomposition of the smaller graph.

Because during the entire process the optimality of the solution is preserved, this tree-decomposition can then be extended to one of the original graph by undoing the reductions. See [15, 16, 46] for more details. Tree-decompositions can also be computed using heuristic methods. This will in general not yield tree-decompositions of minimum width, however. For the problem at hand the heuristically computed tree-decomposition might be applicable, if its width is small enough for practical purposes. These heuristics always give an upper bound on the treewidth. More details on such upper bound heuristics can be found in e.g. [32, 31, 63, 62, 5, 50].

In many cases, exact methods are too slow, and for many instances, there are large gaps between the bounds given by upper bound and lower bound heuristics. Thus, the study of algorithms and heuristics for treewidth is highly interesting also from a practical point of view.

Applications

Exploiting tree-decompositions of graphs is possible in many areas, including VLSI layout, mathematics, expert systems, evolution theory and natural language processing (see [8] for more details). Also, most existing programmes (or better: their control flow graphs) have small treewidth if they are structured (e.g. if they are 'goto-free'). By using this property, they allow efficient automatic static analysis for many well-known problems, such as register allocation (see [92] for programming languages like C or Pascal and [28] for programmes in Ada). In the frequency assignment problem [62, 64] for cellular wireless networks, it has been shown that dynamic-programming algorithms with tree-decompositions can also be used in practice to compute the optimal solution.

From a theoretical point of view, there are very interesting results on graphs of bounded treewidth. These results state that each graph problem that can be expressed by a formula using certain language constructions, can be solved in linear time on graphs of bounded treewidth. This logic language is called Monadic Second Order Language (MSOL), see among others [39, 40], and e.g. [2, 25] for extensions of it. However, even if the running time is linear from a theoretical point of view, these methods often result in very large constant factors hidden in the big O-notation.

One practical application that we will consider in more detail concerns probabilistic networks. These networks are also called (Bayesian) belief networks and are used in decision support systems and expert systems. Informally, decision support systems are systems that take as input a set of facts or observations (e.g. symptoms of a patient), and after probabilistic inference, they give an output that can be used to make a decision concerning the current situation (e.g. make a diagnosis of a patient). A probabilistic network consists of directed acyclic graph, where each vertex represents a statistical variable that can take one of finitely many values. Dependences and independences of the variables are modelled by the directed edges of the acyclic directed graph. An arc (i.e. a directed edge) from vertex u to vertex v in the directed graph means that the state of vertex u causes a direct effect on vertex v. Associated with each vertex v is a function which is a set of (conditional) probabilities. This function describes the influence of all vertices that have an arc to vertex v on the probabilities of the value of the vertex v. Altogether this represents a joint probability distribution. See e.g. [58] for more details on probabilistic networks.

Computing the conditional probabilities of all vertices in a probabilistic network is called probabilistic inference, a problem that is NP-hard in general [34]. One of the best known algorithms (see [95]) for probabilistic inference is the algorithm by Lauritzen and Spiegelhalter [68]. This algorithm takes an undirected triangulated moral graph G' of the directed graph G as input. The moral graph of G is the underlying undirected graph of G augmented by edges between any pair of vertices u and w such that there are arcs from u to v and from w to v in G for a vertex v. Based on this moral graph of G, a triangulation G' of it has to be made, i.e. by adding further edges, such that every simple cycle of length at least 4 possesses a chord (an edge joining two nonconsecutive vertices on the cycle). For us it is important that Lauritzen and Spiegelhalter's algorithm has computational complexity of $O(n \cdot 2^c)$, where n is the number of vertices of G and c is the size of a largest clique in G'. Finding a triangulation G' of the moral graph of G with a maximum clique size that is as small as possible is equivalent to the problem of finding a tree-decomposition of the moral graph of G with width as small as possible. Lauritzen and Spiegelhalter's algorithm solves the probabilistic inference problem by dynamic programming with the triangulated graph G'. In other words, this algorithm solves the probabilistic inference problem using a tree-decomposition of the moral graph of G.

1.2 Motivation

Using Tree-decompositions

Since computing the treewidth is NP-hard, it is rather unlikely to find efficient algorithms for computing the treewidth or a tree-decomposition of smallest possible width. One could ask now 'What do we gain if we use dynamic programming with a tree-decomposition to solve an NP-hard problem, while computing the treewidth is NP-hard?' This reasonable question can be answered in the following ways.

First of all, it might be that we have a very specific problem such that the underlying graph has small treewidth. For instance, we may have a telecommunications network that is rather sparse to reduce its cost. Hence, its treewidth could be suitably small. Perhaps it is even the case that our specific application uses graphs that are already represented by a tree-decomposition of (close to) minimum width. In such a case, we do not have to compute a tree-decomposition ourself, but instead we are given it. Then we can directly concentrate on the design of an algorithm to solve the particular problem.

Another reason for using the approach of dynamic programming with tree-decompositions is the reusability of tree-decompositions. It might be worth to spend much running time to find a tree-decomposition of width as small as possible, because we can use this tree-decomposition multiple times. Computing the treewidth or tree-decompositions with (close to) optimal width of a probabilistic network is such an example. It is sensible to spend much time on computing a tree-decomposition of small width of a moral graph of a probabilistic network's directed graph. As we have seen above, the smaller the width of this tree-decomposition, the faster the probabilistic inference. Once we have a tree-decomposition of (close to) optimal width, we can solve the probabilistic inference problem many times for different situations (e.g. for different patients, if we consider a medical decision support system).

Furthermore, computing a tree-decomposition of minimum width (which is NP-hard) might not be necessary. As mentioned earlier, if an upper bound method computing a tree-decomposition gives a result that is good enough, we do not need to find an optimal tree-decomposition. This means that we do not have to compute the treewidth, if our dynamic-programming method with the heuristically obtained tree-decomposition has a running time that is fast enough for our purposes.

Even if all criterias above do not apply to our current situation, it is possible to spend much time on computing a tree-decomposition of minimum width and applying dynamic programming. However, deciding whether this is worth doing it, heavily depends on the considered problem. An example of this situation is the work in [62].

Treewidth Lower Bounds

Lower bounds for treewidth are examined in e.g. [14, 32, 70, 77] and Chapters 3 and 4. They are useful for a couple of reasons.

As mentioned earlier, the running times of dynamic-programming algorithms with tree-decompositions of minimum width are often exponential in the treewidth of the graph. Hence, treewidth lower bounds are useful for estimating the running times of such methods. A large lower bound on the treewidth of a graph implies that there is little hope for an efficient dynamic-programming algorithm based on a tree-decomposition of that graph. To apply this reasoning, we need the running times of our dynamic-programming algorithm, i.e. we need the size of the tables used for this approach. This size depends on our skill to design efficient algorithms, on the problem itself and on the width of the used tree-decomposition. However, we do not need to compute such a tree-decomposition.

We also mentioned earlier that hard problems can be solved with backtracking. Computing the treewidth of a graph is such a hard problem. Backtracking means systematically constructing all solution candidates. This construction process can be represented by a computation-tree. If we have good lower and upper bounds on the treewidth, we can use branch-and-bound which cuts out branches of the computation-tree as soon as we know that we will not find a good solution in this branch. The better the bounds, the bigger the branches that can be pruned in a branch-and-bound method, and therefore, the smaller the running times. Hence, good treewidth lower bounds can be utilised to decrease the running time of branch-and-bound algorithms. Research on this approach has been carried out in e.g. [50].

In addition, lower bounds in connection with upper bounds help to assess the quality of these bounds. If these bounds are close to each other then we have a good approximation of the optimum. Of course, it would be best, if the lower bound equals the upper bound. In that case, we found the optimum. However, if the gap between lower and upper bound is very big, we know that at least one of these bounds is not very sharp on the graph under consideration.

Of course, apart from previous motivations based on practical issues, treewidth lower bounds are an interesting and fascinating topic purely from the theoretical point of view.

Work on treewidth lower bounds is reported in Chapters 3-5.

Network Reliability

Most of us have been in contact with an information or communication network, such as the Internet, a local area network or the plain old telephone system. Small local area networks are often designed in a tree structure, because this is the cheapest way of connecting a set of communicating sites, i.e. computers. The disadvantage of networks with a tree structure is that if a single connection between two sites breaks down, then communication between any sites belonging to the two different connected components is not possible anymore. In the Internet for example, this problem was overcome by using more links. Hence, if one link breaks down, it usually is still possible to find a path between two given sites in the network along which to communicate.

The reliability of a network measures its quality in the following sense. The higher the probability that communication in the network is still possible after some of its parts (e.g. links, switches or routers) break down, the higher the reliability of the network. Clearly, the reliability of networks is of great importance, since networks are an essential part of current information and communication technology. While the networks in many areas grow rapidly, specialists designing and maintaining them must be able to determine the reliability of a given network or of one to be constructed.

Hence, we would like to solve the problem to compute its reliability, for any given network with probabilities of its elements of breaking down. Also, several variants of the notion of reliability are of interest, e.g. computing the probabilities that a given pair of vertices still can communicate with each other. Unfortunately, many such questions are NP-hard. In Chapters 6 and 7, we study network reliability and variants on general graphs and graphs of bounded treewidth.

1.3 Thesis Outline

In Chapter 2, we state basic terminology. We also formally define the terms treewidth, tree-decomposition and minor, and we elaborate on the notions of edge-contraction and edge-subdivision.

Chapter 3 is devoted to methods for computing treewidth lower bounds. We consider three approaches to obtain or improve such bounds. The first one is based on the degeneracy of a graph (maximum over all subgraphs of the minimum degree), which is known to be a treewidth lower bound. The second approach uses Lucena's treewidth lower bound [70] based on Maximum Cardinality Search. This research leads to a number of parameters that are treewidth lower bounds. We examine relations between these parameters and their computational complexity. The last approach for obtaining and improving treewidth lower bounds is based on a technique introduced by Clautiaux et. al [32]. We combine all these methods with edge-contraction and see that this is an excellent way for improving upon existing treewidth lower bounds.

Some of the treewidth lower bounds that we will consider in Chapter 3 are exactly computable in polynomial time. For these parameters, we develop algorithms for computing them in Chapter 4. Other parameters are NP-hard to compute, and hence, we propose some heuristics to compute lower bounds for these parameters. We also experimentally evaluate these algorithms and heuristics. At the beginning of Chapter 4, we examine a data structure that can be used in some of the considered algorithms and heuristics.

One of the parameters that will be defined in Chapter 3 is the contraction degeneracy. It combines the degeneracy with edge-contraction (or minors). Due to its elementary definition, this parameter appears to be an attractive object of study in its own, and not only as a treewidth lower bound. Computing the contraction degeneracy is NP-hard on general graphs. That is why we look at a special graph class, namely cographs, in Chapter 5. There we develop a polynomial time algorithm for computing the contraction degeneracy on cographs.

A model for network reliability problems is consider in Chapter 6. Based on this model, we give a list of relevant network reliability problems. We analyse the computational complexities of these problems, which involves the definition of a complexity class $\#P'$ as an extension to $\#P$. We show the listed network reliability problems to be $\#P'$-complete, by giving a membership-proof and a transformation from a known $\#P'$-hard problem. At the start of Chapter 6, we consider some classical network reliability problems.

Since in Chapter 6 we prove some network reliability problems to be $\#P'$-complete, we look at how to solve these problems on a special graph class in Chapter 7. There, we develop a framework that can be used to solve these network reliability problems (most of the times efficiently) on graphs of bounded treewidth. With the framework we can answer questions that ask for the probability that there is a connection between vertices of two distinguished sets of vertices. As an example, we might have a set of servers and a set of clients. Then we can ask the question: 'What is the probability that each client is connected to at least one server?' Among others, this question can be answered efficiently on graphs of bounded treewidth.

2
Preliminaries

This chapter covers the basic terminology that we will be using throughout this thesis. We start with general terms and concepts. Most of this is standard graph theory/algorithm terminology. We also elaborate on the notion of edge-contraction. We explain subgraphs and minors, we define tree-decompositions and treewidth, and we elucidate subdivisions. Part of this chapter is adapted from cooperation [98] with Hans L. Bodlaender.

2.1 General

A *directed graph* G is a pair consisting of a finite set of *vertices* $V(G)$ and a multiset of *edges* $E(G)$. Each edge $e \in E(G)$ is an ordered pair of vertices, i.e. $e \in V \times V$. Each pair $f \in V \times V$ with $f \notin E(G)$ is a *nonedge*. The two vertices that are joined by an edge are called the *endpoints* of the edge. An edge $e \in E(G)$ is undirected, if e is a two-element multiset of vertices rather than a pair of vertices. A graph G is *undirected* if all its edges are undirected. Graph G is *simple* if it has neither self-loops (i.e. edges with only one endpoint) nor multiedges, i.e. $E(G)$ is a set and not a multiset. Figure 2.1 depicts an example of a simple, undirected graph G with vertex-set $V(G) = \{a, b, ..., l\}$. Some edges of G are $\{a, b\}$, $\{e, h\}$ and $\{j, k\}$.

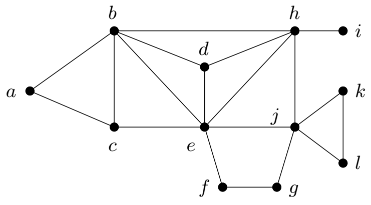

Figure 2.1. Graph G

Throughout this thesis $G = (V(G), E(G))$ denotes a simple, undirected graph, with $V(G)$ the set of vertices and $E(G)$ the set of edges. We use V and E instead of $V(G)$ and $E(G)$, respectively, if it is clear from the context which graph is meant. Two vertices are said to be *adjacent*, if they are joined by an edge. In Figure 2.1, e.g. vertices c and e are adjacent, and they are the endpoints of edge $\{c, e\}$. Two edges $e_1 = \{v_1, v_2\}$ and $e_2 = \{v_3, v_4\}$ are *disjoint* if they do not have an endpoint in common, i.e. if $e_1 \cap e_2 = \emptyset$.

A sequence of vertices $v_0, ..., v_l$ is a *path* from v_0 to v_l of length l in G if v_{i-1} and v_i are adjacent for $i = 1, 2, ..., l$. A path is *simple* if all vertices in the path are pairwise distinct. The sequence a, c, e, h, j is a simple path of length 4 in G in Figure 2.1. A sequence of vertices $v_0, ..., v_l$ is a *cycle* of length l in G if $v_0 = v_l$ and $\{v_{i-1}, v_i\} \in E$ for $i = 1, 2, ..., l$. A *simple cycle* is a cycle where all vertices in the cycle are pairwise distinct. Sequence b, c, e, h, d, b is a simple cycle of length 6 in the example graph in Figure 2.1. A (directed) graph is *acyclic* if it does not have a (directed) cycle. A graph G is *connected* if between any two vertices of G there is a path joining them. A *tree* is a graph that is acyclic and connected. $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A *connected component* of G is a maximal subgraph of G that is connected. A graph G can consist of connected components $O_1, ..., O_q$. In that case, we write $G = O_1 \cup ... \cup O_q$, where $\cup$ denotes the disjoint union of graphs, and each O_i ($1 \leq i \leq q$) is connected. The graph G in Figure 2.1 is connected and hence, consists of exactly one connected component.

Unless otherwise stated, $n(G)$ (or simply n) denotes the number of vertices in G, i.e. $n := |V|$, and $m(G)$ (or simply m) denotes the number of edges $m := |E|$. For G in Figure 2.1, we have that $n = 12$ and $m = 19$. The *open neighbourhood* $N_G(v)$ or simply $N(v)$ of a vertex $v \in V$ is the set of vertices adjacent to v in G. In Figure 2.1, $N_G(j) = \{e, g, h, k, l\}$. $N(S)$ for $S \subseteq V$ denotes the open neighbourhood of S, i.e. $N(S) = \bigcup_{s \in S} N(s) \setminus S$. The *degree* in G of vertex v is $d_G(v) = |N(v)|$ or simply $d(v)$. For instance, we have for G as in Figure 2.1, $d(i) = 1$ and $d(e) = 6$. We denote with $\delta(G)$ the minimum degree of graph G, i.e. $\delta(G) := \min_{v \in V} d(v)$. A vertex v is *universal* in G, if v is adjacent to each vertex $w \in V, w \neq v$. Graph G in Figure 2.1 does not possess a universal vertex. For $v \in V$, we denote by $G - v$ the graph obtained from G by deleting v, i.e. $G - v := G[V \setminus \{v\}]$, and for $e \in E$, we define $G - e := (V, E \setminus \{e\})$.

A set $W \subseteq V$ is a *clique* in G, if for all $v, w \in W$ with $v \neq w$, it holds that $\{v, w\}$ is an edge in G. Example cliques in Figure 2.1 are $\{b, e\}$, $\{j, k, l\}$ and $\{b, d, e, h\}$. A set $U \subseteq V$ is a *vertex cover* of G, if for each edge $\{v, w\} \in E$ at least one of v and w belongs to U. A vertex cover of size 7 of G in Figure 2.1 is e.g. $\{a, b, e, g, h, k, l\}$. The decision problem related to the notion vertex cover is NP-complete [60, 49] and is formulated as follows:

Problem: VERTEX COVER
Instance: Graph $G = (V, E)$ and $k \in \mathbb{N}$.
Question: Is there a vertex cover U of G with $|U| \leq k$?

Note that $\mathbb{N}$ is the set of the nonnegative integers, and $\mathbb{Q}$ denotes the set of rational numbers.

2.2 Edge Contraction

In this section, we give a formal approach to the notion of edge-contraction, and we derive some basic properties of this notion and give formal proofs of (mostly intuitive) results. Informally, the contraction of an edge $\{u, v\}$ replaces u and v by a new vertex that is adjacent to the neighbours of u and v.

Edge contraction is used in several important graph theoretic investigations. We just mention here the much studied notion of *graph minor* – a graph that can be obtained from a graph by a series of vertex deletions, edge deletions and edge contractions. Well-known is the fundamental work of Robertson and Seymour on graph minors, see e.g. [73].

Even though most of the statements proven in this section are intuitive, a formal proof can sometimes be more technical than expected. We start by formally defining single edge contractions and showing the commutativity of such single contractions. We will work towards the notions of contraction-set and contractions of a graph.

It is easy to intuitively understand the meaning of contracting an edge. In Figure 2.2, we can see an example when we contract an edge e that belongs to a cycle of length three. This example is essential, since it shows that there are two ways of looking at edge contractions. In the first case, the result might

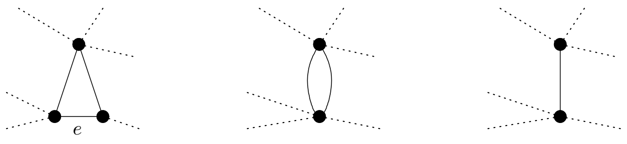

Figure 2.2. Original graph; resulting multigraph after contracting edge e; resulting simple graph after contracting edge e

be a multigraph, since the endpoints of the edge to be contracted might have common neighbours as edge e in Figure 2.2. In the second case, the result is always a simple graph, because parallel edges that might occur will always be replaced by a single edge. It is evident that in that case, contracting an edge can decrease the total number of edges by more than one. These two ways of contracting an edge e in G are sometimes denoted as G/e and $G/\!\!/e$, (e.g. in [73]).

In this thesis, we only consider simple graphs, and we use the notation G/e (as used in [42, 43]) for an edge contraction which results in a simple graph, since we replace parallel edges by a single edge. Therefore, contracting edge $e = \{v_i, v_j\}$ in the graph G, denoted as G/e, is the operation that introduces a new vertex a_e and new edges such that a_e is adjacent to all the vertices in $N(e)$ and delete vertices v_i and v_j and all edges incident to v_i or v_j.

Definition 1. *Let be given a graph $G = (V, E)$. Let be $e \in E$, or let be $e \subseteq V$ with $|e| = 1$. Furthermore, let be $a_e \notin V$. Contracting e in G results in the graph G/e, defined as follows:*

$$\begin{aligned} G/e &:= (V/e, E/e), \text{ where} \\ V/e &= \{a_e\} \cup V \setminus e \\ E/e &= \Big(\bigcup_{f \in E \setminus \{e\}} (f/e)\Big) \setminus \{e\} \\ f/e &= \begin{cases} \{a_e\} \cup f \setminus e & \textit{if } f \cap e \neq \emptyset \\ f & \textit{otherwise} \end{cases} \end{aligned}$$

To be formally consistent, we included in the previous definition the case that e is 'an edge' consisting of a single vertex. This is important, because of previous edge contractions. If e belongs to a set of contracted edges forming a cycle, e can consist of only one vertex, (see Lemma 5).

Lemma 2. *Let a graph $G = (V, E)$ and two distinct edges $e, f \in E$ be given. Let $G_1 = (V_1, E_1) = (G/e)/(f/e)$ and $G_2 = (V_2, E_2) = (G/f)/(e/f)$. Then G_1 and G_2 are isomorphic, i.e. $G_1 \cong G_2$.*

Proof. In case e and f are not adjacent, the lemma is easy to see, since the contractions of e and f do not influence each other. Thus, we assume $e = \{u, v\}$ and $f = \{v, w\}$.

Claim. $V_1 = \{a_f\} \cup V \setminus \{u, v, w\}$.

Proof. This is easy to see:

$$\begin{aligned}(V/e)/(f/e) &= (\{a_e\} \cup V \setminus \{u,v\})/(\{a_e,w\})\\ &= \{a_f\} \cup (\{a_e\} \cup V \setminus \{u,w\}) \setminus \{a_e,w\}\\ &= \{a_f\} \cup V \setminus \{u,v,w\}\end{aligned}$$

◇

In the same way, we obtain: $V_2 = \{a_e\} \cup V \setminus \{u,v,w\}$, and therefore, we see that V_1 and V_2 only differ in the name of one element, namely the only new vertex.

We define a mapping $i : V_1 \leftrightarrow V_2$ in the following way:

$$i(v) = \begin{cases} v & \text{if } v \in V \\ a_e & \text{otherwise, i.e. if } v = a_f \end{cases}$$

Now, we will prove that the mapping i is an isomorphism between G_1 and G_2. From before, we already know that $|V_1| = |V_2|$. What remains to be shown is that two vertices x and y are joined by an edge in G_1 if and only if the corresponding vertices are joined by an edge in G_2.

The next two claims can be deduced from the Definition 1 and the definition of G_1 and G_2. First, we consider the case that the considered vertices x and y are pairwise different from the new vertex. Note that $i(x) = x$ and $i(y) = y$.

Claim. $\{x,y\} \in E_1 \wedge a_f \notin \{x,y\} \iff \{x,y\} \in E_2 \wedge a_e \notin \{x,y\}$

Proof.

$$\begin{aligned}\{x,y\} \in E_1 \wedge a_f \notin \{x,y\} &\iff \{x,y\} \in E/e \wedge \{a_e,w\} \cap \{x,y\} = \emptyset\\ &\iff \{x,y\} \in E \wedge \{u,v,w\} \cap \{x,y\} = \emptyset\\ &\iff \{x,y\} \in E/f \wedge \{u,a_f\} \cap \{x,y\} = \emptyset\\ &\iff \{x,y\} \in E_2 \wedge a_e \notin \{x,y\} = \emptyset\end{aligned}$$

◇

Now, we consider the case that $a_f \in \{x,y\}$. W.l.o.g. let be $x = a_f$. Note that $i(x) = a_e$ and $i(y) = y$.

Claim. $\{a_f,y\} \in E_1 \iff \{a_e,y\} \in E_2$

Proof.

$$\begin{aligned}\{a_f,y\} \in E_1 &\iff \{a_e,y\} \in E/e \vee \{w,y\} \in E/e\\ &\iff \{u,y\} \in E \vee \{v,y\} \in E \vee \{w,y\} \in E\\ &\iff \{u,y\} \in E/f \vee \{a_f,y\} \in E/f\\ &\iff \{a_e,y\} \in E_2\end{aligned}$$

◇

We can summarise this: $\{x,y\} \in E_1 \iff \{i(x),i(y)\} \in E_2$. Therefore, mapping i is an isomorphism and G_1 and G_2 are isomorphic. □

Note that any two permutations of the same elements can be transformed into each other by successively swapping the positions of neighbouring elements. Therefore, with Lemma 2, we can conclude the following corollary.

Corollary 3. *Contracting edges in a graph is commutative.*

Hence, it is sensible to define the contraction of a set of edges. Hereafter, we use the following shorthand:

$$e_1/e_2/e_3/.../e_p := (e_1/e_2)/(e_3/e_2)/.../(((e_p/e_2)/(e_3/e_2))...)$$

Definition 4. *Given a graph G and a set of edges $E' = \{e_1, ..., e_p\}$, we define:*

$$G/E' := G/e_1/.../e_p$$

When contracting a set of edges, the edges to be contracted might be modified due to earlier contractions. The next lemma makes a statement about the edge which is contracted last, if a set of edges is contracted that forms a cycle in the graph.

Lemma 5. *Let $C = (v_1, ..., v_p)$ be a cycle of length $p \geq 3$, and let E' be the set of edges in C, i.e. $E' = \{e_1 = \{v_1, v_2\}, ..., e_p = \{v_p, v_1\}\}$. W.l.o.g. we contract the edges in the following order: $e_1, ..., e_p$. Then e_p will degenerate to a single vertex due to the contractions of $e_1, ..., e_{p-1}$, i.e.:*

$$e_p/e_1/.../e_{p-1} = \{a_{e_{p-1}}\}$$

for a new vertex $a_{e_{p-1}}$.

Proof. We prove this by induction on p. For $p = 3$, we have: $e_1 = \{v_1, v_2\}$, $e_2 = \{v_2, v_3\}$, and $e_3 = \{v_3, v_1\}$.

$$\begin{aligned} e_3/e_1/e_2 &= (e_3/e_1)/(e_2/e_1) \\ &= (\{v_3, v_1\}/\{v_1, v_2\})/(\{v_2, v_3\}/\{v_1, v_2\}) \\ &= \{a_{e_1}, v_3\}/\{a_{e_1}, v_3\} \\ &= \{a_{e_2}\} \end{aligned}$$

Hence, we assume the lemma holds for cycles of length p, and we will show it also holds for cycles of length $p + 1$. Let be given a cycle $e_1, ..., e_{p+1}$ of length $p + 1$. Contracting e_1 introduces a new vertex a_{e_1} and we see that $e_2/e_1 = \{a_{e_1}, v_3\}$ and $e_{p+1}/e_1 = \{a_{e_1}, v_{p+1}\}$. All other edges are not influenced by contracting e_1. Hence, this results in a cycle of length p, for which we know that the lemma holds. □

Definition 1 defines edge contractions. For technical reasons, it also defines the contraction of 'an edge' consisting of a single vertex. However, such a contraction in G results in a graph isomorphic to G.

Lemma 6. *Let a graph $G = (V, E)$ and $x \in V$ be given. Furthermore, let $G' = (V', E') = G/x$. Then we have: $G' \cong G$.*

Proof. Looking at Definition 1, we see that $V' = \{a_x\} \cup V \setminus x$. We do not delete any edges from E to obtain E', but we update all single edges which contained x to contain a_x. Therefore, G' is isomorphic to G, since we only changed the name of vertex x into a_x. □

Lemma 7. *Let a graph $G = (V, E)$, a set of edges $E_1 = \{h_1, ..., h_q\}$ and a set of edges $E_2 = \{e_1, ..., e_p\}$, with $E_1 \cap E_2 = \emptyset$ be given. Let E_2 form a cycle $e_1, ..., e_p$ with $3 \leq p$. Let $E' = \{h_1, ..., h_q, e_1, ..., e_p\}$ and $E'' = \{h_1, ..., h_q, e_1, ..., e_{p-1}\}$. Then G/E' is isomorphic to G/E'', i.e. $G/E' \cong G/E''$.*

Proof. From Corollary 3, we know that edge contractions are commutative. Therefore, we choose to contract edges in the following order: $e_1, ..., e_p, h_1, ..., h_q$. Then we have:

$$\begin{aligned} G/E' &= G/e_1/.../e_{p-1}/e_p/h_1/.../h_q \\ &\quad \text{(from Lemma 5 follows:)} \\ &= G/e_1/.../e_{p-1}/\{a_p\}/h_1/.../h_q \\ &\quad \text{(from Lemma 6 follows:)} \\ &\cong G/e_1/.../e_{p-1}/h_1/.../h_q \\ &\cong G/E'' \end{aligned}$$

□

From the lemma, we easily conclude that we can delete an arbitrary edge in a cycle in a set of edges $E' \subseteq E$ to be contracted, for a graph $G = (V, E)$. We can repeat this until there are no cycles left. The result will be a maximal spanning forest E'' of $G[E']$, and we have G/E' is isomorphic to G/E'', i.e. $G/E' \cong G/E''$. ($G[E']$ denotes the subgraph of G induced by the edge-set E', see Section 2.3 for more details.) Since we can restrict ourself to edge sets without cycles, we use the term *contraction-set* to refer to such sets.

Definition 8. *A* contraction-set *E' in $G = (V, E)$ is a set of edges $E' \subseteq E$, such that $G[E']$ is a forest. A* contraction *H of G is a graph such that there exists a contraction-set E' with: $H = G/E'$.*

After the previous observations, we develop another view on contracting a set E' of edges. The graph $G[E']$ is composed of connected components $O_1, ..., O_z$, with $O_i = (V_i, E_i)$. When contracting E' in G, then every connected component O_i will be replaced by a new single vertex a_i. This vertex a_i will be made adjacent to every vertex in $N_G(V_i)$, i.e. all vertices that are neighbours in G of a vertex in O_i, but that do not belong to V_i. It becomes evident from the previous lemmas that this definition of edge contractions is equivalent to Definition 1.

Note once again that after each single edge-contraction the names of the vertices are updated in the graph. Hence, for two adjacent edges $e = \{u, v\}$ and $f = \{v, w\}$, edge f will be different after contracting edge e, namely in G/e we have $f = \{a_e, w\}$. However, it might be convenient to use f to represents the same edge in G and in G/e. The same applies also to vertices.

The next lemma tells us that an edge contraction might decrease the degree of a vertex, but it can never decrease it by more than one.

Lemma 9. *Let be given a graph $G = (V, E)$, $v \in V$ and $e \in E$.*

$$\begin{aligned} v \notin e &\Longrightarrow d_{G/e}(v) \geq d_G(v) - 1 \\ v \in e &\Longrightarrow d_{G/e}(a_e) \geq d_G(v) - 1 \end{aligned}$$

Proof. We prove this by considering an exhaustive case distinction.
Case 1 '$e = \{u, v\} \wedge v \in e$': Clearly, we have:

$$N_{G/e}[v] = N_G[v] \cup N_G[u] \cup \{a_e\} \setminus \{u, v\}$$

And therefore it holds that:

$$\begin{aligned} d_{G/e}(v) &= |N_{G/e}(v)| = |N_{G/e}[v]| - 1 \\ &= |N_G[v] \cup N_G[u] \cup \{a_e\} \setminus \{u,v\}| - 1 \\ &\geq |N_G[v] \cup \{a_e\} \setminus \{u,v\}| - 1 \\ &\geq |N_G[v]| + 1 - 2 - 1 = |N_G[v]| - 2 = |N_G(v)| - 1 = d_G(v) - 1 \end{aligned}$$

Case 2 ‘$e = \{u,w\} \wedge |e \cap N(v)| = 2$’: We have:

$$N_{G/e}(v) = N_G(v) \cup \{a_e\} \setminus \{u,w\}$$

and thus:

$$\begin{aligned} d_{G/e}(v) &= |N_{G/e}(v)| = |N_G(v) \cup \{a_e\} \setminus \{u,w\}| \\ &= |N_G(v)| + 1 - 2 = |N_G(v)| - 1 = d_G(v) - 1 \end{aligned}$$

Case 3 ‘$e = \{u,w\} \wedge |e \cap N(v)| \leq 1$’: In this case, the neighbourhood of v is not affected, apart from a possible change of the name of one vertex in $N(v)$. Therefore, $d_{G/e}(v) = d_G(v)$. □

2.3 Subgraphs and Minors

When deleting vertices of a graph $G = (V, E)$ and their incident edges, we get an *induced subgraph*. For $V' \subseteq V$, the subgraph induced by V' is denoted by $G[V']$, and it holds that $G[V'] = (V', E')$ with $E' = E \cap (V' \times V')$. A less standard notion is $G[E']$ – the graph induced by an edge set $E' \subseteq E$. It holds that $G[E'] = (V', E')$, with $V' = \bigcup_{e \in E'} e$. A *subgraph* is obtained, if we additionally allow the deletion of edges, i.e. $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. We use $G' \subseteq G$ to denote that G' is a subgraph of G. If we furthermore allow edge-contractions (see Section 2.2 for more details), we get a *minor*. Following the notations of [42, 43], we write $G' \preceq G$, if G' is a minor of G. We explicitely exclude the null graph (the empty graph on 0 vertices), as a subgraph or minor of a graph. The next definition summarises the paragraph above.

Definition 10. *Let $G = (V, E)$ be a graph.*

- *G' is a (vertex-set) induced subgraph of G, if there is a $V' \subseteq V$ with $|V'| \geq 1$, such that $G' = G[V'] = (V', \{\{v,w\} \in E \mid v, w \in V'\})$.*
- *G' is an edge-set induced subgraph of G, if there is a $E' \subseteq E$ with $E' \geq 1$, such that $G' = G[E'] = (\bigcup_{e \in E'} e, E')$.*
- *$G' = (V', E')$ is a subgraph of G, if $V' \subseteq V$, $|V'| \geq 1$ and $E' \subseteq E$.*
- *G' is a minor of G, if there exists a subgraph H of G and a contraction-set $F \subseteq E(H)$, such that $G' = H/F$.*

We consider again the graph G shown in Figure 2.1. Figure 2.3 depicts some example subgraphs and minors of G, respectively. On the very left, we see a subgraph induced by the vertex-set $\{b, c, e, i\}$. This induced subgraph consists of two connected components. The graph right next to it is a subgraph of G, induced by the edge-set $\{\{b,c\}, \{b,e\}, \{d,h\}\}$. A subgraph which is not induced, is the graph second from right. On the very right, we see a minor of G, where at least one edge (one of the edges $\{e,f\}$, $\{f,g\}$ or $\{g,j\}$) has been contracted.

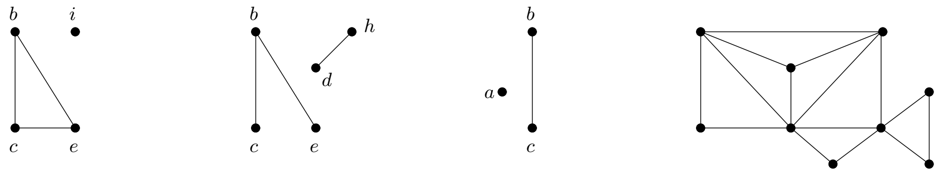

Figure 2.3. Four graphs (three of which are decomposed into two connected components) as subgraphs or minors of the graph G in Figure 2.1

2.4 Treewidth

The notions tree-decomposition and treewidth were introduced by Robertson and Seymour, see e.g. [79, 80].

Definition 11. *A* tree-decomposition *of a graph $G = (V, E)$ is a pair (T, X) with $T = (I, F)$ a tree, and $X = \{X_i \mid i \in I\}$ a family of subsets of V, one for each node of T, such that*

- $\bigcup_{i \in I} X_i = V$.
- *for all edges $\{v, w\} \in E$ there exists an $i \in I$ with $\{v, w\} \subseteq X_i$.*
- *for all $i, j, k \in I$: if j is on the path in T from i to k, then $X_i \cap X_k \subseteq X_j$.*

The width *of a tree-decomposition $((I, F), \{X_i \mid i \in I\})$ is $\max_{i \in I} |X_i| - 1$. The* treewidth $tw(G)$ *of a graph G is the minimum width over all tree-decompositions of G.*

Once more, we consider graph G given in Figure 2.1. Figure 2.4 shows a tree-decomposition of G of width 3. There is no tree-decomposition of G with smaller width, and thus, $tw(G) = 3$. A tree-decomposition (T, X) is *minimal* if for all $v \in V$, the deletion of v from a subset $X_i \in X$ violates at least one of the conditions in Definition 11.

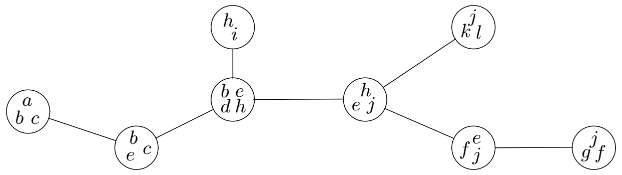

Figure 2.4. A tree-decomposition of width 3 of the graph G in Figure 2.1

For more information on treewidth and how tree-decompositions can be used in the design of bottom-up or dynamic-programming algorithms to solve problems on graphs of bounded treewidth, see e.g. [8] or [10]. Such methods often use nice tree-decompositions.

Definition 12. *A tree-decomposition (T, X) is* nice*, if T is rooted and binary, and the nodes are of four types:*

- Leaf nodes *i are leaves of T and have $|X_i| = 1$.*
- Introduce nodes *i have one child j with $X_i = X_j \cup \{v\}$ for some vertex $v \in V$.*
- Forget nodes *i have one child j with $X_i = X_j \setminus \{v\}$ for some vertex $v \in V$.*

- Join nodes i *have two children* j_1, j_2 *with* $X_i = X_{j_1} = X_{j_2}$

A tree-decomposition can be converted into a nice tree-decomposition with $|I| = O(V|G|)$ of the same width in linear time (see [61, 10]). The properties of a nice tree-decomposition will in general not provide more algorithmic power. However, designing algorithms is often easier when using a nice tree-decomposition.

Unless otherwise stated, $T = (I, F)$ is a nice tree-decomposition of the graph G. The term 'vertex' refers to a vertex of the graph and the term 'node' refers to a vertex of a tree (-decomposition). Popular words for referring to a node of a tree-decomposition are also 'supernode' or 'bag', since they represent a subset of vertices of the original graph.

The next lemma is a well-known result and an important fact for proving the parameters, considered in Chapter 3, to be treewidth lower bounds.

Lemma 13 (see e.g. [11]). *If* G' *is a minor of* G*, then* $tw(G') \leq tw(G)$.

2.5 Subdividing an Edge

Subdividing an edge $e = \{u, w\}$ means putting a new vertex v_e on the edge e. Formally, it is an operation that introduces a new vertex v_e and replaces the edge $e = \{u, w\}$ by the two edges $\{u, v_e\}$ and $\{v_e, w\}$ (see Figure 2.5).

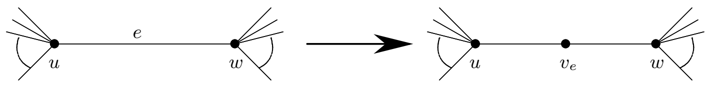

Figure 2.5. Subdividing edge e

When doing this for all edges of a graph, we clearly increase the size of it. Let $G = (V, E)$ be the original graph, and $G' = (V', E')$ be the graph after subdividing each edge of G. Then we have: $|V'| = |V| + |E|$ and $|E'| = 2 \cdot |E|$. For arbitrary graphs, it holds that $|E| \leq \frac{|V| \cdot (|V|-1)}{2}$ which would mean a potential quadratic increment.

For graphs of bounded treewidth, however, the increment is only linear, because due to their special structure, we have for such graphs that $|E| \leq k \cdot |V| - \frac{k \cdot (k+1)}{2}$ (see [9]). Hence, the size of the vertex set $|V'|$ is linear in $|V|$, if k is considered to be a constant. Subdivisions have no effect on the treewidth of a simple graph. For this, consider an edge $e = \{u, w\}$. In our tree-decomposition, there is one node i with $\{u, w\} \subseteq X_i$. When placing vertex v_e on this edge, we can simply attach a new node $\{u, v_e, w\}$ to the node i. It is easy to see that this results in a proper tree-decomposition and does not increase its width, if the width is at least 2. Also, if the graph has treewidth 1, it is a forest, and the graph resulting from subdividing a forest is still a forest and hence, also has treewidth 1. If the graph has treewidth 0, then there are no edges that could be subdivided in the graph. Hence, subdivisions do not affect the treewidth of a graph.

3

Treewidth Lower Bounds: Methods, Relations and Complexity

This chapter focuses on lower bounds on the treewidth of a graph. As mentioned at the beginning of this thesis, good lower bounds can serve to speed up branch-and-bound methods, inform us about the quality of upper bound heuristics, and in some cases, tell us that we should not use tree-decompositions to solve a problem on a certain instance. A large lower bound on the treewidth of a graph implies that we should not hope for computationally efficient dynamic-programming algorithms that use tree-decompositions for this particular instance.

Every lower bound method for treewidth can be extended by taking the maximum of the lower bound over all subgraphs or minors of a graph. In this chapter, we study this extension from a theoretical point of view. We look at treewidth lower bounds (Section 3.1), obtained by combining the minimum degree with taking subgraphs and minors. We also apply this idea to two other parameters that are related to the minimum degree. A combination of taking minors and a treewidth lower bound based on Maximum Cardinality Search (see [70]) is considered in Section 3.2. These investigations, based on the idea to incorporate edge-contraction into existing treewidth lower bounds, result in a number of graph parameters, providing new lower bounds for treewidth. We show relations between these graph parameters and study their computational complexity. The basic concept of another general method to extend and improve treewidth lower bounds is described in Section 3.3. This method was introduced in [32] and is based on so-called 'improved graphs'. The content of this chapter is based on cooperations with Hans L. Bodlaender and Arie M. C. A. Koster, see [17, 18, 65, 66, 99].

3.1 Contraction Degeneracy and Related Lower Bounds

The minimum degree of a graph is known to be a treewidth lower bound. Taking the minimum degree over all subgraphs of a graph can improve this lower bound. Going one step further, we can take the minimum degree over all minors. (This results in the parameter 'contraction degeneracy'.) This section is devoted to a number of parameters that are related to the minimum degree and their combinations with subgraphs or minors.

In [6], the degeneracy of a graph is defined as follows:

Definition 14 (see [6]). *The* degeneracy $s(G)$ *of a graph G is the minimum number s such that G can be reduced to an empty graph by the successive deletion of vertices with degree at most s.*

We now define the parameter δD of a graph G which is the maximum over all subgraphs of G of the minimum degree of the subgraph, and we show that $s(G) = \delta D(G)$.

$$\delta D(G) := \max_{G'}\{\delta(G') \mid G' \subseteq G \wedge n(G') \geq 1\}$$

Lemma 15 (folklore). *For any graph G, it holds that:* $s(G) = \delta D(G)$.

Proof. In each $G' \subseteq G$, there is a vertex with degree at most $\delta D(G)$. Therefore, we can reduce G to an empty graph by successively deleting a vertex of degree at most $\delta D(G)$. Hence, $s(G) \leq \delta D(G)$.

On the other hand, let $G' \subseteq G$ be such that $\delta(G') = \delta D(G)$. Now, consider a sequence S of vertex deletions, such that the deleted vertices have degree at most $s(G)$. Let $v \in V(G')$ be the first vertex that is deleted in S. At this moment of deletion, v has degree at least $d_{G'}(v) \geq \delta(G')$, since v is the first vertex deleted in G'. Furthermore, we know that at the moment of deleting v, we have that the degree of v is at most $s(G)$. Hence, altogether, we have that $s(G) \geq d_{G'}(v) \geq \delta(G') = \delta D(G)$. □

In the following, we use $\delta D(G)$ to denote the degeneracy of a graph. It is interesting that Definition 14 suggests an algorithm to compute the degeneracy: Successively delete a vertex of minimum degree and return the maximum of the encountered minimum degrees. This simple algorithm has been used to compute the degeneracy as a treewidth lower bounds e.g. in [63]. Later in this section and in Chapter 4, we will develop and examine more treewidth lower bounds based on this simple observation.

One key idea for our investigations is based on the fact that the treewidth does not increase when taking minors (see Lemma 13). Looking at the method described above to compute the degeneracy, we also can contract a vertex of minimum degree to one of its neighbours, instead of deleting it. In such a 'degeneracy algorithm' with contractions instead of deletions, we can get different values if we make different choices of which minimum degree vertex to select, and which neighbour to contract it with. The optimal way of doing these contractions is captured by the notion of *contraction degeneracy*. This approach results in the variants of the parameters involving edge contraction.

3.1.1 Definition of the Parameters

We define a number of graph parameters in this section, each of these being a lower bound on the treewidth of a graph, cf. Section 3.1.2. For the sake of completeness, we repeat the definitions of δ and δD. For the *minimum degree* δ of a graph G, we have:

$$\delta(G) := \min_{v \in V} d(v)$$

The *δ-degeneracy* or simply the *degeneracy* δD of a graph G is defined as follows (see also the beginning of Section 3.1 and Definition 14):

$$\delta D(G) := \max_{G'} \{\delta(G') \mid G' \subseteq G \wedge n(G') \geq 1\}$$

The *δ-contraction degeneracy* or simply the *contraction degeneracy* δC of a graph G was first defined in [17]. It is defined as the maximum over all minors G' of G of the minimum degree:

$$\delta C(G) := \max_{G'} \{\delta(G') \mid G' \preceq G \wedge n(G') \geq 1\}$$

Let an ordering $v_1, ..., v_n$ of the vertices of G be given with $n \geq 2$, such that $d(v_i) \leq d(v_{i+1})$, for all $i \in \{1, ..., n-1\}$. The *second smallest degree* δ_2 of a graph G is defined as:

$$\delta_2(G) := d(v_2)$$

Note that it is possible that $\delta(G) = \delta_2(G)$. Similar to the δ-degeneracy and δ-contraction-degeneracy we define the δ_2-degeneracy and δ_2-contraction-degeneracy. The *δ_2-degeneracy* $\delta_2 D$ of a graph $G = (V, E)$ with $n \geq 2$ is defined as follows:

$$\delta_2D(G) := \max_{G'}\{\delta_2(G') \mid G' \subseteq G \wedge n(G') \geq 2\}$$

The δ_2*-contraction degeneracy* δ_2C of a graph $G = (V, E)$ with $n \geq 2$ is defined as:

$$\delta_2C(G) := \max_{G'}\{\delta_2(G') \mid G' \preceq G \wedge n(G') \geq 2\}$$

In [76, 77], Ramachandramurthi introduced the parameter $\gamma_R(G)$ of a graph G and proved that this is a lower bound on the treewidth of G. Here $\gamma_R(G)$ is defined as:

$$\gamma_R(G) := \min(n-1, \min_{v,w\in V, v\neq w, \{v,w\}\notin E} \max(d(v), d(w)))$$

Note that $\gamma_R(G) = n-1$ if and only if G is a complete graph on n vertices. Furthermore, note that $\gamma_R(G)$ is determined by a pair $\{v, w\} \notin E$ with $\max(d(v), d(w))$ as small as possible. For such a pair, we then say that $\{v, w\}$ is a *nonedge determining* $\gamma_R(G)$. If $d(v) \leq d(w)$ then we say that w is a *vertex determining* $\gamma_R(G)$. Once again, we define the 'degeneracy' and 'contraction degeneracy' versions also for the parameter γ_R. The γ_R*-degeneracy* γ_RD of a graph $G = (V, E)$ with $n \geq 2$ is defined as follows:

$$\gamma_RD(G) := \max_{G'}\{\gamma_R(G') \mid G' \subseteq G \wedge n(G') \geq 2\}$$

The γ_R*-contraction degeneracy* γ_RC of a graph $G = (V, E)$ with $n \geq 2$ is defined as:

$$\gamma_RC(G) := \max_{G'}\{\gamma_R(G') \mid G' \preceq G \wedge n(G') \geq 2\}$$

Note that the contraction variants $\delta C(G)$, $\delta_2C(G)$ and $\gamma_RC(G)$ were all defined as the maximum over all minors $G' \preceq G$ of $\delta(G)$, $\delta_2(G)$ and $\gamma_R(G)$, respectively. Using contractions instead of minors in these definitions does not lead to an equivalent notion, unless the considered graph G is connected. If G is not connected, it might be necessary to delete one or more connected components, to obtain a minor G' of G with maximum minimum degree.

3.1.2 Relationships Between the Parameters

In this section, we examine the relationships between ten graph parameters: the treewidth of a graph and the nine parameters which were defined in the previous section.

Lemma 16. *For any graph $G = (V, E)$ with $|V| \geq 2$, it holds that:*

$$\delta(G) \leq \delta_2(G) \leq \gamma_R(G) \leq tw(G)$$

Proof. The first two inequalities follow directly from the definitions of the corresponding parameters. The last one was proven by Ramachandramurthi in [76]. □

Note that $\delta(G)$ is known to be a treewidth lower bound for a long time (see e.g. [87, 11, 63]). One possible proof uses an argument based on minimal tree-decompositions (T, X) of width $tw(G)$. Let i be a leaf node of T, and let j be the neighbour of i. By minimalism of (T, X), there is a vertex $v \in X_i \setminus X_j$. Therefore, all the neighbours of v are contained in X_i, and hence we have that $\delta(G) \leq d(v) \leq |X_i| - 1 \leq tw(G)$. With a similar argument, we can also prove $\delta_2(G)$ to be a treewidth lower bound.

Lemma 17. *For any graph G and $x \in \{\delta, \delta_2, \gamma_R\}$, it holds that:*

$$x(G) \leq xD(G) \leq xC(G) \leq tw(G)$$

Proof. Note that G is a subgraph of G, and any subgraph of G is also a minor of G. Therefore, the first two inequalities are trivial. Furthermore, taking minors does not increase the treewidth (see Lemma 13). We also have that $x(G') \leq tw(G')$ (which follows from Lemma 16) for G' a minor of G. Hence, it follows that $x(G') \leq tw(G') \leq tw(G)$ for any minor G' of G, and therefore $xC(G) \leq tw(G)$. □

Lemma 18. *For any graph $G = (V, E)$ with $|V| \geq 2$ and $X \in \{D, C\}$, it holds that:*

$$\delta X(G) \leq \delta_2 X(G) \leq \gamma_R X(G) \leq tw(G)$$

Proof. Lemma 16 holds for every subgraph or minor G' of G, unless G' has only one vertex. However, in that case the minimum degree is zero, and since G has at least two vertices, it is obvious that $\delta_2 X(G) \geq 0$ and $\gamma_R X(G) \geq 0$. Therefore, the first two inequalities follow. Note that $\gamma_R X(G) \leq \gamma_R C(G) \leq tw(G)$ (see Lemma 17), and hence $\gamma_R X(G) \leq tw(G)$. □

An alternate argument proving $\delta_2 C(G)$ to be a treewidth lower bound is given in the proof of Lemma 32 in Section 3.1.5.

Lemma 19. *For any graph $G = (V, E)$ with $|V| \geq 2$ and $X \in \{D, C\}$, it holds that:*

$$\delta_2 X(G) \leq \delta X(G) + 1$$

Proof. Let $G' = (V', E')$ be a subgraph (if $X = D$) or minor (if $X = C$) of G with $\delta_2(G') = \delta_2 X(G)$, and let v_1 and v_2 be vertices in G' with smallest and second smallest degree in G', respectively, i.e. $d_{G'}(v_1) = \delta(G')$ and $d_{G'}(v_2) = \delta_2(G')$. We consider the graph $G'' := G'[V' \setminus \{v_1\}]$. Note that G'' is also a subgraph ($X = D$) or minor ($X = C$) of G. It is clear that we have:

$$\forall v \in V(G'') \ : \ d_{G'}(v) - 1 \leq d_{G''}(v) \leq d_{G'}(v)$$

Let $w \in V(G'')$ be a vertex with minimum degree in G'', i.e. $\delta(G'') = d_{G''}(w)$. By the definition of v_2, it holds that:

$$\delta_2 X(G) = d_{G'}(v_2) \leq d_{G'}(w)$$

Otherwise v_2 was not a vertex of second smallest degree in G'. Altogether, we have:

$$\delta_2 X(G) - 1 = d_{G'}(v_2) - 1 \leq d_{G'}(w) - 1 \leq d_{G''}(w) = \delta(G'') \leq \delta X(G)$$

□

The next lemma shows some interesting properties of the parameter γ_R, when given a vertex sequence sorted according to nondecreasing degree. It is needed in the proof of a subsequent lemma.

Lemma 20. *Let be given a graph G on n vertices with $G \neq K_n$. Furthermore, let an ordering $v_1, ..., v_n$ of $V(G)$ be given, such that $d(v_i) \leq d(v_{i+1})$, for all $i \in \{1, ..., n-1\}$. We define $j := \min\{i \in \{1, ..., n\} \mid \exists l \in \{1, ..., i-1\} \ : \ \{v_i, v_l\} \notin E(G)\}$. Then we have:*

1. $d(v_j) = \gamma_R(G)$
2. $v_1, ..., v_{j-1}$ *form a clique in G*

Proof. (1.) Since $d(v_l) \leq d(v_j)$ and $\{v_i, v_l\} \notin E(G)$, we clearly have $\max\{d(v_l), d(v_j)\} = d(v_j) \geq \gamma_R(G)$, and because there is no $v_{j'}$ with $d(v_{j'}) \leq d(v_j)$ and $j' < j$, such that there is a $v_{l'} \in \{v_1, ..., v_{j'} - 1\}$, with $\{v_{l'}, v_{j'}\} \notin E(G)$, the equality follows.

(2.) This follows because if v_j is the left-most vertex in the given sequence that is not adjacent to all the vertices $v_1, ..., v_{j-1}$ to its left, then the vertices $v_1, ..., v_{j-1}$ must form a clique. □

Lemma 21. *For any graph $G = (V, E)$ with $|V| \geq 2$ and $X \in \{D, C\}$, it holds that:*

$$\gamma_R X(G) \leq 2 \cdot \delta_2 X(G)$$

Proof. Let $G' = (V', E')$ be a minimal subgraph (in case $X = D$) or a minimal minor (in case $X = C$) of G with $\gamma_R(G') = \gamma_R X(G)$, i.e. there is no subgraph or minor G^* of G' with $\gamma_R(G^*) \geq \gamma_R(G')$. If G' is a complete graph on $n' := |V'|$ vertices, then the lemma follows easily. For technical reasons, we assume in the following w.l.o.g. that $G' \neq K_{n'}$. Let an ordering $v_1, ..., v_{n'}$ of V' be given, such that $d_{G'}(v_i) \leq d_{G'}(v_{i+1})$, for all $i \in \{1, ..., n' - 1\}$. We define $j := \min\{i \in \{1, ..., n'\} \mid \exists l \in \{1, ..., i-1\} : \{v_i, v_l\} \notin E'\}$. From Lemma 20, we know that $d_{G'}(v_j) = \gamma_R(G')$ and that $v_1, ..., v_{j-1}$ form a clique in G'. Thus, these vertices have degree at least $j - 2$. To show the lemma, we need a slightly higher degree as stated by the following claim.

Claim. $\delta_2(G') = d_{G'}(v_2) \geq j - 1$.

Proof. Assume the opposite, namely that $d_{G'}(v_1) = d_{G'}(v_2) = j - 2$. Hence, v_1 and v_2 are only adjacent to the vertices in the clique formed by $v_1, ..., v_{j-1}$, and therefore, both v_1 and v_2 are not adjacent to v_j. Note that $\max\{d_{G'}(v_1), d_{G'}(v_j)\} = \max\{d_{G'}(v_2), d_{G'}(v_j)\} = \gamma_R(G')$. Now, we consider $G'[V' \setminus \{v_1\}]$. The deletion of v_1 decreases the degree of the vertices $v_2, ..., v_{j-1}$ by one. However, $v_2, ..., v_{j-1}$ still form a clique in $G'[V' \setminus \{v_1\}]$. Furthermore, note that due to the deletion of v_1, we only deleted nonedges $\{v_1, v_i\}$, i.e. pairs of vertices $\{v_1, v_i\} \notin E'$. Deleting elements of a set over which a minimum is taken can never decrease the value of that minimum. Therefore, we can conclude that $\{v_2, v_j\}$ is a nonedge in $G'[V' \setminus \{v_1\}]$, determining $\gamma_R(G'[V' \setminus \{v_1\}])$. Since the degree of v_j did not change when deleting v_1, we have that $\max\{d_{G'[V' \setminus \{v_1\}]}(v_2), d_{G'[V' \setminus \{v_1\}]}(v_j)\} = \gamma_R(G'[V' \setminus \{v_1\}]) = \max\{d_{G'}(v_2), d_{G'}(v_j)\} = \gamma_R(G')$. Hence, $\gamma_R(G'[V' \setminus \{v_1\}]) \geq \gamma_R(G')$, which contradicts the choice of G'. ◇

Hence, we have that $j - 1 \leq d_{G'}(v_j) = \gamma_R X(G)$. Note that the following holds:

$$\delta_2 X(G) \geq \delta_2 X(G') \geq \delta_2(G') = d_{G'}(v_2) \geq j - 1$$

We consider now the graph $G'' := G'[V' \setminus \{v_1, ..., v_{j-1}\}]$, which is also a subgraph ($X = D$) or minor ($X = C$) of G. It is clear that deleting $j - 1$ vertices in a graph can decrease the degree of any vertex at most by $j - 1$. Therefore, we have:

$$\delta_2 X(G) \geq \delta X(G) \geq \delta(G'') \geq d_{G'}(v_j) - (j - 1)$$

Hence, altogether, we have:

$$\begin{aligned} \delta_2 X(G) &\geq \max(j - 1, d_{G'}(v_j) - (j - 1)) \\ &\geq \frac{d_{G'}(v_j)}{2} = \frac{\gamma_R(G')}{2} = \frac{\gamma_R X(G)}{2} \end{aligned}$$

□

It follows directly from Lemma 16, Lemma 17 and Lemma 18 that all the parameters defined in Section 2 are lower bounds for treewidth. Furthermore, we see that the gap between the parameters δD and $\delta_2 D$, and between δC and $\delta_2 C$ can be at most one (see Lemma 19). In Section 4.2.3, we will see that $\delta_2 D$ can be computed in polynomial time. Therefore, Lemma 21 gives us a 2-approximation algorithm for computing the parameter $\gamma_R D$, which is NP-hard to compute, see Section 3.1.3.

3.1.3 NP-completeness Results

In this section, we formulate decision problems for some of the parameters defined in Section 3.1.1, and we prove that these problems are NP-complete.

Complexity of Computing δC

The decision problem corresponding to δC is formulated as follows:

Problem: CONTRACTION DEGENERACY
Instance: Graph $G = (V, E)$ and integer $k \geq 0$.
Question: Is the contraction degeneracy of G at least k?

Theorem 22. *The* CONTRACTION DEGENERACY *problem is NP-complete, even for bipartite graphs.*

Proof. Clearly, the problem is in NP as we only have to guess an edge set E', and then compute in polynomial time $\delta(G/E')$.

The hardness proof is a transformation from the VERTEX COVER problem, which is known to be NP-complete, see [49]. In the VERTEX COVER problem, we are given a graph $G = (V, E)$ and an integer k, and look for a vertex cover of size at most k, i.e. a set $W \subseteq V$ with $|W| \leq k$, such that each edge in E has at least one endpoint in W. Let a VERTEX COVER instance (G, k) be given, with $G = (V, E)$.

Construction:

We build a graph in two steps. In the first step, we construct a graph G' by taking the complement $\bar{G}$ of G, two adjacent vertices and k pairwise non-adjacent vertices, and making the new vertices adjacent to each vertex in G. G' is formally defined as follows, see Figure 3.1:

$$\begin{aligned}
G' &:= (V', E') \text{ where} \\
V' &= V \cup \{w_1, w_2\} \cup \{u_1, \dots, u_k\} \\
E' &= (\{\{v, w\} \notin E \mid v, w \in V, v \neq w\}) \cup \{\ \{w_1, w_2\}\ \} \\
&\quad \cup \{\ \{w_i, v\} \mid i \in \{1, 2\} \wedge v \in V\ \} \\
&\quad \cup \{\ \{u_i, v\} \mid i \in \{1, \dots, k\} \wedge v \in V\ \}
\end{aligned}$$

The final graph G^* in our construction is obtained by subdividing any edge in G', i.e. replacing each edge in G' by a path with two edges.

$$\begin{aligned}
G^* &:= (V^*, E^*) \text{ where} \\
V^* &= V' \cup \{\ v_e \mid e \in E'\ \} \\
E^* &= \{\ \{u, v_e\}, \{v_e, w\} \mid e = \{u, w\} \in E'\ \}
\end{aligned}$$

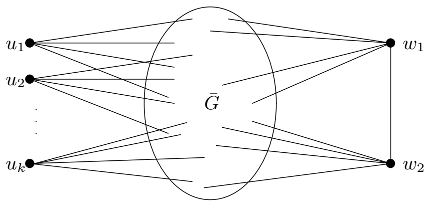

Figure 3.1. Graph G' constructed in the proof of Theorem 22

Let $n = |V|$. The constructed instance of the CONTRACTION DEGENERACY problem is $(G^*, n+1)$. G^* is a bipartite graph, as all edges in G^* are between a vertex in V' and a vertex in $\{ v_e \mid e \in E' \}$. Now, we have to show that there is a vertex cover for G of size at most k if, and only if $\delta C(G^*) \geq n{+}1$.

Claim. If there is a vertex cover of G of size at most k, then there is a $E_3 \subseteq E^*$, such that $\delta(G^*/E_3) \geq n+1$.

Proof: Suppose there is a vertex cover of size at most k. Now take a vertex cover $V_1 = \{v_1, \ldots, v_k\}$ of G of size exactly k. (If we add vertices to a vertex cover, we obtain again a vertex cover.) We build the set of edges to be contracted in two steps. Let E_1 be the edge set, such that $G^*/E_1 = G'$, i.e. E_1 consists of $|E'|$ edges to undo the subdivisions. First, we contract the edges in E_1 and obtain G^*/E_1. Then, in G^*/E_1, we contract edge set E_2 defined as follows: $E_2 := \{ \{u_i, v_i\} \mid i = 1, \ldots, k\}$, i.e. each vertex in the vertex cover has a vertex of the type u_i contracted to it. We write $E_3 := E_1 \cup E_2$. We claim that the resulting graph $G_3 := G^*/E_3$ is an $(n+2)$-clique. Assume there are two vertices x and y with $\{x, y\} \notin E(G_3)$. Since the vertices w_1 and w_2 are universal in G_3, we have $\{x, y\} \subseteq V$. Therefore, $\{x, y\} \in E$, and hence x or y is in V_1. We assume w.l.o.g. $x \in V_1$, i.e. $\exists i \in \{1, \ldots, k\}$, such that $v_i = x$. Since we contracted $\{u_i, v_i\}$ and $\{u_i, y\} \in E(G')$, we have $v_i = x$ is adjacent to y in G_3, which is a contradiction. Hence, $G_3 = G^*/E_3$ is an $(n+2)$-clique and $\delta(G^*/E_3) = n+1$. $\diamond$

Claim. If there is a $E_3 \subseteq E^*$, such that $\delta(G^*/E_3) \geq n+1$, then there is a vertex cover V_1 for G of size at most k.

Proof: We have $|E'|$ vertices $V_{E'} := \{ v_e \mid e \in E' \}$ of degree two in G^*, namely the subdivisions. Assuming G has more than just one vertex, i.e. assuming $n \geq 2$, we see that all vertices in $V_{E'}$ have a degree in G^* that is too small, and hence, must be contracted in order to get a larger degree. Hence, there is a $E_2 \subseteq E_3$, such that $G^*/E_2 = G'$. Let be $E_1 := E_3 \setminus E_2$. Because of the commutativity of contraction-operations, we assume that we first contract all edges in E_2. A vertex $u_i, \forall i \in \{1, \ldots, k\}$ has degree exactly n in G'. Thus, for each $u_i, i \in \{1, \ldots, k\}$, we have to contract an edge incident to u_i. After contracting these edges, there are $n{+}2$ vertices left in the graph. Therefore we cannot contract another edge, since then we could not obtain the minimum degree of $n+1$. Furthermore, we see that G^*/E_3 is an $(n+2)$-clique. Hence, E_1 contains exactly k edges, one for every $u_i, i \in \{1, \ldots, k\}$, with the other endpoint in V. Let be $V_1 := \bigcup_{e \in E_1} e \ \setminus \ \bigcup_{i=1,\ldots k} u_i$. Clearly, $|V_1| = k$, and we claim that V_1 is a vertex cover of G. Assume, there is an edge $f = \{x, y\}$ in G with $V_1 \cap f = \emptyset$. Hence, f is not an edge in G'. Since G^*/E_3 is an $(n+2)$-clique, edge f exists in G^*/E_3, which means: f was created by contracting another edge $\{u_i, v_j\} \in E_1$. This can only be the case if $v_j = x$ or $v_j = y$. According to the definition of V_1, we have: $v_j \in V_1$, which contradicts $V_1 \cap f = \emptyset$. Hence, V_1 is a vertex cover of size k. $\diamond$

As G^* can be constructed in polynomial time, the NP-completeness of the CONTRACTION DEGENERACY problem now follows. $\square$

Complexity of Computing $\gamma_R D$

In this section, we formulate the decision problem corresponding to the parameter $\gamma_R D$, and we show its NP-completeness. We will give a proof that is similar to the proof of the NP-completeness of CONTRACTION DEGENERACY in the previous section. However, before that, we show that only considering induced subgraphs when computing $\gamma_R D$ is sufficient.

Lemma 23. *For all graphs G, there exists an induced subgraph G' of G such that $\gamma_R(G') = \gamma_R D(G)$.*

Proof. Let a subgraph $G' = (V', E')$ of G be given with $\gamma_R(G') = \gamma_R D(G)$. Let $e \in E(G[V']) \setminus E'$. If no such edge exists, then G' is an induced subgraph. Adding edge e to G' has two effects. First, note that the degree of two vertices is increased by one. We have that $\forall v \in V' : d_{G'}(v) \leq d_{(V', E' \cup \{e\})}(v) \leq d_{G'}(v) + 1$. Furthermore, note that adding an edge deletes one of the nonedges $\{v_i, v_j\}$ with $v_i \neq v_j; v_i, v_j \in V'$, over which the minimum of $\max(d_{G'}(v_i), d_{G'}(v_j))$ is taken. However, deleting an element of a set over which a minimum is taken can never decrease the value of that minimum. Therefore, we have $\gamma_R(G') \leq \gamma_R((V', E' \cup \{e\}))$. The lemma is shown by applying this argumentation successively until an induced subgraph is obtained. □

Problem: γ_R-DEGENERACY
Instance: Graph $G = (V, E)$ with $|V| \geq 2$ and integer $k \geq 0$.
Question: Is $\gamma_R D(G) \geq k$?

Theorem 24. *The problem γ_R-DEGENERACY is NP-complete.*

Proof. Membership in NP is easy to see, since we only have to guess a subgraph and then compute γ_R of that subgraph in polynomial time. The hardness proof is a transformation of the known NP-complete problem VERTEX COVER, see [49].

Let a VERTEX COVER instance (G, l) be given with $G = (V, E)$ and $V = \{v_1, ..., v_n\}$. We assume that $1 \leq l \leq n - 1$, which is not a restriction, since $l = 0$ and $l = n$ are trivial instances for VERTEX COVER. We will construct a γ_R-DEGENERACY instance.

Construction: We take a clique with vertex set $U = \{u_1, ..., u_{n+l}\}$, an independent set $W = \{w_1, ..., w_{n+l-1}\}$, and we take the complement $\bar{G}$ of G. We add all edges between vertices in U and W, and all edges between vertices in W and V. The resulting graph G' (see Figure 3.2) is formally defined as follows:

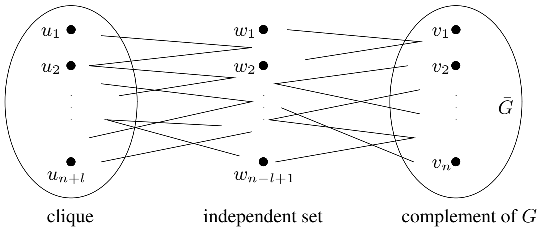

Figure 3.2. Graph G' constructed in the proof of Theorem 24

$$\begin{aligned}
G' &:= (V', E') \text{ where} \\
V' &= U \cup W \cup V \\
E' &= \{\ \{u_i, u_j\} \mid u_i \neq u_j \wedge u_i, u_j \in U\ \} \\
&\quad \cup \{\ \{u, w\} \mid u \in U \wedge w \in W\ \} \\
&\quad \cup \{\ \{w, v\} \mid w \in W \wedge v \in V\ \} \\
&\quad \cup \{\ \{v_i, v_j\} \notin E \mid v_i, v_j \in V, v_i \neq v_j\ \}
\end{aligned}$$

Our constructed γ_R-DEGENERACY instance is $(G', 2 \cdot n)$. Now, we show that there is a vertex cover for G of size at most k if, and only if $\gamma_R D(G') \geq 2 \cdot n$.

Claim. If there is a vertex cover V_1 of G of size at most l, then $\gamma_R D(G') = 2 \cdot n$, i.e. then there is a subgraph G'' of G' with $\gamma_R(G'') = 2 \cdot n$.

Proof. Assume V_1 is a vertex cover of G with $|V_1| = l$. Note that $\bar{G}[V \setminus V_1]$ is a clique of size $n - l$, since a nonedge $\{v_i, v_j\}$ in $\bar{G}[V \setminus V_1]$ ($v_i \neq v_j$), would be an edge in G and therefore, $v_i \in V_1$ or $v_j \in V_1$. We consider $G'' := G'[V' \setminus V_1]$. Since the remaining vertices of G form a clique in G'', the only nonedges are $\{w_i, w_j\}$ with $i \neq j; w_i, w_j \in W$, and $\{u_i, v_j\}$ with $u_i \in U; v_j \in V$. Furthermore, note that $\max(d_{G''}(w_i), d_{G''}(w_j)) = 2 \cdot n$, and $\max(d_{G''}(u_i), d_{G''}(v_j)) = 2 \cdot n$. Therefore, $\gamma_R(G'') = 2 \cdot n$. ◇

Claim. If $\gamma_R D(G') \geq 2 \cdot n$, i.e. if there is a subgraph G'' of G' with $\gamma_R(G'') \geq 2 \cdot n$, then there is a vertex cover V_1 of G of size at most l.

Proof. Let $G'' = (V'', E'')$ be given as an induced subgraph of G' (see Lemma 23), such that $\gamma_R D(G') = \gamma_R(G'') \geq 2 \cdot n$. Note that the only nonedges are $\{w_i, w_j\}$ with $i \neq j; w_i, w_j \in W$, $\{u_i, v_j\}$ with $u_i \in U; v_j \in V$, and $\{v_i, v_j\} \in E(G)$. The degree in G' (and also in G'') of a vertex in V is at most $(n - 1) + (n - l + 1) = 2 \cdot n - l < 2 \cdot n$. Hence, all pairs of vertices in V remaining in G'' are joined by an edge. Therefore, $V'' \cap V$ is a (perhaps empty) clique in G''.

Fact: $V'' \cap V$ is a clique of size at least $n - l$. This can be seen by assuming $|V'' \cap V| < n - l$. Note that every vertex in W has degree in G'' at most $(n + l) + |V'' \cap V| < 2 \cdot n$. If there are at least two vertices in W, then we have $\gamma_R(G'') < 2 \cdot n$. On the other hand, if there is at most one vertex of W left in G'', then every vertex in $V'' \cap U$ has degree at most $n + l < 2 \cdot n$. If $|V'' \cap V| = 0$, then there are at most $n + l + 1 \leq 2 \cdot n$ vertices left in G'', rendering $\gamma_R(G'') = 2 \cdot n$ impossible. If $|V'' \cap V| > 0$, then the only nonedges are $\{u_i, v_j\}$ with $u_i \in U, v_j \in V$. But then we have $\max(d_{G''}(u_i), d_{G''}(v_j)) < 2 \cdot n$, a contradiction. Hence, $V'' \cap V$ is a clique of size at least $n - l$.

Now we define $V_1 := V \setminus (V'' \cap V)$, i.e. V_1 contains exactly those vertices of V that are not present in G''. We will show that V_1 is a vertex cover of G of size at most l. We clearly have that $|V_1| \leq l$. Assume there is an edge $\{v_i, v_j\} \in E$ with $\{v_i, v_j\} \cap V_1 = \emptyset$. Then $\{v_i, v_j\}$ is a nonedge in G', and by definition of V_1, it is also a nonedge in G''. This is a contradiction to the fact that all pairs of vertices in V remaining in G'' are joined by an edge. Hence, V_1 is a vertex cover of G. ◇

Since the transformation described above is polynomial time computable, the NP-completeness of γ_R-DEGENERACY follows. □

Complexity of Computing $\gamma_R C$

Again, we formulate the decision problem corresponding to $\gamma_R C$ and prove its NP-completeness. The proof again resembles the proof of the NP-completeness of CONTRACTION DEGENERACY.

Problem: γ_R-CONTRACTION DEGENERACY
Instance: Graph $G = (V, E)$ with $|V| \geq 2$ and integer $k \geq 0$.
Question: Is $\gamma_R C(G) \geq k$?

Theorem 25. *The problem γ_R-*CONTRACTION DEGENERACY *is NP-complete.*

Proof. It is easy to see that the problem belongs to NP, since we only have to guess a minor and then compute the parameter γ_R of that minor in polynomial time. We prove the hardness by transforming the known NP-complete problem VERTEX COVER, see [49]. Let a VERTEX COVER instance (G, l) be given with $G = (V, E)$, $V = \{v_1, ..., v_n\}$ and $n \geq 2$ (note that $n < 2$ implies a trivial VERTEX COVER instance). From this instance, we will construct a γ_R-CONTRACTION DEGENERACY instance.

Construction:

We take l vertices $u_1, ..., u_l$, the complement $\bar{G}$ of G, a C_4 which is a cycle with vertices $w_1, ..., w_4$, and we take a vertex x. We add all edges between vertices u_i and v_j, between v_j and w_p and we connect vertex x to all vertices w_p, for $i \in \{1, ..., l\}$, $j \in \{1, ..., n\}$, $p \in \{1, ..., 4\}$. We call the resulting graph G', formally defined as follows (see Figure 3.3):

$$
\begin{aligned}
G' &:= (V', E') \text{ where} \\
V' &= \{u_1, \ldots, u_l\} \cup V \cup \{w_1, \ldots, w_4\} \cup \{x\} \\
E' &= \{\ \{u, v\} \mid u \in \{u_1, \ldots u_l\} \wedge v \in V\ \} \\
&\quad \cup \{\ \{v_i, v_j\} \notin E \mid v_i, v_j \in V, v_i \neq v_j\ \} \\
&\quad \cup \{\ \{v, w\} \mid v \in V \wedge w \in \{w_1, \ldots, w_4\}\ \} \\
&\quad \cup \{\ \{w_1, w_2\}, \{w_2, w_3\}, \{w_3, w_4\}, \{w_4, w_1\}\ \} \\
&\quad \cup \{\ \{w, x\} \mid w \in \{w_1, \ldots, w_4\}\ \}
\end{aligned}
$$

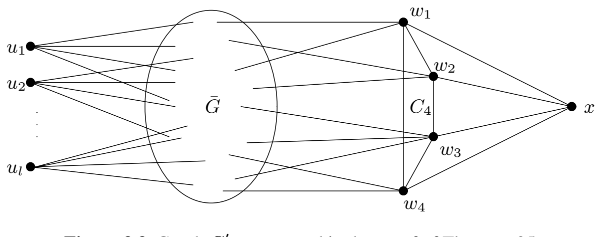

Figure 3.3. Graph G' constructed in the proof of Theorem 25

The constructed instance of the γ_R-CONTRACTION DEGENERACY problem is $(G', n+3)$. We will show that there is a vertex cover for G of size at most l if, and only if $\gamma_R C(G') \geq n + 3$.

Claim. If there is a vertex cover of G of size at most l, then there is a $E^* \subseteq E'$, such that $\gamma_R(G'/E^*) \geq n + 3$.

Proof. Suppose there is a vertex cover of size at most l. Now, take a vertex cover $V_1 = \{y_1, \ldots, y_l\}$ of G of size exactly l. (If we add vertices to a vertex cover, we obtain again a vertex cover.) In G', we contract edge set E^* defined as follows: $E^* := \{ \{u_i, y_i\} \mid i = 1, \ldots, l\}$; i.e. each vertex y_i in the vertex cover V_1 has the vertex u_i contracted to it. We claim that for the resulting graph G'/E^*, it holds that $\gamma_R(G'/E^*) \geq n + 3$. Therefore, we claim that all vertices in G'/E^*, apart from x, have degree $n+3$. Note that each vertex w in $\{w_1, ..., w_4\}$ is adjacent to vertex x, to all vertices in V and to exactly two other vertices from $\{w_1, ..., w_4\}$. Hence, all vertices in $\{w_1, ..., w_4\}$ are adjacent to $n + 3$ vertices. Now, we claim that the vertices of V form a clique in G'/E^*. Assume there are two vertices z_1 and z_2 in V with $z_1 \neq z_2$ and $\{z_1, z_2\} \notin E(G'/E^*)$. Therefore, $\{z_1, z_2\} \in E$, and hence z_1 or z_2 has to be in V_1. Assume w.l.o.g. that $z_1 \in V_1$, i.e. there exists an i in $\{1, ..., l\}$, such that $y_i = z_1$. Since we contracted $\{u_i, y_i\}$ and $\{u_i, z_2\} \in E(G')$, we have that $y_i = z_1$ is adjacent to z_2 in G'/E^*, which is a contradiction. Hence, V forms an n-clique and therefore, the degree in G'/E^* of a vertex in V is $n+3$ (since every vertex in V is also adjacent to $w_1, ..., w_4$). Thus, $\gamma_R(G'/E^*) = n+3$, since x is not adjacent to a vertex in V. ◇

Claim. If there is an $E^* \subseteq E'$, such that $\gamma_R(G'/E^*) \geq n + 3$, then there is a vertex cover $V_1 \subseteq V$ of G of size at most l.

Proof. Assume, there is a $E^* \subseteq E$, such that $\gamma_R(G'/E^*) \geq n + 3$. First, we state some observations about E^* and the structure of G'/E^*. After that, we construct a set $V_1 \subseteq V$ and show that this is a vertex cover of G.

Fact: $l \leq |E^*| \leq l + 1$. This is because, if $|E^*| < l$, then there are at least two vertices in $\{u_1, ...u_l, x\}$ left in G'/E^*, and hence, $\gamma_R(G'/E^*) \leq n$. On the other hand, if $l + 1 < |E^*|$, then G'/E^* has at most $n + 3$ vertices, and thus, $\gamma_R(G'/E^*) \leq n + 2$.

Fact: $|E^*| \neq l + 1$. To see this, we assume $|E^*| = l + 1$. From the previous fact, we already know that at least l vertices from $\{u_1, ...u_l, x\}$ have to be contracted to a neighbour. Depending on which of these vertices were contracted, we distinguish two cases to show $|E^*| \neq l + 1$.
Case 1 '$u_1, ..., u_l$ were contracted, and one more edge e was contracted in G'/E^*': If $e \in V \times V$ then the remaining vertices in V have degree at most $n + 2$, and hence, $\gamma_R(G'/E^*) \leq n + 2$. If $e \in V \times \{w_1, ..., w_4\}$, then one vertex in V is adjacent to x, and all other vertices in V have degree at most $n + 2$; hence, $\gamma_R(G'/E^*) \leq n + 2$. If $e \in \{w_1, ..., w_4\} \times \{w_1, ..., w_4\}$ then the vertices in V have degree at most $n + 2$, and therefore, $\gamma_R(G'/E^*) \leq n + 2$. If $e \in \{x\} \times \{w_1, ..., w_4\}$ then two vertices in $\{w_1, ..., w_4\}$ will have degree at most $n+2$ and are not adjacent; thus $\gamma_R(G'/E^*) \leq n+2$.
Case 2 '$u_1, ..., u_{i-1}, u_{i+1}, ..., u_l$ were contracted, x was contracted w.l.o.g. to w_1 and one more edge e was contracted': If $e \in \{u_i\} \times V$ then this case was already considered above. If $e \in V \times V$ then w_2 and w_4 have degree at most $n + 1$, and hence, $\gamma_R(G'/E^*) \leq n + 1$. If $e \in V \times \{w_1, ..., w_4\}$ then the vertices in $\{w_1, ..., w_4\} \setminus e$ have degree at most $n + 2$; thus $\gamma_R(G'/E^*) \leq n + 2$. If $e \in \{w_1, ..., w_4\} \times \{w_1, ..., w_4\}$ then the remaining vertices in $\{w_1, ..., w_4\}$ have degree at most $n + 2$; therefore, $\gamma_R(G'/E^*) \leq n + 2$.
In all cases and subcases, we concluded $\gamma_R(G'/E^*) \leq n+2$, which contradicts our initial assumption that $\gamma_R(G'/E^*) \geq n + 3$. Hence, $|E^*| = l + 1$ is not possible.

Fact: All vertices in $\{u_1, ..., u_l\}$ were contracted, and x was not contracted. We know that exactly l vertices in $\{u_1, ..., u_l, x\}$ were contracted. If a $u_i \in \{u_1, ..., u_l\}$ was not contracted, then x was contracted w.l.o.g. to w_1. Thus, w_2 and w_4 have degree at most $n + 2$ and are non-adjacent; we conclude $\gamma_R(G'/E^*) \leq n+2$, which is a contradiction. Hence, after all the considerations above, the only possibility is that $u_1, ..., u_l$ were contracted to a neighbour, and x was not contracted.

Fact: The vertices of V form a clique in G'/E^*. Since x is not adjacent to any vertex in V, all vertices in V must have degree $n + 3$. This is only possible if V forms a clique.

Now, we know that G'/E^* has the following structure. It consists of a clique on n vertices, all of which are adjacent to the vertices of a C_4. Furthermore, there is a vertex x adjacent to all vertices of the C_4. Hence, E^* contains exactly l edges, one for each $u_i, i \in \{1, ..., l\}$, with the other endpoint in V. We will now define $V_1 \subseteq V$ and show that this is a vertex cover of G. Let $V_1 := \bigcup_{e \in E^*} e \setminus \bigcup_{i=1,...l} u_i$. Clearly, $|V_1| = l$. We claim that V_1 is a vertex cover of G. Assume, there is an edge $f = \{z_1, z_2\}$ in G with $V_1 \cap f = \emptyset$. Hence, f is not an edge in $\bar{G}$ and also not in G'. Since V forms an n-clique in G'/E^*, edge f exists in G'/E^*, which means: f was created by contracting another edge $\{u_i, v_j\} \in E^*$, since u_i is adjacent to all vertices in V. This can only be the case if $v_j = z_1$ or $v_j = z_2$. According to the definition of V_1, we have: $v_j \in V_1$, which contradicts $V_1 \cap f = \emptyset$. Hence, V_1 is a vertex cover of size l. $\diamond$

As G' can be constructed in polynomial time, the NP-completeness of the γ_R-CONTRACTION DEGENERACY problem now follows. $\square$

We now strengthen the previous result by showing the NP-completeness even for bipartite graphs.

Theorem 26. *It is NP-complete to decide whether $\gamma_R C(G) \geq k$ given a bipartite graph G and an integer k.*

Proof. In Section 3.1.4, we will observe that $\delta C(G) \geq 3$ if, and only if $tw(G) \geq 3$. Since $\delta C(G) \leq \gamma_R C(G) \leq tw(G)$, we also have that $\gamma_R C(G) \geq 3$ if, and only if $tw(G) \geq 3$. It is known that graphs of treewidth at most two can be recognised in linear time, e.g. by using reduction rules (see [3, 11]) or the method described in [9]. Therefore, it is no restriction that we assume for technical reasons that $\gamma_R C(G) \geq 3$ (and hence, also $k \geq 3$).

The membership in NP is obvious. To prove NP-hardness, we use a polynomial transformation from γ_R-CONTRACTION DEGENERACY on general graphs, which is known to be NP-complete (see Theorem 25). Let an instance (G, k) of the γ_R-CONTRACTION DEGENERACY problem be given. We subdivide every edge in G, i.e. we place a new vertex on each edge, and obtain G', formally defined in the following way:

$$\begin{aligned} G' &:= (V', E') \text{ where} \\ V' &= V \cup \{ \, s_e \mid e \in E \, \} \\ E' &= \{ \, \{a, s_e\}, \{s_e, b\} \mid e = \{a, b\} \in E \, \} \end{aligned}$$

The new constructed instance is (G', k). G' is a bipartite graph, as all edges in E' are between a vertex in V and a vertex in $\{ \, s_e \mid e \in E \, \}$. Now, we show that $\gamma_R C(G) \geq k$ if, and only if $\gamma_R C(G') \geq k$.

Claim. If $\gamma_R C(G) \geq k$ then $\gamma_R C(G') \geq k$.

Proof. Note that G is a minor of G', since $G = G'/E^*$, where E^* is set of edges to undo the subdivisions. Therefore, any minor of G is also a minor of G'. Hence, the claim follows. $\diamond$

Claim. If $\gamma_R C(G') \geq k$ then $\gamma_R C(G) \geq k$.

Proof. Let $G_1 \preceq G'$ be a minor of G such that $\gamma_R(G_1) \geq k$. If all vertices in $\{ \, s_e \mid e \in E \, \}$ were contracted in G_1 to a neighbour, then G_1 is also a minor of G, and hence $\gamma_R C(G) \geq k$. Furthermore, if G_1 contains two vertices of $\{ \, s_e \mid e \in E \, \}$, then $\gamma_R(G_1) = 2$, since all vertices in $\{ \, s_e \mid e \in E \, \}$ are pairwise nonadjacent and have degree two. Therefore, we have to consider the case that exactly one vertex $s \in \{ \, s_e \mid e \in E \, \}$ is present in G_1. This vertex s subdivides an edge $\{u, v\} \in E$. We distinguish the following cases.

Case 1 '$\{u,v\} \notin E(G_1)$': Contracting edge $\{s,u\}$ does not change the degree of any vertex in $V(G_1)$ (apart from s). However, s cannot be the vertex determining $\gamma_R(G_1)$, since $d_{G_1}(s) = 2$. This contraction decreases the number of pairs over which the minimum is taken when computing $\gamma_R(G_1)$. Such a decrease can only increase the value of that minimum. Therefore, $k \leq \gamma_R(G_1) \leq \gamma_R(G_1/\{s,u\}) \leq \gamma_R C(G)$, since $G_1/\{s,u\} \preceq G$.
Case 2 '$\{u,v\} \in E(G_1)$': In this case u, v and s form a triangle. We define $G_2 := G_1 - s = G_1/\{s,u\} = G_1/\{s,v\}$. Let $\{x,y\}$ be a nonedge in G_2 that determines $\gamma_R(G_2)$, with $d_{G_2}(x) \leq d_{G_2}(y)$. Since $\{u,v\} \in E(G_2)$, $\{x,y\} \neq \{u,v\}$. If $\{x,y\} \cap \{u,v\} = \emptyset$, then $d_{G_2}(x) = d_{G_1}(x)$ and $d_{G_2}(y) = d_{G_1}(y)$. Hence, we have that $\gamma_R(G_1) \leq \max(d_{G_1}(x), d_{G_1}(y)) = \max(d_{G_2}(x), d_{G_2}(y)) = \gamma_R(G_2)$. We consider the case where $\{x,y\} \cap \{u,v\} \neq \emptyset$ in two subcases.
Subcase 2a '$x \in \{u,v\}$': We have $d_{G_1}(x) \geq d_{G_2}(x)$ and $d_{G_1}(y) = d_{G_2}(y)$, and $\{s,y\}$ is a nonedge in G_1. Therefore, we have that $\gamma_R(G_1) \leq \max(d_{G_1}(s), d_{G_1}(y)) \leq \max(d_{G_2}(x), d_{G_2}(y)) = \gamma_R(G_2)$.
Subcase 2b '$y \in \{u,v\}$': In this case, we have that $d_{G_1}(x) = d_{G_2}(x)$ and $d_{G_1}(y) \geq d_{G_2}(y)$, and $\{s,x\}$ is a nonedge in G_1. Hence, $\gamma_R(G_1) \leq \max(d_{G_1}(s), d_{G_1}(x)) \leq \max(d_{G_2}(x), d_{G_2}(y)) = \gamma_R(G_2)$.
Hence, we can conclude that $\gamma_R(G_1) \leq \gamma_R(G_2)$. Furthermore, note that G_2 is also a minor of G, and thus, we have that $k \leq \gamma_R(G_1) \leq \gamma_R(G_2) \leq \gamma_R C(G)$. $\diamond$

Because the transformation described above is a polynomial one, the theorem follows. $\square$

Complexity of Computing $\delta_2 C$

We formulate the decision problem corresponding to $\delta_2 C$ and state its NP-completeness.

Problem: δ_2-CONTRACTION DEGENERACY
Instance: Graph $G = (V,E)$ with $|V| \geq 2$ and integer $k \geq 0$.
Question: Is $\delta_2 C(G) \geq k$?

Theorem 27. *The problem δ_2-*CONTRACTION DEGENERACY *is NP-complete.*

To prove this, we can use an easier variant of the proof for Theorem 25. When computing γ_R, we consider non-adjacent vertices. However, we can observe that the proof of Theorem 25 also holds when computing δ_2.

3.1.4 Fixed Parameter Case of CONTRACTION DEGENERACY

Now, we consider the fixed parameter case of the CONTRACTION DEGENERACY problem, i.e. for a fixed integer k, we consider the problem to decide for any given graph G whether G has a minor with minimum degree k. Graph minor theory gives a fast answer to this problem. For a good introduction to the algorithmic consequences of this theory, see [45].

Theorem 28. *The* CONTRACTION DEGENERACY *problem can be solved in linear time when k is a fixed integer with $k \leq 5$, and it can be solved in $O(n^3)$ time when k is a fixed integer with $k \geq 6$.*

Proof. Let k be a fixed integer. Consider the class of graphs $\mathcal{G}_k = \{G \mid G$ has contraction degeneracy at most $k-1\}$. $\mathcal{G}_k$ is closed under taking minors: if H is a minor of G and H has contraction degeneracy at least k, then G has also contraction degeneracy at least k. As every class of graphs that is closed under minors has an $O(n^3)$ algorithm to test membership by Robertson-Seymour graph minor theory (see [45]), the theorem for the case $k \geq 6$ follows.

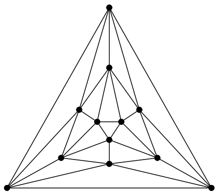

Figure 3.4. The icosahedron is a planar graph G with minimum degree $\delta(G) = 5$.

Suppose now that $k \leq 5$. There exists a planar graph G_k with minimum degree k (for example for $k = 5$ the icosahedron, see [22] or Figure 3.4). Hence, $G_k \notin \mathcal{G}_k$. A class of graphs that is closed under taking minors and does not contain all planar graphs has a linear time membership test (see [45]), which shows the result for the case that $k \leq 5$. □

It can be noted that the cases that $k = 1, 2$ and 3 are very simple: a graph has contraction degeneracy at least 1, if and only if it has at least one edge, and it has contraction degeneracy at least 2, if and only if it is not a forest. For a graph to have contraction degeneracy at least 3, all vertices of degree 2 or less have to be contracted recursively. If the result is a non-empty graph, the contraction degeneracy is at least 3. Vertices of degree 2 can be contracted to either of the neighbours without loss of generality. Thus, this gives a simple algorithm to identify the graphs with $\delta C(G) \geq 3$. The algorithm to determine if a graph has treewidth at least 3 works in exactly the same way, and hence, graphs with $\delta C(G) \geq 3$ are exactly those with $tw(G) \geq 3$.

The result is non-constructive when $k \geq 6$; when $k \leq 5$, the result can be made constructive by formulating the property that G has contraction degeneracy k in Monadic Second Order Logic (MSOL) for fixed k. Our formulation of the CONTRACTION DEGENERACY problem is based on [25] and uses macros defined there. Conn(V_1, E_1) is a macro that tests whether the vertex-set V_1 is connected using only the edges of E_1. Adj(x, y, E) checks whether there is an edge in E that joins x and y. We first give a macro that specifies if there is a path between two vertices x and y using only the vertices in V_1 and edges in E_1:

$$\mathrm{ConnectingPath}(x, y, V_1, E_1) \Leftrightarrow (\exists V_2 \subseteq V_1)(x \in V_2 \wedge y \in V_2 \wedge \mathrm{Conn}(V_2, E_1))$$

The problem CONTRACTION DEGENERACY can now be formulated as follows. We are constructing a minor by specifying V_0 – a non-empty set of vertices that remain in the graph or vertices that are contracted. Vertices not in V_0 are deleted. E_0 is the set of edges that are contracted. We then require that for every vertex $v_0 \in V_0$ there exist k vertices v_i ($1 \leq i \leq k$) which are not connected by edges in E_0. These vertices are the neighbours of v_0 after the edge-contractions, and hence, v_0 has degree at least k. If two such vertices would be connected by edges in E_0, then they would be contracted to one vertex, forming the same neighbour after contraction. Furthermore, there must be a path from v_0 to v_i using an edge $\{x, y\}$ that is not contracted and vertices that are not deleted. As an MSOL formula, this is written as:

$$\begin{aligned}
(\exists V_0, \exists E_0)\ &(\neg V_0 = \emptyset \wedge V_0 \subseteq V \wedge E_0 \subseteq E)(\forall v_0 \in V_0)(\exists v_1, ..., v_k \in V_0) \\
&\neg \mathrm{ConnectingPath}(v_i, v_j, V, E_0)(1 \leq i < j \leq k) \\
&\wedge(\exists x, y \in V_0)(\mathrm{Adj}(x, y, E) \\
&\wedge \mathrm{ConnectingPath}(v_0, x, V_0, E_0) \\
&\wedge \mathrm{ConnectingPath}(y, v_i, V_0, E_0)(1 \leq i \leq k))
\end{aligned}$$

Thus, we can solve the problem as follows: the result of [83] applied to G_k, a planar graph with minimum degree k, gives an explicit upper bound c_k on the treewidth of graphs in $\mathcal{G}_k = \{G \mid G \text{ has contraction degeneracy at most } k-1\}$. Test if G has treewidth at most c_k, and if so, find a tree-decomposition with width at most c_k with the algorithm of [9]. If G has treewidth at most c_k, use the tree-decomposition to test if the MSOL formula holds for G [39]; if not, we directly know that G has contraction degeneracy at least k. It should also be noted that the constant factors hidden in the O-notation of these algorithms are very large; it would be nice to have practical algorithms that do not rely on graph minor theory. We summarise the different cases in the following table.

k	Time	Reference
1	$O(n)$	trivial
2	$O(n)$	G is not a forest
3	$O(n)$	$tw(G) \geq 3$
4, 5	$O(n)$	[9, 39, 83], MSOL
fixed $k \geq 6$	$O(n^3)$	[81, 82]
variable k	NP-complete	Theorem 22

Table 3.1. Complexity of contraction degeneracy

3.1.5 On the Contraction Degeneracy

In this section, we consider some useful observations on the contraction degeneracy. We start by examining the contraction degeneracy of connected components and blocks of a graph, compared to the contraction degeneracy of the graph itself. Later, we will see that CONTRACTION DEGENERACY on chordal graphs is trivial. Furthermore, we elaborate on an upper bound for the contraction degeneracy.

Connected Components and Blocks

When G is connected, $\delta C(G)$ can also be defined as the maximum over all contractions of G of the minimum degree of the contraction. This does not necessarily hold for disconnected graphs: when G has connected components whose contraction degeneracy is smaller than the contraction degeneracy of G, we must delete this component entirely to obtain the minor with maximum minimum degree. The following lemma is easy to see.

Lemma 29. *For any graph G, it holds that:*

$$\delta C(G) = \max_{G_c}\{\delta C(G_c) \mid G_c \text{ is a connected component of } G\}$$

The maximal 2-connected subgraphs of G are the blocks of G (blocks are also called 2-connected components or biconnected components). Separating the graph into smaller subgraphs can help to solve problems or to compute graph parameters. That is why we are interested whether we can compute the contraction degeneracy of a graph, given the contraction degeneracy of the blocks of a graph. For the treewidth of a graph, we have the following lemma.

Lemma 30 (e.g. [11, 87]). *For any graph G, it holds that:*

$$tw(G) = \max\{tw(G') \mid G' \text{ is a block of } G\}$$

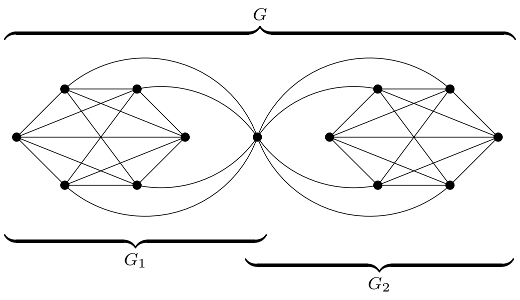

Figure 3.5. Graph G with two blocks G_1 and G_2

Unlike the treewidth, the contraction degeneracy of a graph does not equal the maximum contraction degeneracy over its blocks, as we can see from the example in Figure 3.5. It is easy to see that $\delta C(G) = 5$ and $\delta C(G_1) = \delta C(G_2) = 4$. However, we can prove the following lemma.

Lemma 31. *For any graph G, it holds that:*

$$\delta C(G) \geq \max\{\delta C(G') \mid G' \textit{ is block of } G\}$$

Proof. This is easy to see, since all vertices and edges not belonging to the block G' with maximum contraction degeneracy, can be deleted in G'. Hence, G' is a minor of G and therefore $\delta C(G) \geq \delta C(G')$. □

From Figure 3.5, we see another interesting consequence for the second smallest degree of a contraction or minor. By identifying a fixed vertex of minimum degree, we can compose a graph and show that $\delta_2 C(G) \leq tw(G)$ (which already has been shown using an alternative proof in Section 3.1.2).

Lemma 32. *For any graph G, it holds that:*

$$\delta_2 C(G) \leq tw(G)$$

Proof. Consider a minor G' of G. Let v be a vertex of minimum degree in G'. If $d_{G'}(v) = 0$, then $\delta_2(G') = \delta(G' - v) \leq tw(G' - v) \leq tw(G)$, since the minimum degree of a graph is a lower bound for its treewidth.

If $d_{G'}(v) \geq 1$, then let $v_1, v_2, ... v_{|V(G')|}$ be the sequence of vertices of G' ordered according to nondecreasing degree. We take $c := \left\lceil \frac{d_{G'}(v_2)}{d_{G'}(v_1)} \right\rceil$ copies of G', and we identify all vertices corresponding to v_1 as a single vertex v. Call the resulting graph G''. Let u be a vertex in G'' corresponding to a copy of v_2 in G'. Clearly, we have that $c \cdot d_{G'}(v_1) = d_{G''}(v) \geq d_{G''}(u) = d_{G'}(v_2)$. Therefore, u is a vertex of smallest degree in G''. From Lemma 30, it follows that $tw(G') = tw(G'')$. Because u is a vertex with smallest degree, we have that $d_{G''}(u) \leq tw(G'') = tw(G') \leq tw(G)$. Since this holds for all minors G' of G, we have $\delta_2 C(G) \leq tw(G)$. □

Chordal Graphs

Chordal graphs (also called triangulated graphs) form a special class of graphs that can be represented by a so called clique-tree. In a chordal graph, each simple cycle $C = v_1, ..., v_p$ of length $p \geq 4$ has a chord, i.e. an edge connecting two nonconsecutive vertices of C. In other words, chordal graphs do not contain any C_p as an induced subgraph for $p \geq 4$. All maximal cliques of a chordal graph can be arranged in a tree structure, such that a set of pairwise nondisjoint cliques forms a subtree of the tree (see e.g. [11]). This clique tree can be computed in linear time [7], and is often very useful to solve problems. However, exploiting this clique tree to compute the contraction degeneracy is not necessary. Note that chordal graphs can be recognised in linear time, and also a maximum clique of a chordal graph can be computed in linear time (see e.g. [51]).

Lemma 33. *Let $G = (V, E)$ be a chordal graph and let $W \subseteq V$ be a maximum clique in G. Then we have:*

$$\delta C(G) = |W| - 1$$

Proof. For any graph G, it holds that: $\delta C(G) \leq tw(G)$, see Section 3.1.2. If G is a chordal graph with maximum clique W, we have (see e.g. [11]): $tw(G) = |W| - 1$, and therefore: $\delta C(G) \leq |W| - 1$. Note that $G[W]$ is a minor of G and $\delta C(G) \geq \delta(G[W]) = |W| - 1$. □

As mentioned in the previous proof, it holds for a chordal graph with maximum clique W that $tw(G) = |W| - 1$ (see e.g. [11]). Therefore with Lemma 33, we have the following corollary.

Corollary 34. *For any chordal graph G, it holds that:*

$$\delta C(G) = tw(G)$$

An Upper Bound on the Contraction Degeneracy

In Chapter 4, we will see that the contraction degeneracy is a very successful treewidth lower bound. However, as we will also see, it has its limits, e.g. on planar graphs; or more general: on graphs with small genus (see also Section 3.1.6). Therefore, it is very interesting to have upper bounds on the contraction degeneracy. Such upper bounds on lower bounds, which can be NP-hard to compute, can inform us about the perspective to improve these lower bounds with better heuristics. As we will see in Section 3.1.6, the genus of the graph determines an upper bound for the contraction degeneracy. However, it is NP-hard to compute the genus of a graph [91]. Another idea for a contraction degeneracy upper bound is to take the maximum over all minors of $G = (V, E)$ of the average degree $\bar{d}$ of the minor.

$$\bar{d}(G) := \frac{2 \cdot m(G)}{n(G)} \qquad \bar{d}C(G) := \max_{G'}\{\bar{d}(G') \mid G' \preceq G \wedge n(G') \geq 1\}$$

It is clear that $\delta C(G) \leq \bar{d}C(G)$ for all graphs G. Unfortunately, computing this parameter is also NP-hard as we will now show.

Lemma 35. *Let a graph G and an integer $k \geq 0$ be given. The problem to decide whether $\bar{d}C(G) \geq k$ is NP-complete.*

Proof. The proof is very similar to the proof that the problem CONTRACTION DEGENERACY is NP-complete, see Section 3.1.3. Let (G, k) be a VERTEX COVER instance, with $G = (V, E)$, $n := |V|$, $n > 2$, $1 \leq k < n$, $m := |E|$, $m \geq 1$. We construct the graph G' as in Figure 3.6.

$$
\begin{aligned}
G' &:= (V', E') \text{ where} \\
V' &= \{u_1, ..., u_k\} \cup V \cup \{w_1, ..., w_{4n}\} \\
E' &= \{\ \{u_i, v\} \mid i \in \{1, ..., k\}, v \in V\} \cup \\
&\quad \{\ \{v_i, v_j\} \mid v_i, v_j \in V, v_i \neq v_j, \{v_i, v_j\} \notin E\} \cup \\
&\quad \{\ \{v, w_i\} \mid v \in V, i \in \{1, ..., 4n\}\} \cup \{\ \{w_i, w_j\} \mid i \neq j, i, j \in \{1, ..., 4n\}\}
\end{aligned}
$$

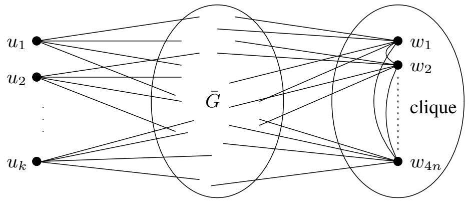

Figure 3.6. Graph G'

The constructed instance of the problem stated in the lemma is $(G', 5n-1)$. We will now show that there is a vertex cover for G of size at most k, if and only if $\bar{d}C(G') \geq 5n-1$.

Claim. If there is a vertex cover for G of size at most k, then $\bar{d}C(G') \geq 5n-1$.

Proof. Let be given a vertex cover of size at most k; add some vertices to it to obtain a vertex cover $V_1 = \{v_1, ..., v_k\}$ of size exactly k. Now we define the contraction-set $E_1 := \{\ \{u_i, v_i\} \mid i = 1, ..., k\}$. We will show that G'/E_1 is a clique. Assume there is a nonedge e in G'/E_1. By the construction this can only be between two vertices x and y in V. Hence, let be $e = \{x, y\}$. Therefore, e is an edge in G, and that is why $e \cap V_1 \neq \emptyset$. W.l.o.g. let be $x \in V_1$. That means there is an $i \in \{1, ..., k\}$, with $x = v_i \in V_1$. Note that u_i was contracted to $x = v_i$, and thus, v_i is adjacent to any vertex in V. This is a contradiction. Therefore, G'/E_1 is a clique with exactly $5n$ vertices, and hence, $\bar{d}C(G') \geq \bar{d}(G'/E_1) = 5n-1$. $\diamond$

Claim. If $\bar{d}C(G') \geq 5n-1$, then there is a vertex cover for G of size at most k.

Proof. Note that $n' := |V'| = k + 5n$ and $m' := |E'| = kn + \frac{n^2-n}{2} - m + \frac{4n(4n-1)}{2}$. Since $\bar{d}C(G') \geq 5n-1$, there is a minor G_1 with $\bar{d}(G_1) \geq 5n-1$, such that $G_1 = G'/E_1$. Furthermore, note that we can assume w.l.o.g. that G_1 is a contraction of G', i.e. there is a contraction-set E_1, because deleting edges does not increase the average degree, and instead of deleting vertices we can contract them to a neighbour.

In G', all vertices in $V \cup \{w_1, ..., w_{4n}\}$ have degree at least $4n$, while the vertices in $\{u_1, ..., u_k\}$ have degree exactly n. In a series of derivations, we will now show that even deleting vertices in $\{u_1, ..., u_k\}$ increases the average degree. Let G_2 be a minor of G' with at least $5n$ vertices, such that $\exists i \in \{1, ..., k\}$ with $u_i \in V(G_2)$, $n_2 = n(G_2)$ and $m_2 = m(G_2)$.

$$
\begin{aligned}
& 2 < n \\
\Rightarrow\quad & n^2 < 2n^2 - 2n \\
\Rightarrow\quad & n^2 < 8n^2 - 2n - n(6n) \\
\Rightarrow\quad & n^2 < 8n^2 - 2n - k(k+5n-1) \\
\Rightarrow\quad & nk + 4n^2 + n^2 < nk + 4n^2 + \tfrac{16n^2-4n}{2} - k(k+n+4n-1) \\
\Rightarrow\quad & n \cdot n_2 \leq n(k+4n+n) < nk + 4n^2 + \tfrac{4n(4n-1)}{2} - k(k+n+4n-1) \leq m_2
\end{aligned}
$$

Using just a few transformations, we see that $n_2 \cdot n < m_2$ is equivalent to $\frac{2m_2}{n_2} < \frac{2(m_2-n)}{n_2-1}$ (note that $n_2 > 1$), and hence, we have:

$$\bar{d}(G_2) = \frac{2m_2}{n_2} < \frac{2(m_2-n)}{n_2-1} \leq \frac{2(m_2-d_{G_2}(u_i))}{n_2-1} = \bar{d}(G_2-u_i)$$

This means that the average degree in G_2 is smaller than the average degree in $G_2 - u_i$. We now examine the structure of G_1. G_1 has at least $5n$ vertices, otherwise it could not have average degree at least $5n-1$. If G_1 contains a vertex from $\{u_1, ..., u_k\}$, we can delete it, which increases the average degree of G_1. If the new G_1 still has at least $5n$ vertices, we can iterate this process, stepwise delete a vertex from $\{u_1, ..., u_k\}$ and increasing the average degree. At the end, we obtain a graph with at most $5n$ vertices and hence, with average degree at most $5n-1$. Therefore, we can conclude that G_1 must have at most $5n$ vertices, otherwise it contains a vertex in $\{u_1, ..., u_k\}$. To have average degree $5n-1$, G_1 must be a complete graph on $5n$ vertices. Hence, the contraction-set E_1 (with $G'/E_1 = G_1$) contains exactly k edges to contract all vertices in $\{u_1, ..., u_k\}$ to a neighbour in V to turn V into a clique. We define $V_1 := \bigcup_{e \in E_1} e \cap V$, and show that V_1 is a vertex cover for G. Clearly, $|V_1| \leq k$. Assume that there is an edge $e := \{x, y\} \in E$ with $e \cap V_1 = \emptyset$. Thus, $e \notin E'$. Since e is an edge in G_1, there was an edge $\{u_i, x\}$ or $\{u_i, y\}$ contracted to obtain G_1. Hence, this edge is in E_1, and therefore $x \in V_1$ or $y \in V_1$, which is a contradiction. $\diamond$

The transformation above is a polynomial time transformation. Furthermore, membership in NP of the considered problem is obvious. Hence, the lemma follows. $\square$

It is interesting to note that the above proof also holds when replacing the average degree $\bar{d}$ by the minimum degree δ, because it is easy to see that $\bar{d}(G') = \delta C(G')$, for G' as in the proof. The origin of the upper bound, examined in this section, was the consideration of the following strategy. Successively contract an edge $\{x, y\}$ in a graph G, such that x and y have the minimum number of common neighbours, and record the maximum of the average degree $ad_{\max}(G)$ encountered during this process. It is clear that this strategy gives a lower bound on $\bar{d}C(G)$. Unfortunately, it is unlikely to be an upper bound on the contraction degeneracy. If $ad_{\max}(G)$ would also be an upper bound on the contraction degeneracy $\delta C(G)$, then we would have $\delta C(G) \leq ad_{\max}(G) \leq \bar{d}C(G)$. In the previous proof, however, we constructed an instance G' with $\delta C(G') = ad_{\max}(G') = \bar{d}C(G')$, and therefore, if $ad_{\max}(G)$ would be an upper bound on the contraction degeneracy, then we could compute $\bar{d}C(G')$ or $\delta C(G')$ in polynomial time, solving VERTEX COVER in polynomial time. Summarising, we can say that if $\delta C(G) \leq ad_{\max}(G)$ for all graphs G, then $P = NP$.

3.1.6 Graphs of Genus γ

Harary writes in [54] that every planar graph G with at least four vertices has at least four vertices of degree not exceeding five. Therefore, we have for a planar graph G that $\delta(G) \leq \delta_2(G) \leq 5$. As a consequence of this, we have that $\delta C(G) \leq \delta_2 C(G) \leq 5$, since planar graphs are closed under taking minors. This fact can also be observed in Chapter 4 in experiments computing δC and similar parameters. In this section, we examine a few basic observations, similar to the result above and related to the genus of a graph.

A graph G is embeddable in a surface S when it can be drawn on S in such a way that no two edges intersect apart from intersecting at vertices. G is planar, iff G can be embedded in the plane or in the sphere. We can generalise this concept by changing the underling surface S. One way of changing S is to add a handle to it. A handle is a kind of bridge over which an edge can go to avoid crossing the edge below the handle, see e.g. [53, 54]. The closed orientable surfaces S_0, S_1, S_2, ... are defined as follows. S_0 is the sphere, and S_{n+1} is obtained by adding a handle to S_n. The genus of S_i is i, for all $i \geq 0$. We now review some definitions related to the embeddings of graphs on surface S_i. A region of a graph embedding is a component of the surface the graph is embedded in, obtained by deleting the image of the graph from the surface (i.e. a component of the disconnected surface after carving out the edges of the graph). A cellular embedding of a graph G in S is an embedding such that each region is topologically equivalent to an open disk. A face of an embedding is the union of a region and its boundary. Introductory chapters on planarity and its generalisations to higher genus can be found in e.g. [53, 54]. See [67, 73] for more details about graphs on surfaces.

Definition 36. *The* genus $\gamma(G)$ *of the graph G is defined as follows.*

$$\gamma(G) := \min\{i \mid G \textit{ can be embedded in } S_i\}$$

The corresponding decision problem: 'Given a graph G and an integer $k \geq 0$, is $\gamma(G) \leq k$?' is NP-complete (see [91]). However, if k is fixed, there is a polynomially bounded algorithm for computing the genus (see [47]). The most well-known and well-studied case are the planar graphs, i.e. graphs of genus 0 (see e.g. [41, 57, 24]). Given a graph G and a minor G' of G, it is easy to see that $\gamma(G') \leq \gamma(G)$. The Euler-criterion for planar graphs also holds for higher genus:

Theorem 37 (see e.g. [53]). *Let $G = (V, E)$ be a connected graph with n vertices and m edges, and let G be cellularly embedded in S_γ. Let f be the number of faces of this embedding. Then we have that $n - m + f = 2 - 2 \cdot \gamma$.*

Since each face is surrounded by at least three edges, and each edge separates at most two faces, we have the edge-face inequality $3 \cdot f \leq 2 \cdot m$ (see e.g. [53]). With the last two formulas together, we obtain an upper bound on the number of edges of a graph with genus γ and n vertices (see e.g. [73]):

$$m \leq 3 \cdot (n - 2 + 2 \cdot \gamma) \tag{3.1}$$

Lemma 38. *Given a graph G with n vertices and genus γ, the following holds:*

1. $\delta(G) \leq 6 + \frac{12 \cdot \gamma - 12}{n}$
2. $\delta_2(G) \leq 6 + \frac{12 \cdot \gamma - 6}{n-1}$

Proof. (1.) It is obvious that G has at least $\frac{\delta(G) \cdot n}{2}$ edges. Hence, we have that $\frac{\delta(G) \cdot n}{2} \leq 3 \cdot (n - 2 + 2 \cdot \gamma)$, from which we can derive $\delta(G) \leq 6 + \frac{12 \cdot \gamma - 12}{n}$. (2.) At least $n - 1$ vertices have degree at least $\delta_2(G)$ in G. Therefore, G has at least $\frac{\delta_2(G) \cdot (n-1)}{2}$ edges, which implies $\delta_2(G) \leq 6 + \frac{12 \cdot \gamma - 6}{n-1}$. □

The following theorem is due to Ringel and Youngs and solves the genus problem for complete graphs.

Theorem 39 (see e.g. [73, 78]). *If $n \geq 3$ then*

$$\gamma(K_n) = \left\lceil \frac{(n-3) \cdot (n-4)}{12} \right\rceil$$

Lemma 40. *For any given graph G with at least two vertices, it holds that:*

$$\delta_2 C(G) \leq 5 + \gamma(G)$$

Proof. With Lemma 38, we have:

$$\delta_2 C(G) = \max_{G' \preceq G,\ n(G') \geq 2} \delta_2(G') \leq \max_{G' \preceq G,\ n(G') \geq 2} \left(6 + \frac{12 \cdot \gamma(G') - 6}{n(G') - 1} \right)$$

We first consider minors with at least 13 vertices. The following is easy to see:

$$\max_{G' \preceq G,\ n(G') \geq 13} \left(6 + \frac{12 \cdot \gamma(G') - 6}{n(G') - 1} \right) \leq \left(6 + \frac{12 \cdot \gamma(G) - 6}{12} \right) = 5.5 + \gamma(G)$$

We will show the same result for minors G' with at most 12 vertices in a case distinction, using Lemma 38, the fact that the genus does not increase when taking minors and that $\delta(G') \leq \delta_2(G') \leq n(G') - 1$. Let G' be a minor of G with at least 2 and at most 12 vertices.

Case 0 '$\gamma(G') = 0$': We have that $\delta_2(G') \leq 6 - \frac{6}{n(G')-1}$, which implies $\delta_2(G') \leq 5$ for all G'. Therefore, $\delta_2(G') \leq 5 \leq 5 + \gamma(G)$.

Case 1 '$\gamma(G') = 1$': Here, we have that $\delta_2(G') \leq 6 + \frac{6}{n(G')-1}$. If $n(G') \leq 7$, then $\delta_2(G') \leq 6$; and if $8 \leq n(G') \leq 12$, then $\delta_2(G') \leq 6 + \frac{6}{n(G')-1} < 7$. Hence, $\delta_2(G') \leq 6 \leq 5 + \gamma(G)$.

Case 2 '$\gamma(G') = 2$': To show $\delta_2(G') \leq 7 \leq 5 + \gamma(G)$, we distinguish subcases.
Subcase 2a '$n(G') \leq 8$': It is clear that $\delta_2(G') \leq 7$.
Subcase 2b '$n(G') = 9$': $G' \neq K_9$, since $\gamma(K_9) = 3$ (see Theorem 39), therefore, $G' \subseteq K_9 - e$ for an edge e. The two endpoints of e have degree at most 7 in G', hence $\delta_2(G') \leq 7$.
Subcase 2c '$n(G') = 10$': If $\delta_2(G') \geq 8$, then there are at least 9 vertices of degree at least 8 in G'. Thus, there are at least $\frac{8 \cdot 9}{2} = 36$ edges. However, with inequality (3.1), there are at most $3 \cdot (n - 2 + 2 \cdot \gamma) = 36$ edges. Hence, $G' = K_1 \cup K_9$, and therefore, $\gamma(G') \geq 3$. This is a contradiction, so $\delta_2(G') \leq 7$.
Subcase 2d '$n(G') = 11$': We have $\delta_2(G') \leq 6 + \frac{18}{n(G')-1} < 8$. Hence, $\delta_2(G') \leq 7$.

Case 3 '$\gamma(G') = 3$': Again, we use subcases to show $\delta_2(G') \leq 8 \leq 5 + \gamma(G)$.
Subcase 3a '$n(G') \leq 9$': It is clear that $\delta_2(G') \leq 8$.
Subcase 3b '$n(G') = 10$': $G' \neq K_{10}$, since $\gamma(K_{10}) = 4$ (see Theorem 39), therefore, $G' \subseteq K_{10} - \{e\}$ for an edge e. The two endpoints of e have degree at most 8 in G', hence $\delta_2(G') \leq 8$.
Subcase 3c '$n(G') = 11$': If $\delta_2(G') \geq 9$, then there are at least 10 vertices of degree at least 9 in G'. Thus, there are at least $\frac{9 \cdot 10}{2} = 45$ edges. However, with inequality (3.1), there are at most $3 \cdot (n - 2 + 2 \cdot \gamma) = 45$ edges. Hence, $G' = K_1 \cup K_{10}$, and therefore, $\gamma(G') \geq 4$. This is a contradiction, so $\delta_2(G') \leq 8$.
Subcase 3d '$n(G') = 12$': We have $\delta_2(G') \leq 6 + \frac{30}{n(G')-1} < 9$. Hence, $\delta_2(G') \leq 8$.

Case 4 '$\gamma(G') = 4$': Once more, considering subcases helps to show $\delta_2(G') \leq 9 \leq 5 + \gamma(G)$.
Subcase 4a '$n(G') \leq 10$': It is clear that $\delta_2(G') \leq 9$.
Subcase 4b '$n(G') = 11$': $G' \neq K_{11}$, since $\gamma(K_{11}) = 5$ (see Theorem 39), therefore, $G' \subseteq K_{11} - \{e\}$ for an edge e. The two endpoints of e have degree at most 9 in G', hence $\delta_2(G') \leq 9$.
Subcase 4c '$n(G') = 12$': We have $\delta_2(G') \leq 6 + \frac{42}{n(G')-1} < 10$. Hence, $\delta_2(G') \leq 9$.

Case 5 '$\gamma(G') = 5$': In this case, we have that $\delta_2(G') \leq 6+\frac{54}{n(G')-1}$. If $n(G') \leq 11$, then $\delta_2(G') \leq 10$; and if $n(G') = 12$, then $\delta_2(G') \leq 10$, since in that case $6 + \frac{54}{n(G')-1} < 11$. Hence, $\delta_2(G') \leq 10 \leq 5 + \gamma(G)$.

Case 6 '$\gamma(G') = 6$': For $n(G') \leq 12$, we automatically have that $\delta_2(G') \leq 11 \leq 5 + \gamma(G)$.

This case distinction is exhaustive, since a graph with at most 12 vertices can have genus at most 6, as can be derived from Theorem 39. The lemma now follows. □

It is easy to see that for any graph G with at least two vertices, it holds that $\delta C(G) \leq \delta_2 C(G)$ (see Section 3.1.2). Therefore, we have:

$$\delta C(G) \leq \delta_2 C(G) \leq \gamma(G) + 5$$

Note that $\forall G : \delta C(G) \leq \gamma(G) + 5$ can be proven directly by a similar and easier version of the proof of Lemma 40.

That the genus $\gamma(G)$ of a graph G determines an upper bound on $\gamma_R(G)$ can be derived from the fact that $\gamma_R C(G) \leq 2 \cdot \delta_2 C(G)$ (see Section 3.1.2) and Lemma 40. Taking both together, we have $\gamma_R C(G) \leq 10 + 2 \cdot \gamma(G)$.

To prove a similar statement for $\gamma_R C$ using combinatorial arguments as in Lemma 40 might be more complicated, because it seems to be difficult to combinatorially capture the property that the considered vertices must be nonadjacent. The following result does not give better upper bounds, but its proof helps to gain more insight.

Lemma 41. *There exists a function $c(\gamma)$, such that for all graphs G of genus γ, $\gamma_R(G) \leq c(\gamma)$.*

Proof. Let be given an ordering $v_1, ..., v_n$ of the vertices of G, such that $d(v_i) \leq d(v_{i+1})$, for all $i \in \{1, ..., n-1\}$. We define $\delta_i(G) := d_G(v_i)$, for all $i \in \{1, ..., n\}$.

We define a function $x(\gamma)$, that is related to the equation in Theorem 39 as follows:

$$x(\gamma) := \min\{k \mid \gamma(K_k) > \gamma\}$$

Thus, $x(\gamma)$ is the smallest number, such that every graph of genus at most γ does not contain $K_{x(\gamma)}$ as a subgraph. (For example, for planar graphs, i.e. $\gamma = 0$, we have $x(0) = 5$ by Kuratowski's theorem, see e.g. [53].) Now, note that any $x(\gamma)$ vertices in G do not form a clique. This is also true for the $x(\gamma)$ vertices that have smallest degree in G, and hence, we have that $\gamma_R(G) \leq \delta_{x(\gamma)}(G)$ (see Lemma 20).

We observe that G has at least $n-(x(\gamma)-1)$ vertices of degree at least $\delta_{x(\gamma)}(G)$. Therefore, G has at least $\frac{\delta_{x(\gamma)}(G)\cdot(n-(x(\gamma)-1))}{2}$ edges. Furthermore, with inequality (3.1), G has at most $3\cdot(n-2+2\cdot\gamma)$ edges. Altogether, this yields for a graph G with at least $x(\gamma)$ vertices:

$$\gamma_R(G) \leq \delta_{x(\gamma)}(G) \leq 6 \cdot \frac{n-2+2\cdot\gamma}{n-x(\gamma)+1} \leq \max_{n \geq x(\gamma)} 6 \cdot \frac{n-2+2\cdot\gamma}{n-x(\gamma)+1} = 6\cdot(x(\gamma)-2+2\cdot\gamma)$$

The last equality follows from the next claim and the observation that the maximum is taken for $n = x(\gamma)$.

Claim. For $n \geq x$,

$$\frac{n-2+2\gamma}{n-x+1} > \frac{n+1-2+2\gamma}{n+1-x+1}$$

Proof: With equivalence transformation, we obtain that the claim is true iff $(n-2+2\gamma)\cdot(n+1-x+1) > (n+1-2+2\gamma)\cdot(n-x+1)$. This is equivalent to $3\cdot x(\gamma)+2\cdot\gamma > 3$, which holds, since γ and $x(\gamma)$ are nonnegative and $x(\gamma)\geq 5$ is nondecreasing. $\diamond$

If the graph G has at most $x(\gamma)-1$ vertices, then $\delta_{x(\gamma)}(G)$ is not defined and $\gamma_R(G)\leq x(\gamma)-2$. Note that $x(\gamma)-2 < 6\cdot(x(\gamma)-2+2\cdot\gamma)$. Therefore, we define:

$$c(\gamma) := 6\cdot(x(\gamma)-2+2\cdot\gamma) < \infty \tag{3.2}$$

Now, it easily follows that $\gamma_R(G)\leq c(\gamma)$. $\square$

When γ and $x(\gamma)$ are considered to be constant, it is easy to see that:

$$\lim_{n\to\infty} 6\cdot\frac{n-2+2\cdot\gamma}{n-x(\gamma)+1} = 6$$

It is interesting to note that we therefore can conclude that for any fixed γ and $x(\gamma)$ (as in the previous proof), $\delta_{x(\gamma)}(G)$ will be at most 6, when n is increased towards infinity. The same is also true for $\delta(G)$, $\delta_2(G)$ and $\gamma_R(G)$, since $\delta(G)\leq\delta_2(G)\leq\gamma_R(G)\leq\delta_{x(\gamma)}(G)$.

Lemma 42. *Let G be a graph of genus γ. Then $\gamma_R C(G)\leq c(\gamma)$, where $c(\gamma)$ is as in equation (3.2).*

Proof. Let be given $G'\preceq G$, with $\gamma_R(G') = \gamma_R C(G)$. Let $x(\gamma)$ be defined as in the proof of Lemma 41.

Claim. $\gamma_1\leq\gamma_2 \Longrightarrow x(\gamma_1)\leq x(\gamma_2)$

Proof: Let be given γ_1 and γ_2 with $\gamma_1\leq\gamma_2$. For each k, we have $\gamma(K_k)>\gamma_2 \Rightarrow \gamma(K_k)>\gamma_1$. Therefore, $\gamma(K_{x(\gamma_2)})>\gamma_1$ and hence, using the definition of $x(.)$, we have $x(\gamma_1)\leq x(\gamma_2)$. $\diamond$

We know that $\gamma(G')\leq\gamma(G)$. Using Lemma 41 and the previous claim, we obtain $\gamma_R C(G) = \gamma_R(G')\leq 6\cdot(x(\gamma(G'))-2+2\cdot\gamma(G'))\leq 6\cdot(x(\gamma(G))-2+2\cdot\gamma(G)) = c(\gamma)$. $\square$

These upper bounds on $\delta C(G)$, $\delta_2 C(G)$ and $\gamma_R C(G)$ are not always sharp in the following sense. For a graph G on n vertices, we have that $\delta C(G)\leq n-1 = O(n)$, while there are graphs G on n vertices (e.g. K_n, see Theorem 39), such that $\gamma(G) = \Theta(n^2)$. The same is true for $\delta_2 C(G)$ and $\gamma_R C(G)$, since $\delta_2 C(G), \gamma_R C(G) = O(\delta C(G))$ (see Section 3.1.2).

The genus of a graph determines not only an upper bound on $\delta C(G)$, $\delta_2 C(G)$ and $\gamma_R C(G)$; it also determines a lower bound. Graph G is a minimal forbidden minor for S, if G cannot be embedded in S, but every proper minor of G can be embedded in S. The next theorem is the Excluded Minor Theorem, due to Robertson and Seymour.

Theorem 43 (see e.g. [73]). *For each surface S, the set of all minimal forbidden minors for S is finite.*

Lemma 44. *There exist functions $c_1(\gamma)$, $c_2(\gamma)$ and $c_3(\gamma)$, such that for all graphs G of genus γ:*

$$c_1(\gamma)\leq\delta C(G);\quad c_2(\gamma)\leq\delta_2 C(G);\quad c_3(\gamma)\leq\gamma_R C(G);$$

Proof. Let $F(S)$ denote the set of all minimal forbidden minors for S. We know that G can be embedded in S_γ, but not in $S_{\gamma-1}$. Therefore, G contains a minor G', such that $G'\in F(S_{\gamma-1})$. Since $F(S_i)$ is finite (Theorem 43), we can easily define $c_1(\gamma)$ to be the minimum over all graphs in $F(S_{\gamma-1})$ of the minimum degree of the graph. It is easy to see that we have:

$$c_1(\gamma) := \min_{G_1 \in F(S_{\gamma-1})} \delta(G_1) \leq \delta(G') \leq \delta C(G)$$

Hence, $c_1(\gamma)$ only depends on γ and it is a lower bound on $\delta C(G)$ for any graph G of genus γ. In the same way, we can use $c_2(\gamma) := \min_{G_1 \in F(S_{\gamma-1})} \delta_2(G_1) \leq \delta_2 C(G)$ and $c_3(\gamma) := \min_{G_1 \in F(S_{\gamma-1})} \gamma_R(G_1) \leq \gamma_R C(G)$. □

Note that the proof of Lemma 44 is nonconstructive, due to its use of Theorem 43. Even though $F(S)$ is finite for each surface S, its size can grow rapidly when the genus of the surface increases. For example, $F(S_0) = \{K_5, K_{3,3}\}$. Therefore, $c_1(1) = c_2(1) = c_3(1) = 3$. However, Cattell et al. and Mohar and Thomassen mention in [30, 73] that $F(S_1)$ probably contains more than 2000 graphs.

3.2 Maximum Cardinality Search Lower Bound

Maximum Cardinality Search (abbreviated: MCS) is a method to number the vertices of a graph. It was first introduced by Tarjan and Yannakakis for the recognition of chordal graphs [90]. It works as follows: We start by giving some vertex number 1. In step $i = 2, \ldots, n$, we choose an unnumbered vertex v that has the largest number of already numbered neighbours, breaking ties as we wish. Then we associate number i to vertex v. An *MCS ordering* ψ can be defined by mapping each vertex to its number $\psi(v)$ in this procedure. For a fixed MCS ordering ψ, let $v_i := \psi^{-1}(i)$.

Definition 45. *Let a graph G and an MCS ordering ψ of G be given, and let $v_i := \psi^{-1}(i)$. The visited degree $vd_\psi(v_i)$ of v_i is defined as follows:*

$$vd_\psi(v_i) := d_{G[v_1,\ldots,v_i]}(v_i)$$

The visited degree $MCSLB_\psi$ of an MCS ordering ψ is defined as follows:

$$MCSLB_\psi := \max_{i=1,\ldots,n} vd_\psi(v_i)$$

The maximum visited degree $MCSLB(G)$ of graph G is defined as follows:

$$MCSLB(G) := \max_\psi \{MCSLB_\psi \mid \psi \text{ is MCS ordering of } G\}$$

Lucena showed the following theorem.

Theorem 46 (see [70]). *For any MCS ordering ψ of graph G, it holds that:*

$$MCSLB_\psi \leq tw(G)$$

Thus, an MCS numbering gives a lower bound on the treewidth of a graph. This lower bound is never smaller than the degeneracy, but can be larger. However, it is NP-hard to compute $MCSLB(G)$.

Lemma 47 (see [14]).

- *For every graph G and MCS ordering ψ of G, it holds that: $\delta D(G) \leq MCSLB_\psi$.*
- *It is NP-complete to decide for any given graph G and integer $k \leq |V|$, whether $MCSLB(G) \geq k$.*

3.2.1 MCS Treewidth Lower Bound with Contraction

As introduced in the previous section, we obtain a lower bound on the treewidth of a graph from a maximum cardinality search ordering. Motivated by this, we also study contraction in combination with the MCS algorithm.

Definition 48.

$$MCSLBC(G) := \max_{G' \preceq G} MCSLB(G')$$

From the definition, it is evident that $MCSLB(G) \leq MCSLBC(G)$, for any graph G.

Lemma 49. *For any given graph G, it holds that:*

$$\delta C(G) \leq MCSLBC(G) \leq tw(G)$$

Proof. To see the first inequality, let $G' \preceq G$ be given, such that $\delta(G') = \delta C(G)$. Applying Lemma 47 and Definition 48, we have that $\delta C(G) = \delta(G') \leq \delta D(G') \leq MCSLB(G') \leq MCSLBC(G)$.

For the second inequality, let $G' \preceq G$ be given, such that $MCSLB(G') = MCSLBC(G)$. Then we have $MCSLBC(G) = MCSLB(G') \leq tw(G') \leq tw(G)$, using Theorem 46 and Lemma 13. □

3.2.2 NP-completeness

We define four decision problems, and we show that each of these is either NP-complete or NP-hard, respectively. For some of these, we also can show in a subsequent section that the fixed parameter cases are tractable.

Problem: MCSLB WITH CONTRACTION
Instance: Graph $G = (V, E)$, integer k.
Question: Does G have a contraction H, and H an MCS ordering ψ with the visited degree of ψ at least k?

Problem: MCSLB WITH MINORS
Instance: Graph $G = (V, E)$, integer k.
Question: Is $MCSLBC(G) \geq k$? I.e. does G have a minor H, and H an MCS ordering ψ with the visited degree of ψ at least k?

Problem: MINMCSLB WITH CONTRACTION
Instance: Graph $G = (V, E)$, integer k.
Question: Does G have a contraction H, such that every MCS ordering ψ has visited degree at least k?

Problem: MINMCSLB WITH MINORS
Instance: Graph $G = (V, E)$, integer k.
Question: Does G have a minor H, such that every MCS ordering ψ has visited degree at least k?

We now analyse the complexity of finding an optimal way of contracting and building an MCS ordering to obtain the best lower bound possible with this method. Unfortunately, the problem to determine if some bound can be obtained with MCS for a graph obtained from G by contracting edges is also NP-complete.

Theorem 50. MCSLB WITH CONTRACTION *is NP-complete.*

Proof. Clearly MCSLB WITH CONTRACTION belongs to NP. We just have to guess a contraction H and an MCS ordering ψ and check in polynomial time, whether the visited degree of ψ in H is at least k.

To prove NP-hardness, we use a transformation from VERTEX COVER. Let a VERTEX COVER instance (G, k) be given, where $G = (V, E)$ with $n = |V|$, and k is an integer. We construct a graph G' in the following way:

Construction.

First, we take $n+2$ copies of the complement of G. We call the vertices in these copies *graph vertices*. We add $k \cdot (n + 2)$ *extra vertices*. Each extra vertex has degree n: it is adjacent to all graph vertices in one copy of $\bar{G}$ and no other vertex; each copy has exactly k such extra vertices. Hence, in total, we have $k(n + 2)$ extra vertices. Finally, we add an edge between each pair of graph vertices that belong to different copies. Let G' be the resulting graph, see Figure 3.7. The MCSLB WITH CONTRACTION instance is $(G', n(n + 2) - 1)$.

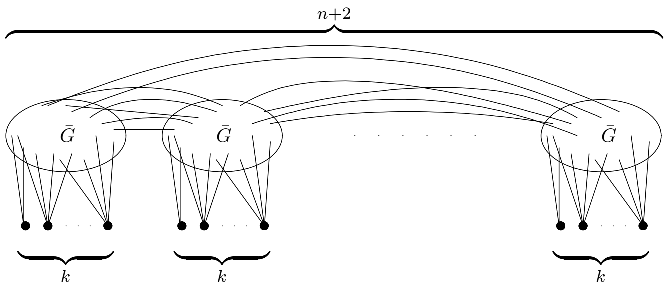

Figure 3.7. The graph G' constructed for the transformation

Now, we will show that G' has a contraction H that has an MCS ordering ψ with the visited degree of ψ at least $n(n + 2) - 1$, if and only if G has a vertex cover of size at most k.

Claim. If G has a vertex cover of size at most k, then G' has a contraction H that has an MCS ordering ψ with the visited degree of ψ at least $n(n + 2) - 1$.

Proof: Let V' be a vertex cover of G of size at most k. Now, we perform the following in each copy of $\bar{G}$. Contract all the extra vertices to vertices in the vertex cover V', such that each vertex in V' has at least one extra vertex contracted to it. This turns the set of graph vertices of G' into a clique of size $n(n+2)$, because for each pair of nonadjacent graph vertices v, w in G', $\{v, w\}$ is an edge in G, so an extra vertex, adjacent to v and w is contracted to v or w, after which the edge $\{v, w\}$ is formed in H. (Compare with the proof of Claim 3.1.3.) As H is a clique of $n(n + 2)$ vertices, any MCS ordering of H has visited degree exactly $n(n + 2) - 1$. ◇

Now, we will show the other direction. For this, we need a series of claims. Suppose that G' has a contraction H that has an MCS ordering ψ with the visited degree of ψ at least $n(n + 2) - 1$. Let y be the first vertex in ψ that is visited with visited degree $n(n + 2) - 1$, and let Y be the vertices that are visited up to y (including y). Note that y must be a graph vertex. With Theorem 46, $H[Y]$ has treewidth at least $n(n+2)-1$. Let X be the set of the vertices in H that are extra vertices that are not contracted.

Claim. There are at most $n+1$ copies of $\bar{G}$ that have at least one extra vertex that belongs to $X \cap Y$.

Proof: Consider the MCS ordering ψ up to the point that there are $n+1$ copies of $\bar{G}$ with at least one extra vertex in $X \cap Y$. As the set of visited vertices is connected, each copy must have a (possibly contracted) graph vertex that is visited. Before we can visit a vertex in X of the last copy, we must first visit a (possibly contracted) graph vertex of that copy. After that visit, each graph vertex has visited degree at least $n+1$, while vertices in X have degree at most n, so yet unvisited vertices in X will not be visited before all graph vertices are visited, in particular, only after y is visited. $\diamond$

So, there is at least one copy of $\bar{G}$ that has no uncontracted extra vertices in Y. Let V_i be the set of vertices of that copy in Y.

Claim. There are at least $n(n+2)$ graph vertices in Y.

Proof: If the opposite holds, then the treewidth of $H[Y]$ would be less than $n(n+2)-1$. Consider e.g. the following triangulation of $H[Y]$: turn the set of (possibly contracted) graph vertices into a clique. The maximum clique size will be less than $n(n+2)$ and the treewidth less than $n(n+2)-1$. This contradicts the fact that the treewidth of $H[Y]$ is at least $n(n+2)-1$. $\diamond$

Claim. V_i is a clique.

Proof: Assume the opposite. Let v and w be non-adjacent vertices in V_i. We can triangulate $H[Y]$ as follows: Add an edge between each pair of non-adjacent (possibly contracted) graph vertices, except that we do not add the edge $\{v, w\}$. Since V_i does not have extra vertices that are not contracted, this gives a chordal graph. The vertices in X are simplicial with degree at most n. After we remove these, we get a graph that is obtained by removing an edge from a clique with at most $n(n+2)$ vertices, yielding a graph with clique-size at most $n(n+2)-1$. Hence the treewidth is at most $n(n+2)-2$, which is a contradiction. $\diamond$

Because there are $n(n+2)$ graph vertices in Y, we know that $|V_i| = n$, and we cannot have contracted other graph vertices to vertices in V_i, since then we would have less then $n(n+2)$ graph vertices in Y. So, V_i was formed into a clique by the contraction of the k extra vertices of the copy to the graph vertices in V_i. Let Z be the set of vertices in V_i that have an extra vertex contracted to it. We have $|Z| \leq k$.

Claim. Z is a vertex cover.

Proof: For each edge $\{v, w\} \in E$, v and w are non-adjacent in H. Thus, we must have an extra vertex contracted to v or an extra vertex contracted to w. Therefore, we have $v \in Z$ or $w \in Z$ for each edge $\{v, w\} \in E$. $\diamond$

Hence, we can conclude that if G' has a contraction H that has an MCS ordering ψ with the visited degree of ψ at least $n(n+2)-1$, then G has a vertex cover of size at most k, which proves the other direction. This concludes the proof of the NP-completeness of MCSLB WITH CONTRACTION. $\square$

The same proof can be used for the related problems given at the beginning of this section, except that membership in NP is trivial only for MCSLB WITH MINORS, and we have no proof for membership in NP for the other two problems. Therefore, we conclude the following statement.

Corollary 51. MCSLB WITH MINORS *is NP-complete, and* MINMCSLB WITH CONTRACTION *and* MINMCSLB WITH MINORS *are NP-hard.*

3.2.3 Fixed Parameter Case

The fixed parameter case of MCSLB WITH MINORS can be solved in linear time with the help of graph minor theory. Observing that the set of graphs $\{G \mid G$ does not have a minor H, such that H has an MCS ordering ψ with the visited degree of ψ at least $k\}$ is closed under taking of minors and does not include all planar graphs (see [14]), the Graph Minor theorem of Robertson and Seymour and the results in [9, 83] give us the following result. See again [45] for more background information.

Theorem 52. MCSLB WITH MINORS *and* MINMCSLB WITH MINORS *are linear time solvable for fixed k.*

The fixed parameter cases of MINMCSLB WITH MINORS give an $O(n^3)$ algorithm (linear when $k \leq 5$), similar as for Theorem 28. Note that these results are non-constructive, and that the constant factors in the O-notation of these algorithms can be expected to be too large for practical purposes.

3.3 Improved Graphs Might Improve Lower Bounds

A further improvement of the lower bounds can be obtained by using a method found by Clautiaux et al. [32]. This method is based on 'improved graphs' and uses another treewidth lower bound algorithm as a subroutine. In [32], the authors use the degeneracy as a subroutine, but one can also use other algorithms. Our experiments showed that the contraction degeneracy heuristics generally outperform the method of [32] with degeneracy, but when we combine the method of [32] with some of the heuristics of Chapter 4, we get in several cases an additional small improvement of the lower bound. We finally propose a heuristic that combines the method of [32] and contraction in another way, by doing a contraction between every round of 'graph improvement'. See Section 4.3 for more details. This latter heuristic often costs considerably more time, but can give also significant increases to the lower bound.

In [12], two notions of *improved graphs* were introduced. Let k be an integer. The $(k+1)$-*neighbours improved graph* $G' = (V, E')$ of $G = (V, E)$ is obtained as follows: we take G, and then, as long as there are non-adjacent vertices u and v that have at least $k+1$ common neighbours in the graph, we add the edge $\{u, v\}$. This improvement step is motivated by the following lemma.

Lemma 53 (see [9, 12, 32]). *Any tree-decomposition of G with width at most k is also a tree-decomposition of the $(k+1)$-neighbours improved graph G' of G with width at most k, and vice versa.*

Clautiaux et al. use improved graphs to provide iterative methods to improve existing lower bounds for treewidth [32]. They use an algorithm for δD for computing lower bounds, but their approach works with every lower bound heuristic. Their algorithm LB_N works as follows:

- Suppose we have a lower bound $LB \leq tw(G)$ on the treewidth of G (e.g. LB was computed with an algorithm for the degeneracy).
- Use as hypothesis that $LB = tw(G)$. Build the $(LB+1)$-neighbours improved graph G' of G. (Note that if the hypothesis holds, then $tw(G) = tw(G')$)
- Compute a lower bound LB' of G' (e.g. with an algorithm for the degeneracy).
- If $LB' > LB$, we have a contradiction, showing the hypothesis $LB = tw(G)$ to be wrong.
- Therefore, $LB < tw(G)$ and $LB + 1$ is also a lower bound.
- Set LB to $LB + 1$, and repeat the process until there is no contradiction.

In Chapter 4, we propose several heuristics or exact algorithms to compute (lower bounds) of the parameters we considered here. For simplicity, we will give these algorithms the same name (maybe extended by a strategy used in the algorithm) as the parameter they are computing.

We see that the LB_N algorithm uses another treewidth lower bound algorithm as a subroutine, and thus, for every choice of such an algorithm, we obtain a different version of the LB_N algorithm. If algorithm Y is used as subroutine, then we call the resulting algorithm LBN(Y), e.g. the algorithm discussed by Clautiaux et al. in [32] is the LBN(δD) algorithm.

In [32], Clautiaux et al. also propose a related method, that sometimes gives better lower bounds, but also uses more time. Here, we have a different notion of improved graph. Let k be an integer. The $(k+1)$*-paths improved graph* $G'' = (V, E'')$ of $G = (V, E)$ is obtained by adding an edge $\{u, v\}$ to E for all vertex pairs u and v such that there are at least $k+1$ vertex-disjoint paths between u and v in G. Similar to Lemma 53, we have here the following.

Lemma 54 (see [9, 12, 32]). *Any tree-decomposition of G with width at most k is also a tree-decomposition of the $(k+1)$-paths improved graph G'' of G with width at most k, and vice versa.*

We can build the $(k+1)$-paths improved graph in polynomial time, as we can decide in polynomial time whether there are at least $k+1$ vertex-disjoint paths between a pair of vertices with help of network flow techniques. (See [32] for more details on how an undirected graph is transformed into a network flow instance, in order to find $k+1$ vertex-disjoint paths between a pair of vertices.) However, the running time to compute the paths improved graph is much larger than for the neighbour version. If we use $(k+1)$-paths improved graphs instead of $(k+1)$-neighbours improved graphs, then we obtain a new lower bound heuristic for treewidth, called LB_P in [32]. If we use as subroutine a lower bound algorithm Y in this algorithm, we call the resulting algorithm LBP(Y).

3.4 Concluding Remarks

In this chapter, we gave a theoretical analysis of a number of graph parameters, all related to treewidth. The first set of parameters is based on the degree of specific vertices in the graph, or in a subgraph or minor of the graph. The other parameters are based on Maximum Cardinality Search. Each of the parameters is a treewidth lower bound. Some of these parameter are a new combination of known parameters with taking subgraphs or minors. Especially the parameters involving edge contraction (i.e. minors) will experimentally prove to be a very vital idea.

We obtained results that show how the parameters are related to each other. We also examined the computational complexity of the parameters. Here, it is interesting to note that all contraction degeneracy problems are NP-hard, while the degeneracy problems are polynomial (see Section 4.2). However, an exception is the computation of the γ_R-degeneracy, which we showed to be NP-hard. Figure 3.8 represents some of the theoretical results. A thick line between two parameters indicates that the parameter below is smaller or equal to the parameter above, as stated by Lemmas 16, 17, 18, 47 and 49. The thin line marks the border between polynomial time computability and NP-hardness of the corresponding parameters, see Lemma 47, Theorems 22, 24, 25, 27, 50 and Corollary 51.

We also made some simple statements on the contraction degeneracy and related parameters. We observed that the genus of a graph determines upper bounds on these parameters. As mentioned in Section 3.1.6, these upper bounds are not sharp. It therefore remains a topic for further research to find better bounds that might be based on the genus of a graph. Another topic for further investigation is the relationship of the parameters $MCSLB$ and $MCSLBC$ compared to δC, $\delta_2 C$ or $\gamma_R C$. Some observations in this direction are already given in [14].

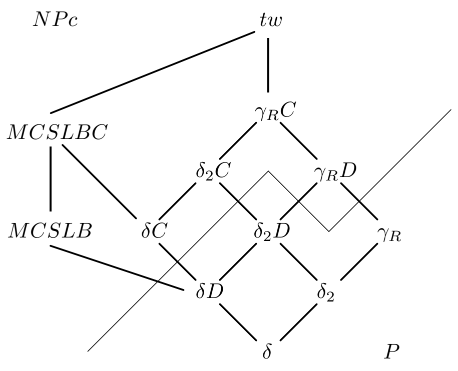

Figure 3.8. An overview of some theoretical results

At the end of this chapter, we looked at a general method to improve treewidth lower bounds. This method is based on so called graph improvements (see [9, 12, 32]). The potential of this method from a theoretical point of view seems to be hard to analyse, but might be an interesting topic for further considerations.

Apart from its function as a treewidth lower bound, the contraction degeneracy appears to be an attractive and elementary graph measure, worth further study. For instance, interesting topics are its computational complexity on special graph classes (for chordal graphs, see Section 3.1.5, and for cographs see Chapter 5), or the complexity of approximation algorithms with a guaranteed performance ratio.

4

Treewidth Lower Bounds: Algorithms, Heuristics and Experiments

In the previous chapter, we described methods to improve on existing treewidth lower bounds, and based on these methods, we introduced parameters that are also treewidth lower bounds. In this chapter, we give an experimental evaluation of these treewidth lower bounds and examine how well they perform in practice on input graphs from real world applications. Some of the treewidth lower bounds can be computed in polynomial time, others are NP-hard to compute. Before looking at algorithms and heuristics for computing treewidth lower bounds, we consider a data structure in Section 4.1 that is used in some of the algorithms and heuristics. In Section 4.2, we consider algorithms for the lower bounds that can be computed in polynomial time. For the lower bounds that are NP-hard to compute, we propose several heuristics in Section 4.3. For simplicity, we will give these algorithms and heuristics the same name (maybe extended by a strategy used in the algorithm or heuristic) as the parameter they are computing. The experimental results of the algorithms and heuristics can be found in Section 4.4. The content of this chapter is adapted from joint work with Hans L. Bodlaender and Arie M. C. A. Koster, see [17, 18, 65, 66, 99]. The implementations of the heuristics were done by Arie M. C. A. Koster.

4.1 A Bucket Adjacency-List Data Structure

In this section, we describe a data structure that is used in algorithms computing the degeneracy δD and similar parameters (see Section 3.1.1). It extends the graph adjacency-list data structure by a bucket structure (similar to bucket sort, sorted on vertex degree), enabling fast operations on this structure. It avoids a logarithmic factor that is typical for tree-based priority queues (as used in e.g. [63]).

Design of the Data Structure

The advanced adjacency-list of $G = (V, E)$ is the extension of the standard adjacency-list data structure, where for every vertex pointers to its adjacent vertices are stored in a doubly linked list. We use this advanced adjacency-list, and furthermore, we use cross pointers for each edge $\{v_i, v_j\}$. Such a cross pointer connects vertex v_i in the list for vertex v_j directly with vertex v_j in the list for vertex v_i.

In addition to this advanced adjacency-list, we create $n = |V|$ buckets that can be implemented by doubly-linked lists $B_0, ..., B_{n-1}$. List B_d contains exactly those vertices (or pointers to them) with degree d in the current graph. We associate to every vertex v a pointer $p(v)$ that points to the exact position in the list B_d that contains v for the appropriate d. We also maintain a value $d(v)$ for every vertex v that simply is the degree of v, see Figure 4.1.

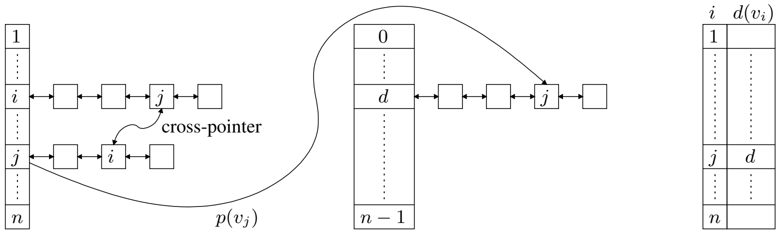

Figure 4.1. Data structure with doubly-linked adjacency-lists and cross pointers; buckets as doubly-linked lists in the middle of the figure; j in bucket d means $d(v_j) = d$ which can also be seen in the array on the right.

Cost of Operations

Here, we briefly describe how the graph operations that we want to perform in our algorithms can be carried out on our data structure and how much running time they need.

Construction.

Given a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$ as a (standard) adjacency-list, the advanced adjacency-list structure with cross pointers as described above can be constructed in time $O(n + m)$ by traversing the (standard) adjacency-list and constructing doubly-linked lists and the cross pointers during the traversal. Applying a bucket sort and building the pointers $p(v_i)$ for all $i = 1, ..., n$ can also be done in time $O(n + m)$. It is an easy task to compute the values $d(v_i)$ for all $i = 1, ..., n$ in the required time.

Degree Update.

Let be given a vertex v_i. Changing the degree of v_i can be done in constant time in the following way. The value of $d(v_i)$ gives us the bucket v_i is contained in. From the new degree, we also have the bucket in which v_i has to be inserted. This insertion (at the start or end of the corresponding doubly linked list), the deletion in the old bucket-list (using pointer $p(v_i)$) and also updating $d(v_i)$ can all be done in constant time.

Vertex Deletion.

Deleting a vertex v_i means deactivating it in the advanced adjacency-list structure. For every vertex v_j in the adjacency-list of v_i, we have to delete v_i in the adjacency-list of v_j (using cross pointers), and we also have to update the degree of v_j. All can be done in constant time per neighbour v_j of v_i. Also, deleting v_i from its corresponding bucket list can be done in constant time using $p(v_i)$. Therefore, deleting vertex v_i takes time $O(d(v_i))$. Note that also $d(v_i)$ edges are deleted in the graph.

Edge Contraction.

Contracting an edge $\{v_i, v_j\}$, which results in a vertex that we also call v_i, can be done as follows. First, we traverse the list of neighbours of v_j and for each such neighbour v, we exchange v_j by v_i in the adjacency-list of v (using cross pointers). Therefore, all vertices v that were adjacent to v_j are now adjacent to v_i according to their own lists. Now, we concatenate the adjacency-list of v_j to the one of v_i, and we deactivate (i.e. delete) v_j. This concatenation including checking for multiple occurrences of vertices and occurrences of vertices v_i and v_j can be done in $O(d(v_i) + d(v_j))$ time as follows. We use an additional (binary) array A running from 1 to n. We initialise it to be 0 on all positions. This initialisation has to be done only once for all edge-contractions. Now we traverse the adjacency-list of v_i and set $A(x) = 1$ for all neighbours v_x of v_i. When we encounter v_j in this pass, we do not modify A, and we cut out v_j of the adjacency-list of v_i. We then traverse the adjacency-list of v_j. For each neighbour v_y of v_j with $v_y \neq v_i$, we can check in constant time, if v_y is already included in the adjacency-list of the new vertex, i.e. we check whether $A(y) = 1$. If this is the case, we process the next neighbour, otherwise we include v_y into the adjacency-list of v_i. At the end, we traverse the adjacency-list of v_i once again to set all 1 to 0 in the array A. On vertices occurring more than once a degree update has to be done. Furthermore, the degree of the new vertex (also called v_i) has to be updated. All can be done in $O(d(v_i) + d(v_j))$ time.

Find a vertex of smallest degree.

Maintaining a pointer δB to the nonempty bucket B_i with smallest index i helps to find a vertex with minimum degree in constant time. Also this pointer may have to be updated when a vertex is deleted or an edge is contracted. Note that in either case the degree of any vertex can only decrease by one, and the number of vertices is also decreased by one. Assuming δB points to B_i, updating δB after a vertex deletion or edge-contraction can be done as follows. If $|B_{i-1}| \geq 1$ then we set δB to B_{i-1}, otherwise to B_k with $|B_k| \geq 1$, $i \leq k$ and k as small as possible. Hence, pointer δB can make at most n single steps to the left, i.e. from a B_i to a B_{i-1} (if the buckets are sorted as $B_0, ..., B_{n-1}$). Thus this pointer can make at most $2 \cdot n$ single steps to the right, and therefore altogether it needs $O(n)$ time for a sequence of n operations (vertex deletions or edge contractions). Hence, we have amortised time $O(1)$ per operation.

Find a vertex of second smallest degree.

To find a vertex with second smallest degree in constant time, we use a pointer $\delta_2 B$. Maintaining this pointer is slightly more complicated than maintaining δB and involves a case distinction. Note again that deleting a vertex or contracting an edge can only decrease the degree of any remaining vertex by at most one. Let $\delta_2 B$ point to B_j. We first update δB. Updating $\delta_2 B$ can be done as follows.
Case 1 'δB points to B_l with $l < j$':
Subcase 1a 'B_{j-1} is not empty': If δB points to B_{j-1} and $|B_{j-1}| = 1$ then we set $\delta_2 B$ to B_k with $|B_k| \geq 1$, $j \leq k$ and k as small as possible, otherwise we set $\delta_2 B$ to B_{j-1}.
Subcase 1b 'B_{j-1} is empty': We set $\delta_2 B$ to B_k with $|B_k| \geq 1$, $j \leq k$ and k as small as possible.
Case 2 'δB points to B_j': If $|B_j| \geq 2$ then we leave $\delta_2 B$ unchanged, otherwise we set $\delta_2 B$ to B_k with $|B_k| \geq 1$, $j < k$ and k as small as possible.
Case 3 'δB points to B_l with $j < l$': If $|B_l| \geq 2$, then we set $\delta_2 B$ to B_l, otherwise to B_k with $|B_k| \geq 1$, $l < k$ and k as small as possible.
Also this pointer makes at most $O(n)$ single steps to the left and hence it makes amortised $O(1)$ single steps per operation, for a sequence of n operations.

Lemma 55. *Let be given a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$. There exists a data structure, such that a sequence of $O(n)$ vertex deletions and searches for a vertex with smallest or second smallest degree costs $O(n + m)$ time.*

Proof. Using the data structure described in this section, the result follows directly from the analysis of the running time of the corresponding operations. □

A sequence of operations that includes edge-contractions can have an asymptotically larger running time when using the described data structure. Possibly, many ($\Omega(n)$) vertices and/or edges are inspected at each edge-contraction: concatenating two adjacency-lists and checking for double occurrences of vertices can cost $\Omega(n)$ time per edge contraction.

Another possibility to implement this data structure is to keep the adjacency-lists sorted, e.g. in nondecreasing order of the name (represented as a numerical index) of the vertices. The initial sorting can be done by a linear time sorting algorithm like e.g. radix sort. Sorted adjacency-lists will ease the implementation of the edge-contraction operation, because this could be done very similarly to the 'merge'-procedure of the 'merge'-sort algorithm (see e.g. [36]). Therefore, it is easy to see that the sorted adjacency-lists can be maintained by all the described operations within the same complexity bound.

Experimental Comparison

The data structure was implemented in C++ and experimentally evaluated on a PC with a 3.0 GHz Intel Pentium 4 processor running under Linux. However, our implementation does not strictly follow the description above. Instead, it uses and extends libraries and procedures provided by the Boost C++ libraries (especially the Boost Graph library; see [23] for more information on Boost). An implementation from scratch, i.e. not building upon existing libraries such as Boost, might yield different results and running times.

For comparison, we use exact algorithms and heuristics which are explained in more detail in Sections 4.2.1, 4.2.3 and 4.3.1. We tested each of the algorithms with the bucket adjacency-list and with a tree-based priority queue for the graph operations as used in e.g. [17, 18, 63]. In Table 4.1, we see the results of the δD, $\delta_2 D$ and δC lower bound algorithms and heuristics (see the columns labelled 'LB') for a selection of input graphs. We also see the running times (columns headed by 'CPU') obtained with a tree-based priority queue (see columns 'tree' in Table 4.1) and the bucket adjacency-list (see columns 'bucket' in Table 4.1).

instance	size		δD				$\delta_2 D$				δC(least-c)			
			tree		bucket		tree		bucket		tree		bucket	
	$\|V\|$	$\|E\|$	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU
link	724	1738	4	0.01	4	0.01	4	5.22	4	3.15	11	0.04	11	0.04
munin1	189	366	4	0.00	4	0.00	4	0.29	4	0.19	10	0.01	10	0.01
munin3	1044	1745	3	0.01	3	0.01	3	11.02	3	5.65	7	0.03	7	0.02
pignet2	3032	7264	4	0.05	4	0.03	4	108.91	4	64.28	38	0.21	38	0.22
celar06	100	350	10	0.00	10	0.00	11	0.09	11	0.09	11	0.01	11	0.00
celar07pp	162	764	11	0.00	11	0.01	12	0.30	12	0.24	15	0.01	15	0.01
graph04	200	734	6	0.01	6	0.01	6	0.42	6	0.31	19	0.02	20	0.02
rl5934-pp	904	1800	3	0.01	3	0.01	3	9.18	3	5.13	5	0.04	5	0.04
school1	385	19095	73	0.03	73	0.04	74	7.61	74	10.15	122	0.84	122	0.94
school1-nsh	352	14612	61	0.02	61	0.03	62	5.55	62	7.14	106	0.62	106	0.70
zeroin.i.1	126	4100	48	0.01	48	0.01	48	0.58	48	0.76	50	0.05	50	0.05

Table 4.1. Comparison of running times of basic algorithms with two different data structures: tree-based priority queue and bucket adjacency-list

The running times of the algorithm for computing δD are nearly equal for the two data structures. This is likely explained by the fact that these running times are very small, and hence, large absolute improvements are not possible and relative improvements are difficult to observe.

Similar behaviour can be observed in the columns representing the running times of the heuristic computing δC. Also these values are very small and a clear trend is difficult to be identified. It is interesting that a different lower bound value for ‘graph04’ is obtained with the two different data structures. This is possible, because the corresponding algorithm is a heuristic and does not always compute the exact value. Therefore, different data structures may lead to a different order in which the vertices are processed. This, in turn, can lead to different results.

The running times of the algorithm computing $\delta_2 D$ are the largest, and here, we can see the following trend (that can also be observed in the other columns to a much smaller degree): If the graph is rather sparse (like the first eight graphs in Table 4.1), then the implementations using the bucket adjacency-list gives smaller running times. On the other hand, if the graph is rather dense (as the last three graphs in Table 4.1), then the tree-based priority queue is faster.

4.2 Exact Algorithms for Some Treewidth Lower Bounds

Fortunately, not all parameters introduced in Section 3.1.1 are NP-hard to compute. (See Figure 3.8 for an overview over the computational complexity of the parameters.) For some of them, the exact value can be computed in polynomial or even linear time. In this section, we describe and analyse these polynomial time algorithms.

4.2.1 Algorithms for δ, δ_2 and δD

An implementation of algorithms to compute δ and δ_2 is straightforward. It is obvious that both parameters can be computed exactly in linear time. The degeneracy δD can be computed as described at the beginning of Section 3.1. Using the data structure, examined in Section 4.1, it is easy to see that the degeneracy can be computed in linear time $O(n+m)$.

4.2.2 Algorithms for γ_R

Ramachandramurthi shows in [76] that γ_R can be computed in $O(n+m)$ time. In our experiments, we use a different algorithm that does not use an adjacency-matrix. Our algorithm appears to be simpler and is easy to implement. Let a sequence $v_1, ..., v_n$ of the vertices of the graph be given, such that for all $i \in \{1, ..., n-1\}$, $d(v_i) \leq d(v_{i+1})$. Our algorithm and the algorithm in [76] are based on the fact that γ_R is determined by the leftmost vertex in this sequence that is not adjacent to all vertices to the left of it in the sequence (see Lemma 20). We have for each vertex v_i ($i \in \{1, ..., n\}$) a counter $c(v_i)$ that is initially 0. This counter $c(v_i)$ counts the number of neighbours of v_i that are left of v_i. Therefore, it is easy to find the first vertex with $i \neq c(v_i)+1$.

```
γR-Algorithm

1   obtain sequence $v_1, ..., v_n$ by bucket sorting
    the vertices according to nondecreasing degree
2   for all $j \in \{1, ..., n\}$
        initialise counter $c(v_j) := 0$
    endfor
3   $i := 1$
4   while $i = c(v_i) + 1$ and $i < n$ do
```

```
 5        for all $u \in N(v_i)$ do
 6            $c(u) := c(u) + 1$
 7        endfor
 8        $i := i + 1$
 9    endwhile
10    return $d(v_i)$
```

Note that after the loop of lines 5-7 is executed, we have for all $i \in \{1, ..., n\}$ that $c(v_i)$ equals the number of neighbours of v_i in $\{v_1, ..., v_i\}$. It is easy to see that the algorithm runs in $O(n + m)$ time, since for each vertex (line 4 to 9), we make at most one pass over its adjacency-list (line 5 to 7). Bucket sorting and initialising (line 1-3) also can be done in $O(n + m)$ time.

4.2.3 An Algorithm for $\delta_2 D$

First we note that the simple strategy to compute the degeneracy δD of a graph G as described in Section 3.1 cannot be used to compute $\delta_2 D$. This becomes evident when considering the example graph in Figure 4.2. There, we see that successively deleting a vertex of smallest degree (in the example of

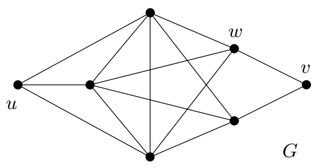

Figure 4.2. Counterexample to algorithm for computing $\delta_2 D$ by deleting smallest degree vertices

Figure 4.2 the vertex of smallest degree is v) does not lead to a subgraph where δ_2 is maximal, i.e. $\delta_2 D(G \setminus \{v\}) = 3$. Instead, we must delete vertex u (a vertex with second smallest degree), and we obtain $\delta_2 D(G \setminus \{u\}) = \delta_2 D(G) = 4$. One could now conjecture that deleting a vertex with second smallest degree might be correct, but it is also easy to find a counterexample for this approach.

One possible strategy to compute $\delta_2 D$ is as follows. We can fix a vertex v of which we suppose it will be the vertex of minimum degree in a subgraph G' of G with $\delta_2(G') = \delta_2 D(G)$. Starting with the original graph, we successively delete a vertex in $V(H) \setminus \{v\}$ of smallest degree, where H is the current considered subgraph of G (initially: $H = G$). Since we do not know whether our choice of v was optimal, doing this for all vertices $v \in V$ leads to a correct algorithm to compute $\delta_2 D(G)$. Using the bucket data structure, described above, this method can be implemented to take $O(n \cdot m)$ time. The following pseudo-code makes this algorithm more precise.

```
$\delta_2 D$-Algorithm

1    $delta2D := 0$
2    for each $v \in V$ do
3        $H := G$
4        repeat
5            if $\delta_2(H) > delta2D$ then $delta2D := \delta_2(H)$ endif
```

```
            $V^* := V(H) \setminus \{v\}$
            let $u \in \{w \in V^* \mid \nexists w' \in V^* : d_H(w') < d_H(w)\}$
            $H := H[V(H) \setminus \{u\}]$
        until $|V(H)| = 1$
    endfor
    return $delta2D$
```

Lemma 56. *The $\delta_2 D$-*`Algorithm` *computes the parameter $\delta_2 D(G)$ and can be implemented to run in $O(n \cdot m)$ time for any given connected graph $G = (V, E)$ with $|V| \geq 2$.*

Proof. First, we will show that the returned value of $delta2D = \delta_2 D(G)$. Note that every H considered in the algorithm is a subgraph of G. Therefore, it is easy to see that $delta2D \leq \delta_2 D(G)$, since:

$$delta2D = \max_H \{\delta_2(H) \mid H \text{ occurs during the run of the algorithm}\}$$

Now, we show that there is a subgraph $H \subseteq G$ considered during the algorithm with $\delta_2(H) = \delta_2 D(G)$. Let $G' = (V', E') \subseteq G$ with $\delta_2(G') = \delta_2 D(G)$ be given. Furthermore, let $v \in V'$ be a vertex of minimum degree in G'. We consider the run of the for-loop of the $\delta_2 D$-`Algorithm`, where v was chosen (in Line 2) to always remain in the graph. Note that the algorithm selects and deletes successively a vertex $u \neq v$ whose degree is as small as possible. Now, consider the first time when in the repeat-loop, i.e. in the current graph H, a vertex $u \in V'$ is selected to be deleted. Because u is the first such vertex, we have $G' \subseteq H$. Therefore, for all $w \in V' \setminus \{v\}$, we have $\delta_2(G') \leq d_{G'}(w) \leq d_H(w)$. Hence, since $u \in V' \setminus \{v\}$, it holds that $\delta_2(G') \leq d_H(u)$. Because u is a vertex in $V^* = V(H) \setminus \{v\}$ with degree in H as small as possible, all vertices in V^* have degree at least $d_H(u) \geq \delta_2(G')$. Therefore, we have $\delta_2(H) \geq \delta_2(G') = \delta_2 D(G)$. Hence, the algorithm considers a graph $H \subseteq G$ with $\delta_2(H) = \delta_2 D(G)$. This proves our initial claim $delta2D = \delta_2 D(G)$.

It is not difficult to see that the algorithm uses $O(n \cdot m)$ time when we use the data structure described in Section 4.1. □

4.3 Heuristics

In this section, we give heuristics for the treewidth lower bound parameters from Chapter 3 that are NP-hard to compute. Some of these heuristics are build upon the algorithms given in Section 4.2. Each heuristic gives a lower bound for treewidth. We describe the heuristics, and in some cases, give some analysis of them. In Section 4.3.1 we propose and analyse some heuristics for the contraction degeneracy δC. Some heuristics for computing $\delta_2 C$, $\gamma_R D$ and $\gamma_R C$ are given in Sections 4.3.2 to 4.3.4. In Section 4.3.5, we discuss heuristics for $MCSLBC$. In Section 4.3.6, we look at the LBN and LBP heuristics. These can be combined with any of the other heuristics, but we also propose a new heuristic where contractions alternate with constructions of neighbours or paths improved graphs. An experimental evaluation of the heuristics is given in Section 4.4.

4.3.1 δC-Heuristics

An almost trivial heuristic for the contraction degeneracy is the degeneracy $\delta D(G)$. As mentioned earlier, it can be computed in linear time, by iteratively selecting a vertex of minimum degree, and deleting it and its incident edges. The largest minimum degree encountered in these steps is the degeneracy.

From this algorithm, we derive three δC heuristics. In these heuristics, we select a vertex v of minimum degree and instead of deleting it, we contract it with one of its neighbours u. In each case, the heuristic outputs the maximum over all vertices of its degree when it was selected as minimum degree vertex. (Note that, sometimes, the δD algorithm is called MMD, which stands for 'Maximum Minimum Degree', and the δC heuristics are sometimes called MMD+.) Clearly, this is a lower bound on the contraction degeneracy of a graph. We consider three strategies how to select a neighbour:

- *min-d* selects a neighbour with minimum degree. This heuristic is motivated by the idea that the smallest degree is increased as fast as possible in this way.
- *max-d* selects a neighbour with maximum degree. This heuristic is motivated by the idea that we end up with some vertices of very high degree.
- *least-c* selects a neighbour u of v, such that u and v have the least number of common neighbours. Note that for each common neighbour w of u and v, the two edges $\{u, w\}$ and $\{v, w\}$ become the same edge in the graph after contracting $\{u, v\}$, meaning that for each common neighbour, effectively one edge is removed from the graph. Thus, the least-c heuristic is motivated by the idea to delete as few edges as possible in each iteration in order to get a high minimum degree.

We call the resulting heuristics δC(min-d), δC(max-d) and δC(least-c). We observe that each of the δC heuristics gives a value that is at least the degeneracy: consider a subgraph H of G with minimum degree the degeneracy of G. Consider the graph G' that we currently have just before the first vertex v from H is to be contracted by the heuristic. All neighbours of v in H are also neighbours of v in G', hence the algorithm gives as bound at least the degree of v in H, hence at least the degeneracy of G.

In a recent paper, Gogate and Dechter [50] propose a branch-and-bound algorithm for treewidth with a good anytime performance. (An anytime algorithm is an algorithm for an optimisation problem that, when stopped at any time, returns the best solution found so far. The more time the algorithm is given, the better the result. In that sense, a branch-and-bound algorithm is always an anytime algorithm.) Independently of this work, Gogate and Dechter also propose the lower bound heuristic which we call the δC(min-d) heuristic. We compare this heuristic with our other heuristics and see that the δC(least-c) heuristic almost always outperforms the δC(min-d) heuristic. For more details, see [50], Section 3.1 and Section 4.4.3.

While the heuristics (and especially the least-c heuristic) do often well in practice, unfortunately, each of the heuristics can do arbitrarily bad. In the next four paragraphs, we give examples of graphs where there is a large difference between the contraction degeneracy and a possible lower bound for it obtained by one of the four considered heuristics described above.

A bad example for the δD heuristic

The δD algorithm can perform arbitrarily bad when considered as a δC heuristic. To see this, take a clique with n vertices, and then subdivide every edge. Let G be the resulting graph. Clearly, $\delta(G) = 2$. We also have $\delta D(G) = 2$ since all subdivisions have degree 2 and must be deleted, which also deletes all edges in G. However, $\delta C(G) = n - 1$, because undoing the subdivisions results in an n-clique with minimum degree $n - 1$.

A bad example for the δC(max-d) heuristic

An example where the δC(max-d) heuristic can perform arbitrarily bad is not hard to find. One simple example is the following. Take a clique with n vertices, subdivide every edge, and then add one universal vertex x. Let G be the resulting graph. The δC(max-d) heuristic will contract each vertex to x, and hence will give 3 as a result. However, $\delta C(G) = n$, since if we undo the subdivisions by contracting the subdivision vertices to the clique vertices, we obtain a clique with $n + 1$ vertices.

A bad example for the δC(min-d) heuristic

The example where the δC(min-d) heuristic performs bad is somewhat more involved. For each r, we build a graph where the δC(min-d) heuristic can possibly give a lower bound of 3, while the contraction degeneracy of the graph is r. We assume, as 'adversary', that we can decide in which way tie-breaking is done (i.e. the adversary can select a vertex among those which have minimum degree).

Let $r \geq 3$ be some integer. Build a graph G_r as follows. We take for each i, j, $1 \leq i < j \leq r$ a vertex v_{ij}. We take for each i, $1 \leq i \leq r$ a vertex w_i, and we take another vertex x. Now, we add edges $\{v_{ij}, w_i\}$, $\{v_{ij}, w_j\}$ and $\{v_{ij}, x\}$, for each i, j, $1 \leq i < j \leq r$.

To the graph thus obtained, we add a number of cliques. Each clique consists of three new vertices and one vertex of the type w_i, $1 \leq i \leq r$ or x. We have one such clique that contains x. For each i, we take r^2 such cliques that contain w_i, $1 \leq i \leq r$. (It is possible to make a more compact construction, using about $r^2/6$ cliques.) We call the new vertices in these cliques the *additional clique vertices*. In this way, each w_i has a degree that is larger than $3r^2$. Let G_r be the resulting graph, see Figure 4.3.

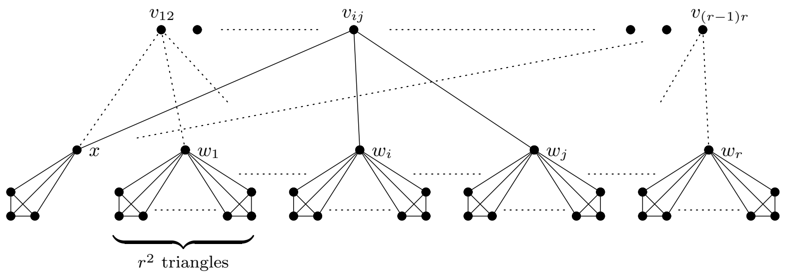

Figure 4.3. The structure of G_r

Lemma 57. *Let $r \geq 3$. The contraction degeneracy and treewidth of G_r both equal r.*

Proof. If we contract v_{1r} to w_r and each other vertex of the form v_{ij} to w_i, and each additional clique vertex to its neighbour of type w_i or x, then the resulting graph is a clique on $\{w_i \mid 1 \leq i \leq r\} \cup \{x\}$. Each vertex in this clique has degree r, so the contraction degeneracy of G_r is at least r, and hence the treewidth of G_r is at least r.

If we add to G_r an edge between each distinct pair of vertices in $\{w_i \mid 1 \leq i \leq r\} \cup \{x\}$, then we obtain a chordal graph with maximum clique size $r + 1$. So, the treewidth of G_r is at most r, and hence also its contraction degeneracy is at most r. □

Lemma 58. *The δC(min-d) heuristic can give a lower bound of 3 when applied to G_r.*

Proof. Consider the following start of a sequence of contractions: first, contract the vertices of the form v_{ij} one by one to x, for all i, j, $1 \leq i < j \leq r$. Note that the δC(min-d) heuristic can start with this sequence: at each point during this phase, the vertices of the form v_{ij} have degree 3, which is the minimum degree in the graph, and the degree of x is at most $r(r-1)/2+3+r$, which is less than the degree of vertices of the form w_i, which have degree at least $3r^2$. So, during this first part of the running of the algorithm, the bound for the contraction degeneracy is not larger than 3.

After all vertices v_{ij} have been contracted to x, the graph has the following form: x is adjacent to all w_i; there are no edges between vertices $w_i, w_{i'}, i \neq i'$; and then there are a number of four-cliques that have one vertex in common with the rest of the graph. This is a chordal graph with maximum clique size 4. So, this graph has treewidth 3, and hence contraction degeneracy at most 3. Hence, the min-d heuristic cannot give a bound larger than 3 in the remainder of the run of the algorithm. Thus, the maximum bound it obtains can be 3. □

Corollary 59. *The δC(min-d) heuristic can give a solution that is a factor of $\Omega(\sqrt{n})$ away from optimal.*

We can use cliques with four instead of three additional clique vertices. In that case, it holds that every possible run of the δC(min-d) heuristic gives a lower bound of 4 on the graph.

A bad example for the δC(least-c) heuristic

The example for the δC(least-c) heuristic is a modification of the one for the δC(min-d) heuristic.

Let $r \geq 3$ again be an integer. We take G_r and modify it as follows. Each edge of the form $\{v_{ij}, w_i\}$ or $\{v_{ij}, w_j\}$ is replaced by the gadget given in Figure 4.4. In words: the edge is subdivided, and we add a clique with three new vertices and the subdivision vertex to the graph. Let G'_r be the resulting graph.

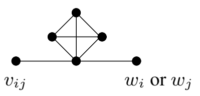

Figure 4.4. The gadget that replaces edges of the form $\{v_{ij}, w_i\}$ or $\{v_{ij}, w_j\}$

Lemma 60. *Let $r \geq 3$. The contraction degeneracy and treewidth of G'_r both equal r.*

Proof. We can contract G'_r to G_r: contract each structure as in Figure 4.4 to the vertex of the form v_{ij}. So, the contraction degeneracy of G'_r is at least the contraction degeneracy of G_r, hence at least r. Therefore, the treewidth of G'_r is at least r.

The treewidth of G'_r is at most r: add to G'_r edges between each pair of distinct vertices in $\{w_i \mid 1 \leq i \leq r\} \cup \{x\}$. Then, for each i, j, $1 \leq i < j \leq r$, add edges $\{v_{ij}, w_i\}$ and $\{v_{ij}, w_j\}$. This gives a chordal graph with maximum clique size $r+1$. So, the treewidth of G'_r is at most r. Hence, its contraction degeneracy is also at most r. □

Lemma 61. *The δC(least-c) heuristic can give a bound of 3 when applied to G'_r.*

Proof. Like for the δC(min-d) heuristic, the algorithm can start with contracting each vertex of the form v_{ij} to x. During this phase, vertices v_{ij} have the minimum degree in the graph, namely 3, and have no common neighbours with x. So, during this phase, the lower bound is set to 3.

After all vertices of the form v_{ij} are contracted to x, the graph G'' has treewidth 3. This can be seen as follows. The treewidth of a graph is the maximum treewidth of its biconnected components.

The biconnected components of G'' are either cliques with four vertices, single edges, or consist of x, a vertex w_i, and a number of paths of length 2 between x and w_i (for some i, $1 \leq i \leq r$). In each of the cases, the treewidth of the component is at most 3. So, after the contractions of the v_{ij}-vertices to x, the bound of 3 cannot be increased. □

Corollary 62. *The δC(least-c) heuristic can give a solution that is a factor of $\Omega(\sqrt{n})$ away from optimal.*

It is possible to modify the construction such that any run of the δC(least-c) heuristic gives a result far from optimal. Instead of cliques with three new vertices and one 'old' vertex, we use cliques with five new vertices and one old vertex. The structure of Figure 4.4 is replaced by the structure of Figure 4.5. In this way, we obtain a graph that has contraction degeneracy r, but for which any run of the δC(least-c) heuristic gives a lower bound that is at most 5.

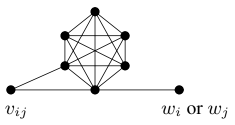

Figure 4.5. An alternative structure that replaces edges of the form $\{v_{ij}, w_i\}$ or $\{v_{ij}, w_j\}$

4.3.2 $\delta_2 C$-Heuristics

For computing $\delta_2 C$, we develop three heuristic algorithms. The first one is called $\delta_2 C$(all-v), and it is based on the polynomial time implementation for $\delta_2 D$. We fix all vertices once at a time and perform the δC(least-c) heuristic on the rest of the graph. The best second smallest degree recorded provides a lower bound on $\delta_2 C$. (We only tested this with the least-c strategy for selecting which neighbour to contract to, because it seems to be preferable over the other strategies.)

The other two $\delta_2 C$-heuristics are based on the heuristics for δC. Instead of recording the minimum degree, we also can record the second smallest degree (*m*aximum *s*econd *d*egree with contraction, abbreviated msd). If we contract a vertex of minimum degree with one of its neighbours (according to the least-c strategy), we obtain the heuristic $\delta_2 C$(msd1). If the vertex of second smallest degree is contracted with one of its neighbours (also according to the least-c strategy), we obtain the heuristic $\delta_2 C$(msd2).

4.3.3 $\gamma_R D$-Heuristics

For the parameter $\gamma_R D$, we develop three heuristics based on the following observation: Let $v_1, ..., v_n$ be a sorted sequence of the vertices in order of nondecreasing degree in G, and let $\gamma_R(G)$ be determined by v_j for some $j > 1$ (see Lemma 20 and the description of the algorithm to compute γ_R). Thus, v_j is not adjacent to some vertex v_k with $k < j$, whereas $v_1, ..., v_{j-1}$ induce a clique in G. Let V^* be the set of all vertices v_i with $i < j$ and $\{v_i, v_j\} \notin E$. Now, for any subgraph $G' \subset G$ with $(\{v_j\} \cup V^*) \subseteq V(G')$, we have that $\gamma_R(G') \leq \gamma_R(G)$. Hence, only subgraphs without either v_j or V^* are of further interest. Based on this observation, we implemented three heuristics. In the heuristic $\gamma_R D$(left), we remove the vertices in V^* (the vertices that are more to the left in the sequence) from the graph and continue. Whereas in the heuristic $\gamma_R D$(right), we delete the vertex v_j (the vertex that is more to the right in the sequence) and go to the next iteration. The heuristic $\gamma_R D$(min-e) (minimum number of edges) chooses to remove either V^* or v_j depending on which of the two deletes fewer edges to obtain an induced subgraph with as many edges as possible.

4.3.4 $\gamma_R C$-Heuristics

For $\gamma_R C$ the same strategies as for $\gamma_R D$ have been implemented. The only difference is that instead of removing all vertices in V^* or v_j, we contract each of the vertices with a neighbour that is selected according to the least-c strategy. (Again, we only use the least-c strategy for selecting a neighbour, since this strategy appears to perform very well.) Inspired by the good results of the $\delta_2 C$(all-v) heuristic, we furthermore implemented the all-v strategy as described above also for the γ_R-contraction degeneracy. The difference is that instead of computing δ_2 of each obtained minor, we now compute γ_R.

4.3.5 $MCSLB$ and $MCSLBC$-Heuristics

Based on the result by Lucena [70] that the visited degree of an MCS ordering of G is a lower bound for the treewidth (see Theorem 46), we will consider heuristics for computing the parameter $MCSLBC$, which is also a treewidth lower bound (see Lemma 49).

For the sake of comparison, we consider a $MCSLB$-heuristic. This heuristic computes $|V|$ MCS orderings ψ – one for each vertex as start vertex. It returns the maximum over these orderings of $MCSLB_\psi$. See [14] for more details on the $MCSLB$ treewidth lower bound.

The $MCSLBC$ heuristic starts by using the $MCSLB$ heuristic to find a start vertex w with largest $MCSLB_\psi$. To reduce the CPU time consumption, an MCS is carried out only with start vertex w (or vertices resulting from contractions that involve w) instead of with all possible start vertices. Then, we iteratively select a vertex and a neighbour to contract, compute an MCS ordering and repeat until the graph has no edges. Three strategies for selecting a vertex v to be contracted are examined:

- *min-deg* selects a vertex of minimum degree.
- *last-mcs* selects the last vertex in the just computed MCS ordering.
- *max-mcs* selects a vertex with maximum visited degree in the just computed MCS ordering.

Once a vertex v is selected, we select a neighbour u of v using the two strategies min-d and least-c that are already explained for the δC heuristics. We thus have six versions of the $MCSLBC$ heuristic (e.g. the $MCSLBC$(max-mcs least-c) heuristic). These are experimentally evaluated in Section 4.4.5. We do not evaluate the $MCSLBC$(* max-d) heuristic, because of the negative experimental results for the δC(max-d) strategy.

4.3.6 LBN and LBP Heuristics

In Section 3.3, we have seen that for each treewidth lower bound algorithm or heuristic Y, we have two lower bound heuristics LBN(Y) and LBP(Y), based on the technique by Clautiaux et al. [32]. An example is the LBN(δC(least-c)) heuristic.

A different method to combine the LBN or LBP methods with contraction is to alternate improvement steps with contractions. We describe the LBN+(Y) algorithm for some treewidth lower bound heuristic Y below. If we instead of making a neighbours improved graph, take a paths improved graph, we obtain the LBP+(Y) algorithm; the latter one is slower but often gives better bounds.

LBN+(Y)-Algorithm

```
1   Initialise $L$ to some lower bound on the treewidth of $G$
    (e.g. $L := 0$)
2   $H := G$
3   repeat
4       $H :=$ the $(L+1)$-neighbours improved graph of $H$
5       $b := Y(H)$
6       if $b > L$
7       then $L := L+1$
8            goto step 2
9       else select a vertex $v$ of minimum degree in $H$
10           select a neighbour $u$ of $v$
             according to the least-c strategy
11           contract the edge $\{u,v\}$ in $H$
12      endif
13  until $H$ is empty
14  output $L$
```

In the description above, we used the least-c strategy, as this strategy performed best for the other heuristics; of course, variants with other contraction strategies can also be considered.

Lemma 63. *If Y is an algorithm that outputs a lower bound on the treewidth of its input graph, then LBN+(Y) and LBP+(Y) output lower bounds on the treewidth of their input graph.*

Proof. Let G be the input graph of algorithm LBN+(Y) or LBP+(Y). An invariant of the algorithm is that the treewidth of G is at least L. A second invariant of the algorithm is that when the treewidth of G equals L, then the treewidth of H is at most L. Clearly, these invariants hold initially. Lemmas 53 and 54 show that the second invariant holds also after making an improved graph in step 4. The fact that contraction cannot increase treewidth shows that the second invariant holds after a contraction in step 11. Similar as in [32], when Y outputs a value larger than L on H, then the treewidth of H and hence the treewidth of G (by the second invariant) is larger than L, so increasing L by one in step 7 maintains the first invariant. □

In our experiments, we started the algorithm by setting the lower bound L to the value computed by the δC(least-c) heuristic.

Faster Implementation of LBP+

A straightforward implementation of an LBP+(Y) heuristic can be very slow. However, we can observe that some steps are not necessary. Contracting an edge can increase the number of vertex-disjoint paths between two vertices, but not for all pairs of vertices. Lemma 64 tells us that contracting an edge $\{x,y\}$ cannot increase the number of vertex-disjoint paths between u and v, if $\{x,y\} \cap \{u,v\} = \emptyset$.

Lemma 64. *Let vertices u and v and edge $e = \{x,y\}$ in $G = (V,E)$ be given. Furthermore, let N be the maximum number of vertex-disjoint paths between u and v in G, and let N' be the maximum number of vertex-disjoint paths between u and v in $G/\{x,y\}$. Then we have:*

$$\{x,y\} \cap \{u,v\} = \emptyset \Longrightarrow N' \leq N$$

Proof. Let a_e be the new vertex created by contracting edge e. We consider a set P' of vertex-disjoint paths $p_1, ..., p_{N'}$ between u and v in G/e. Since these paths are vertex-disjoint and $\{x, y\} \cap \{u, v\} = \emptyset$, there can be at most one path p' in P' going through the new vertex a_e, i.e. a_e is contained in at most one path p' of P'.

One easily sees that there is a path p in G between u and v that uses all vertices of p' except a_e and x and/or y. Therefore, we have a set P of N' vertex-disjoint paths between u and v in G. Hence, $N' \leq N$. □

In other words, the number of vertex-disjoint paths between u and v can be increased by an edge contraction, only if an edge incident to u or v is contracted. A consequence of this is that after contracting edge e which results in a new vertex a_e, we only have to look for the number of vertex-disjoint paths of pairs of vertices that contain a_e. This results in a drastic speed up compared to the case when checking all pairs of vertices for $L+1$ vertex-disjoint paths, as we check $O(n)$ pairs instead of $\Theta(n^2)$ pairs. However, once we have found an improvement edge in the graph, we then must check all other pairs, as possibly, after an improvement edge is added, pairs of vertices that do not contain a_e can have $L+1$ vertex-disjoint paths.

LBN+(δ) versus LBN+(δD)

We now compare the LBN+(δ) heuristic with the LBN+(δD) heuristic, and similarly, the LBP+(δ) heuristic with the LBP+(δD) heuristic. In the LBN+(δ) heuristic, we just use the minimum degree of a vertex in H as lower bound, while in the LBN+(δD) heuristic, we use the degeneracy.

The intuitive idea is that LBN+(δ) and LBN+(δD) give the same result, because due to the additional contraction step, the subgraphs considered by the δD algorithm, will also be considered in the LBN+(δ) heuristic; similarly for the version with paths improvement. Below, we show that this intuition is correct.

Lemma 65. *Let $G = (V, E)$ be a graph. Let the results of running the LBN+(δ), LBN+(δD), LBP+(δ) and LBP+(δD) heuristics on G be α_n, β_n, α_p and β_p, respectively. Then $\alpha_n = \beta_n$ and $\alpha_p = \beta_p$.*

Proof. The proof is the same for the versions with neighbours and paths improvement. Thus, in the proof below, we write LBX+(δ) and LBX+(δD), where the X can stand for N or P, and we drop the subscripts n and p from α and β.

First note that when the LBX+(δ) and LBX+(δD) enter the loop at step 3 with the same value of L, then they will work with the same graph H. Thus, we have that $\alpha \leq \beta$: when LBX+(δ) increases L by one, we have that L is smaller than the minimum degree of H, hence also smaller than the degeneracy of H, and hence the LBX+(δD) algorithm will also increase L by one at the corresponding point during the execution of the algorithms. To show equality, we assume the following and derive a contradiction:

$$\alpha < \beta \tag{4.1}$$

Consider the moments step 2 is done by algorithm LBX+(δD) and by algorithm LBX+(δ) when $L = \alpha$. As LBX+(δ) outputs α, this is the last time step 2 is done by the LBX+(δ) algorithm, while the LBX+(δD) algorithm will increase L further (as $\alpha < \beta$), and hence will execute later the 'goto step 2' command at least once.

Let H^* be the graph H at the moment the LBX+(δD) algorithm is at step 7 and 8 when the algorithm increases L from α to $\alpha+1$. This graph H^* is formed from G by a sequence of contractions and $(\alpha+1)$-neighbours or $(\alpha+1)$-paths improvement steps. As the test in step 6 was true, the degeneracy of H^* is at least $\alpha+1$.

The LBX+(δ) algorithm has started a run of the main iteration with $L = \alpha$. As the algorithm outputs α, this is its last iteration. During this iteration, it does the same improvement steps as the LBX+(δD) algorithm, and hence, at some point, creates the graph H^*. However, it cannot execute steps 7 and 8 now, so the test in step 6 was false for the LBX+(δ) algorithms. Thus, we have:

$$\delta(H^*) \leq \alpha < \delta D(H^*)$$

Write $d = \delta D(H^*)$. Therefore, there exists an induced subgraph $H' \subset H^*$ with

$$\delta(H^*) < \delta(H') = d$$

Note that all vertices in $V(H')$ have degree at least $d := \delta D(H^*)$ in H^*. We now consider the execution of LBX+(δ), starting when H is the graph H^*, up to just before the point that the first vertex from H' is selected as minimum degree vertex v in step 9. During this part of the execution, we have that H' is a subgraph of the graph H used by the algorithm: improvement steps only add edges, and no edges between vertices in H' are contracted.

Now, consider the first vertex v^* from H' that is selected as minimum degree vertex v in step 9 by LBX+(δ). As H' is a subgraph of the graph H, we have that the degree of v^* at the moment it is selected is at least its degree in H', which is at least d. But, as v^* is the minimum degree of a vertex in H, all vertices in H have at this point degree at least d. This gives a contradiction. Consider the test at step 6 just before v^* was selected: the minimum degree of H is at least d, which is larger than the current value of L, i.e. α. So, this test is true, and the algorithm will increase L, contradiction.

So, we can conclude that the assumption $\alpha < \beta$ is false, hence $\alpha = \beta$. □

Whether in practice LBX+(δ) is more time-efficient than LBX+(δD) for a given graph, is unclear from the above. Computing the lower bound δD is more time-consuming than computing δ, but can result in a $b > L$ earlier during the contraction process, in this way avoiding a number of graph improvement steps. By Lemma 65, the number of graph improvement steps in the last iteration will be equal, slowing down the algorithm on this point.

Similarly LBX+(δC(*)) can be more time-efficient than LBX+(δ): δC(*) is more time-consuming but can reduce the number of improvement steps. Moreover, LBX+(δC(*)) can return a better bound than LBX+(δ), although this rarely happens. Experimental results with LBX+(δ), LBX+(δD) and LBX+(δC(*)) have shown that the computation times of LBX+(δ) are often significantly smaller than those for LBX+(δD) which in turn are significantly smaller than those for LBX+(δC(*)).

4.4 Experiments

In this section, we report on the results of the computational experiments we have carried out. In the tables in the subsequent sections, the results are presented for selected instances only. The result of these representative instances reflect typical behaviour for the whole set of instances. Experimental results for more graphs can be found in e.g. [17, 18, 93]. In some tables, we give the size of the graphs, i.e. the number of vertices and the number of edges. The best known upper bound for treewidth (see [63, 31, 50, 93]) is reported in the column with title UB. Columns headed by LB give treewidth lower bounds in the terms of the according parameter or a lower bound for the parameter. Values in columns headed by CPU are running times in seconds. Note that all algorithms and heuristics for computing the parameters δ, δ_2, γ_R and their D and C variants were implemented using the data structure explained in Section 4.1. All other heuristics use a tree-based priority queue implementation. However, before looking at the results, we describe how the experiments were carried out.

4.4.1 Experimental Setup and Input Graphs

The algorithms and heuristics described in Sections 4.2 and 4.3 have been tested on a large number of graphs from various application areas. The first set of instances consists of probabilistic networks from existing decision support systems from fields like medicine and agriculture (see Section 1.1 for more details on how tree-decompositions of probabilistic networks are used). The second set consists of instances from frequency assignment problems from the EUCLID CALMA project. In [62, 64], tree-decompositions were used to solve the frequency assignment problem on many of the networks from this collection of instances. In addition, we use versions of the networks, obtained by preprocessing [16]. A third set of instances is taken from the work of Cook and Seymour [33]. They present a heuristic for the travelling salesman problem (TSP) where they use branch-decompositions (a notion strongly related to tree-decompositions) of graphs formed by merging a number of TSP-tours. Finally, we computed the lower bounds for many of the DIMACS colouring instances [44].

Many of the tested graphs as well as most of their experimental results on treewidth can be obtained from [93]. As mentioned above, we only give the results for a selection of twelve graphs. These graphs were selected in a way such that at least one graph of every set of instances belongs to the selection. Furthermore, we tried to select graphs with interesting behaviour in order to show that some algorithms or heuristics do or do not work well on some instances.

All algorithms and heuristics were been written in C++, and the computations were carried out on a PC with a 3.2 GHz Intel Pentium 4 HT processor with 2 GB main memory operating under Linux.

4.4.2 Results for δ, δ_2, γ_R, δD, $\delta_2 D$ and $\gamma_R D$

Table 4.2 shows the sizes of the graphs, treewidth upper bounds and the results obtained from the treewidth lower bound algorithms without edge-contraction. These bounds are the exact parameters apart from the values for the three $\gamma_R D$ heuristics. As the computation times for δ, δ_2 and γ_R are negligible, we omit them in the table. Also the δD can be computed within a fraction of a second. The computational complexity of $\delta_2 D$ is a factor of $O(n)$ larger than the one of δD which is reflected in the CPU times for this parameter.

instance	size		UB	δ	δ_2	γ_R	δD		$\delta_2 D$		$\gamma_R D$					
											left		right		min-e	
	$\|V\|$	$\|E\|$		LB	LB	LB	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU
link	724	1738	13	0	0	0	4	0.01	4	3.67	4	0.01	4	0.01	4	0.01
munin1	189	366	11	1	1	1	4	0.00	4	0.23	4	0.00	4	0.00	4	0.00
munin3	1044	1745	7	1	1	1	3	0.01	3	6.70	3	0.02	3	0.01	4	0.01
pignet2	3032	7264	135	2	2	2	4	0.04	4	69.87	4	0.04	4	0.05	4	0.04
celar06	100	350	11	1	1	1	10	0.01	11	0.08	11	0.00	10	0.00	10	0.00
celar07pp	162	764	16	3	3	3	11	0.01	12	0.27	12	0.00	11	0.01	11	0.00
graph04	200	734	51	3	3	3	6	0.01	6	0.36	6	0.00	6	0.00	6	0.01
queen15-15	225	5180	163	42	42	42	42	0.01	42	0.87	42	0.00	42	0.01	42	0.01
school1	385	19095	188	1	1	1	73	0.04	74	7.89	75	0.03	73	0.03	73	0.03
school1-nsh	352	14612	162	1	1	1	61	0.02	62	5.69	62	0.03	61	0.02	61	0.03
zeroin.i.1	126	4100	50	28	29	32	48	0.00	48	0.58	50	0.01	50	0.01	50	0.01
rl5934-pp	904	1800	21	3	3	3	3	0.01	3	5.33	3	0.01	3	0.01	3	0.01

Table 4.2. Graph sizes, treewidth upper bounds, treewidth lower bounds and running times of the algorithms and heuristics without edge-contraction

The results of the algorithms and heuristics without edge-contractions (Table 4.2) show that the degeneracy lower bounds (i.e. the lower bounds involving subgraphs) are significantly better than the simple lower bounds, as could be expected. Comparing the results for δD and $\delta_2 D$, we see that in four cases we have that $\delta_2 D = \delta D + 1$. In the other seven cases $\delta_2 D = \delta D$. Bigger gaps than one between δD and $\delta_2 D$ are not possible (see Lemma 19). In some cases other small improvements (compared to δD and $\delta_2 D$) could be obtained by the heuristics for $\gamma_R D$. The three $\gamma_R D$-heuristics are all comparable in value and running times. Apart from the running times for computing $\delta_2 D$, the computation times are in all cases marginal, which is desirable for methods involving computing lower bounds many times (e.g. branch-and-bound). Even though the $\delta_2 D$ algorithm has much higher running times than the other algorithms in Table 4.2, it is still much faster than some heuristics with contraction. Furthermore, we expect that its running time could be improved by a more efficient implementation. We do not further investigate these parameters without contraction, as the parameters with contraction are of considerably more interest.

4.4.3 Results for δC

In Table 4.3, we again include the best known upper bounds for the sake of comparison. Furthermore, for an easy comparison between the δD algorithm and the different δC heuristics, we also present the results of the δD algorithm.

instance	size		UB	δD		δC					
						min-d		max-d		least-c	
	$\|V\|$	$\|E\|$		LB	CPU	LB	CPU	LB	CPU	LB	CPU
link	724	1738	13	4	0.00	8	0.01	5	0.01	11	0.03
munin1	189	366	11	4	0.00	8	0.00	5	0.01	10	0.00
munin3	1044	1745	7	3	0.01	6	0.01	4	0.01	7	0.02
pignet2	3032	7264	135	4	0.01	28	0.10	10	0.07	38	0.10
celar06	100	350	11	10	0.00	11	0.00	10	0.01	11	0.00
celar07pp	162	764	16	11	0.00	13	0.00	12	0.00	15	0.01
graph04	200	734	51	6	0.01	12	0.01	7	0.00	20	0.01
queen15-15	225	5180	163	42	0.01	52	0.06	42	0.02	58	0.10
school1	385	19095	188	73	0.02	98	0.19	74	0.09	122	0.47
school1-nsh	352	14612	162	61	0.01	81	0.13	61	0.06	106	0.33
zeroin.i.1	126	4100	50	48	0.01	50	0.02	48	0.01	50	0.03
rl5934-pp	904	1800	21	3	0.00	5	0.02	4	0.01	5	0.02

Table 4.3. Graph sizes, upper bounds and results of the δD algorithm and the δC heuristics

We can see from these results that edge-contraction is a very powerful method for obtaining lower bounds for treewidth. The improvements obtained by using the δC heuristics instead of the δD algorithm are in many cases quite significant, and they show the impressive impact of combining edge-contraction with treewidth lower bounds.

Concerning the different strategies for δC, we can observe that the least-c strategy is best. In many cases, it performs much better than the other two strategies, and in all our experiments, there are only very rare cases where its bound is not the best amongst the three strategies for computing δC. The max-d strategy appears to do bad, giving in general much smaller lower bounds than the other two. Thus, we did not use this strategy as a part or subroutine for other heuristics.

4.4.4 Results for $\delta_2 C$ and $\gamma_R C$

Table 4.4 shows the results of the $\delta_2 C$ and $\gamma_R C$ heuristics. Furthermore, we give the results of the δC(least-c) heuristic for easy comparison.

instance	δC		$\delta_2 C$						$\gamma_R C$							
	least-c		all-v		MSD+1		MSD+2		left		right		min-e		all-v	
	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU
link	11	0.03	11	13.37	11	0.02	11	0.02	11	0.02	12	0.02	11	0.02	11	106.26
munin1	10	0.00	10	0.51	10	0.00	10	0.01	9	0.00	10	0.00	10	0.00	10	2.43
munin3	7	0.02	7	11.52	7	0.01	7	0.01	7	0.01	7	0.01	7	0.02	7	294.07
pignet2	38	0.10	40	300.05	39	0.11	39	0.13	38	0.11	39	0.11	39	0.11	40	7962.11
celar06	11	0.00	11	0.15	11	0.00	11	0.00	11	0.00	11	0.00	11	0.00	11	0.46
celar07pp	15	0.01	15	0.63	15	0.01	15	0.00	15	0.01	15	0.00	15	0.00	15	1.86
graph04	20	0.01	21	2.20	20	0.01	19	0.01	20	0.02	19	0.01	20	0.01	21	4.52
queens15-15	58	0.10	59	22.89	58	0.11	59	0.10	59	0.10	60	0.09	59	0.09	60	28.36
school1	122	0.47	124	181.73	123	0.48	122	0.47	122	0.46	122	0.52	122	0.45	125	214.83
school1-nsh	106	0.33	108	110.34	106	0.34	107	0.33	104	0.33	106	0.33	106	0.30	108	130.71
zeroin.i.1	50	0.03	50	4.14	50	0.03	50	0.04	50	0.02	50	0.03	50	0.04	50	5.68
rl5934-pp	5	0.02	6	20.25	5	0.02	5	0.02	5	0.03	6	0.03	5	0.02	6	230.49

Table 4.4. Treewidth lower bounds with contraction obtained by the $\delta_2 C$ and $\gamma_R C$ heuristics

The results show that values for $\delta_2 C$ are typically equal or only marginal better than the value for δC. The same is true for $\gamma_R C$ with respect to $\delta_2 C$. The best results are obtained by the most time-consuming algorithms: $\delta_2 C$(all-v) and $\gamma_R C$(all-v). By construction of the heuristic for $\gamma_R C$(all-v), it is clear that it is at least as good as the heuristic for $\delta_2 C$(all-v) strategy. Sometimes, it is even a little bit better. As in the case of the $\delta_2 D$ algorithm, the computation times of the $\delta_2 C$(all-v) and $\gamma_R C$(all-v) heuristics could probably be improved by more efficient implementations. The other strategies for $\delta_2 C$ and $\gamma_R C$ are comparable in value and running times. No clear trend between them can be identified. In a few cases, we can observe that the gap between the heuristically obtained values for δC and $\delta_2 C$ is more than one. This does not contradict Lemma 19, because the given values are not the exact values of δC and $\delta_2 C$, but lower bounds on them. Different strategies for heuristics can result in different values with larger gaps between them. With the same argument, we can explain that in a few cases lower bounds of one parameter that in theory is larger than another parameter, can be smaller than lower bounds of the other parameter.

4.4.5 Results for $MCSLB$ and $MCSLBC$

The results of the experiments with the $MCSLB$ and $MCSLBC$ heuristics in Table 4.5 show again the high potential of combining edge-contractions with treewidth lower bounds.

For the $MCSLBC$ heuristics, again the least-c strategy for selecting a neighbour seems to be better than the min-d strategy. We observe that min-deg and last-mcs for selecting the contraction vertex, combined with least-c outperform the other strategies. The differences between δC(least-c) and $MCSLBC$(least-c) are usually small, but in a few cases, the $MCSLBC$(least-c) gives a significant larger bound. The running times of these heuristics are often much larger compared to the δC heuristics. However, we can often observe that there is no large difference between the running times of the $MCSLBC$ heuristics.

instance	$MCSLB$		$MCSLBC$ LBs											
			min-deg				last-mcs				max-mcs			
			min-d		least-c		min-d		least-c		min-d		least-c	
	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU
link	5	3.26	8	51.51	10	52.19	8	50.52	11	51.58	8	17.62	6	18.25
munin1	4	0.17	8	0.88	10	0.88	9	0.84	10	0.86	9	0.82	7	0.86
munin3	4	6.42	6	30.49	7	30.57	7	30.79	7	30.91	6	29.20	7	29.45
pignet2	5	61.60	28	443.59	39	461.07	30	450.02	39	455.30	16	419.95	18	449.78
celar06	11	0.06	11	0.35	11	0.35	11	0.29	11	0.30	11	0.26	11	0.27
celar07pp	12	0.17	14	1.33	15	1.36	13	1.06	15	1.09	13	0.94	15	0.93
graph04	8	0.26	12	1.51	20	1.78	13	1.49	20	1.78	14	1.50	16	1.73
queen15-15	42	0.91	52	13.15	59	13.87	52	12.89	62	13.91	52	13.31	58	14.15
school1	85	5.40	97	131.78	122	138.15	110	134.59	118	137.64	104	106.86	118	112.78
school1-nsh	72	3.45	81	83.10	108	88.16	92	84.25	101	89.52	88	66.54	105	72.01
zeroin.i.1	50	0.39	50	4.64	50	4.79	50	4.59	50	4.89	50	4.28	50	4.62
rl5934-pp	4	5.38	5	26.69	6	26.91	5	25.11	6	26.66	5	26.02	6	26.43

Table 4.5. Results of MCSLB+ for selected instances

4.4.6 Results for the LBN and LBP Heuristics

Table 4.6 presents the results of some heuristics with graph improvement. We consider the heuristics LBN(δD), LBN(δC(least-c)), LBN+(δ) and LBN+(δD). We also look at the variants of these four heuristics where we use the path-improved graph instead of the neighbour-improved graph. For more details on these heuristics, see Sections 3.3 and 4.3.6. All the heuristics considered here use as the initial value the lower bound obtained by the δC(least-c) heuristic (see Table 4.3).

instance	LBN		LBN		LBN+		LBN+		LBP		LBP		LBP+		LBP+	
	(δD)		(δC(least-c))		(δ)		(δD)		(δD)		(δC(least-c))		(δ)		(δD)	
	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU	LB	CPU
link	11	0.01	11	0.03	11	0.19	11	1.92	11	6.25	11	6.48	12	38.85	12	41.68
munin1	10	0.01	10	0.00	10	0.02	10	0.13	10	0.10	10	0.10	10	0.17	10	0.26
munin3	7	0.01	7	0.02	7	0.24	7	3.21	7	12.41	7	12.82	7	27.66	7	28.40
pignet2	38	0.04	38	0.15	41	8.53	41	144.29	38	52.89	40	106.62	48	701.78	48	1011.68
celar06	11	0.00	11	0.01	11	0.01	11	0.05	11	0.09	11	0.09	11	0.12	11	0.15
celar07pp	15	0.00	15	0.01	15	0.03	15	0.15	15	0.82	15	0.85	16	1.95	16	2.43
graph04	20	0.01	20	0.01	21	0.11	21	0.45	20	0.01	20	0.02	24	2.89	24	2.69
queen15-15	58	0.01	58	0.12	60	3.52	60	5.08	58	0.01	58	0.12	73	4125.61	73	1571.34
school1	122	0.24	122	0.71	132	78.20	132	107.57	122	1627.78	128	2419.98	149	135748.37	149	92871.70
school1-nsh	106	0.08	106	0.41	116	52.06	116	74.07	106	741.33	111	1024.36	132	74905.93	132	58722.77
zeroin.i.1	50	0.02	50	0.04	50	0.34	50	0.57	50	23.08	50	24.95	50	23.38	50	23.99
rl5934-pp	5	0.00	5	0.03	5	0.20	5	2.76	6	5.18	6	5.41	9	12944.76	9	21.24

Table 4.6. Treewidth lower bounds obtained with the LBN/LBP and LBN+/LBP+ heuristics

As expected, the heuristics based on graph improvements can indeed improve treewidth lower bounds. However, as we observe, we do not obtain improvements with the LBN(δD) and the LBN(δC(least-c)) heuristics compared to the δC(least-c) heuristic for the selected graphs. Hence, neighbour-improvement is not a strong enough concept to improve on the lower bounds given by the δC(least-c) heuristic. The situation is similar for the LBP(δD) heuristic. The running times are often small, which can be explained by the fact that only one round of improvement happened.

The LBP(δC(least-c)) heuristic gives now and then small improvements. Whether this heuristic is preferable over an $MCSLBC$ heuristic depends on the input graph. The lower bound values usually do not differ much. Often the LBP(δC(least-c)) heuristic is faster, except for rather dense graphs.

The LBP+(δ) and LBP+(δD) heuristic can give considerable improvements to the lower bounds and as stated by Lemma 65, they are equal. But they appear to use very much time on some instances. A few cases could not be run to completion due to the large amount of time used; others give a result only after several many hours of computation time. Hence, if the lower bound has to be computed frequently, e.g. within a branch-and-bound algorithm, it is advisable to first compute a lower bound that can be computed quickly (e.g. δC(least-c)), and only in tight cases use a slower but hopefully better lower bound. While often no clear trend among the running times of the LBP+(δ) and LBP+(δD) heuristics can be observed, the situation for the TSP-instance rl5934-pp is remarkable. Here the LBP+(δ) heuristic finishes after nearly 13000 seconds, and the LBP+(δD) heuristic only needs a bit more than 21 seconds. A possible explanation for this is that δD delivers a better lower bound much earlier than δ, which results in an earlier increase of the lower bound computed by the LBP+ heuristic. Especially the path-improvements are very time-consuming. An earlier increase of the lower bound therefore means that much less improvements and contraction iterations are performed during the run of the LBP+(δD) heuristic.

4.5 Concluding Remarks

In our experiments, we could observe impressive improvements when comparing the simple parameters with their degeneracy counterparts. An even bigger improvement was achieved when edge-contractions (i.e. taking of minors) were used. Therefore, the experiments show that contracting edges is a very good approach for obtaining lower bounds for treewidth, as it considerably improves known lower bounds.

The δC heuristics appear to be attractive, due to the fact that the running time of these heuristics is almost always negligible, and the bound is reasonably good. The added value of $\delta_2 C$ and $\gamma_R C$ in comparison to δC is from a practical perspective marginal. The $MCSLBC$ heuristics have much larger running time, and often give only a small improvement on the δC based lower bound. The LBN, LBP, LBN+ and LBP+ heuristics often use more time than the δC heuristics, but less than the $MCSLBC$ heuristics (except for LBP+(δ)), and can give further lower bound improvements. The LBP+(δ) heuristic usually is slowest but gives often the best results.

Furthermore, we see that the strategy (least-c) for selecting a neighbour u of v with the least number of common neighbours of u and v often performs best and appears to be the clear choice for such a strategy.

Notice that although the gap between lower and upper bound could be significantly closed by contracting edges within the algorithms, the absolute gap is still large for many graphs (pignet2, graph...). However, there are also a lot of graphs with small gaps between lower and upper bound. For some graphs, it is even the case that the lower bound equals the upper bound, and hence, the treewidth is known now (see e.g. [18]).

While it is known that the treewidth has polynomial-time approximation algorithms with logarithmic performance ratios [13], the existence of polynomial-time approximation algorithms for treewidth with constant bounded ratios remains a long-standing open problem. Thus, obtaining good lower bounds for treewidth is both from a theoretical and from a practical viewpoint a highly interesting topic for further research.

As mentioned earlier, using γ_R instead of δ_2 in the degeneracy and contraction degeneracy heuristics, gives only small improvements in some cases. Therefore, the ratio of 2 between those parameters as stated in Lemma 21 is far from the ratios observed in our experiments.

It remains an interesting topic to research other treewidth lower bounds that can be combined with edge-contraction or minor-taking, in the hope to obtain large improvements. Furthermore, good lower bounds for graphs with bounded genus are desirable, because lower bounds based on δ, δ_2 or γ_R do not perform very well on such graphs (see Section 3.1.6). Treewidth lower bounds for planar graphs (i.e. graphs with genus zero) can be obtained e.g. by computing the branchwidth of the graph (see [55, 88]).

5 Contraction Degeneracy on Cographs

Contracting edges was shown to be of great use for obtaining new lower bound heuristics for treewidth (see Chapter 4). The new parameter *contraction degeneracy* arose from this research. The corresponding decision problem is NP-complete (see Section 3.1.3). See Section 3.1 for more details on this parameter. Because of its elementary formulation, research on this parameter is an fascinating topic in its own right and not only as a treewidth lower bound. Therefore, it is interesting to look for special graph classes where the contraction degeneracy can be computed in polynomial time. So far, little is known about this. In Section 3.1.5, we have seen that computing the contraction degeneracy of chordal graphs is trivial.

In this chapter, we consider the contraction degeneracy problem for the class of cographs. Cographs are graphs with a special structure. They are perfect and form a proper subset of the permutation graphs. Cographs can be defined recursively using two operations (disjoint union and complement, see Section 5.1) on singletons, and hence, every cograph can be represented by a tree, called its cotree. In Section 5.2, we present a polynomial time algorithm for computing the contraction degeneracy of a cograph, using a dynamic-programming approach on the cotree of a cograph. It should be noted that the contraction degeneracy of a cograph is not needed as a treewidth lower bound as the treewidth of a cograph is polynomial time computable [19]. This chapter is based on a cooperation with Hans L. Bodlaender [20].

5.1 Cographs

There are several independently discovered, but equivalent, definitions of *cographs*. As mentioned before, cographs (also called *complement reducible* graphs) can be defined recursively as follows: a single vertex is a cograph; given two cographs G_1 and G_2, the disjoint union $G = G_1 \cup G_2$ is a cograph; and if G is cograph, then the complement $\bar{G}$ of G is also a cograph. Another characterisation of cographs is that cographs are exactly those graphs that do not contain a P_4 (a path on four vertices) as an induced subgraph [69].

In the following, we will give an alternative, equivalent definition of cographs which requires the two operations 'disjoint union' and 'product'.

Definition 66. *Let two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be given, with $V_1 \cap V_2 = \emptyset$.*

- *The* disjoint union *of G_1 and G_2 is*
  $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$.
- *The* product *of G_1 and G_2 is*
  $G_1 \times G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{\{v_1, v_2\} : v_1 \in V_1 \wedge v_2 \in V_2\})$.

Note that the operations $\cup$ and $\times$ are commutative and associative. Hence, the result of a sequence of equal operations is well-defined.

Definition 67. *A graph G is a cograph if, and only if one of the following holds:*

- $|V| = 1$
- $G = G_1 \cup G_2 \cup ... \cup G_p$ *for cographs* $G_1, ..., G_p$.
- $G = G_1 \times G_2 \times ... \times G_p$ *for cographs* $G_1, ..., G_p$.

A consequence from this recursive definition is that a cograph can be represented as an expression using the operations $\cup$ and $\times$ on singletons. Such an expression can be represented by a tree from which we can derive a tree-representation, the *cotree* of the cograph [69, 37]. Very often, a dynamic-programming approach can be applied to such a cotree, enabling efficient algorithms on cographs for problems that are NP-hard on general graphs.

The cotree T_G (or simply T) of G is a labelled tree. Leaves of the cotree are in a one-to-one correspondence with the vertices of the cograph. Internal nodes of the cotree are labelled with either '0' or '1'. To each node of the cotree, we can associate a cograph in the following way: Each leaf of the cotree represents a graph with a single vertex, hence a cograph. A 0-node represents a cograph that is the disjoint union of the cographs corresponding to the children of the 0-node, and a 1-node represents a cograph that is the product of the cographs corresponding to the children of that 1-node. Here, we see the advantage of using the operations 'disjoint union' and 'product' for the definition of a cograph: namely the strong correlation between the recursive definition of the graph and the structure of the cotree. Note that there is an edge between two vertices v and w if and only if the lowest common ancestor of v and w in the cotree is a 1-node. There are linear time algorithms for recognising cographs and building the cotree [38, 27].

In Definition 67, we can restrict p to be 2. Consequently, the cotree can be taken as a binary tree, which is very helpful for formulating dynamic-programming algorithms. The size of such a binary cotree T_G of cograph G is linear in the size of G, because T_G has exactly $n = |V(G)|$ leaves, and after at most $O(n)$ nonempty disjoint union or product operations, we obtain the graph G. Therefore, we have at most $O(n)$ internal nodes in T_G. In the following, we always assume that the cotree is a binary cotree, i.e. 'cotree' refers to 'binary cotree'. Clearly, the contraction degeneracy of a disconnected graph is the maximum contraction degeneracy of its connected components. Thus, we may assume w.l.o.g. that the given cograph is connected, and hence, the root of the cotree is a 1-node.

5.2 Computing Contraction Degeneracy

In this section, we present a dynamic-programming method for computing the contraction degeneracy of a cograph G. We assume that G is given with a binary cotree T (otherwise, a cotree can be build in linear time [38, 27], from which a binary cotree can be easily constructed). The special structure of a binary cotree makes it easier to present a dynamic-programming algorithm. As already described earlier, cotrees have two kind of internal nodes: 1-nodes and 0-nodes. Every node i represents a subgraph G_i of the input graph G. Each internal node i of a binary cotree has exactly two children j_1 and j_2. When considering a 0-node i of a cotree, the graph G_i is simply the disjoint union of the two graphs G_{j_1} and G_{j_2}. Consequently, all edges that can be contracted in G_i can either be contracted in G_{j_1} or in G_{j_2}. However, if i is an 1-node, new edges are created between vertices of G_{j_1} and vertices of G_{j_2}. Hence, not all edges that can be contracted in G_i can be contracted in G_{j_1} or in G_{j_2}. Consider some fixed internal node i of T with children j_1 and j_2. We use the following terminology:

- G_i is the graph corresponding to node i, i.e. $G_i = G_{j_1} \cup G_{j_2}$ if i is a 0-node, and $G_i = G_{j_1} \times G_{j_2}$ if i is a 1-node.
- $V_i = V(G_i)$, $V_{j_1} = V(G_{j_1})$, $V_{j_2} = V(G_{j_2})$.
- $E_i = E(G_i)$, $E_{j_1} = E(G_{j_1})$, $E_{j_2} = E(G_{j_2})$.
- $n_i = |V_i|$.
- $e \in E_i$ is called a *cross edge* if i is a 1-node and $e \notin E_{j_1} \cup E_{j_2}$.
- $G \setminus V' = G[V \setminus V']$ for a graph $G = (V, E)$ and $V' \subseteq V$.

The following lemma is easy to see, but it is an important part of the correctness proof of our approach.

Lemma 68. *Contracting a cross edge of G_i creates a universal vertex in G_i.*

Proof. Given a cross edge $e = \{u, v\}$, $u \in V_{j_1}$ and $v \in V_{j_2}$, we know by definition of a 1-node, that u is adjacent to all $w \in V_{j_2}$ in G_i. Also, v is adjacent to all $w \in V_{j_1}$ in G_i. When contracting e, we get a vertex, adjacent to all $w \in V_i \setminus e$ in G_i. □

In our dynamic-programming approach, we associate to each node i, i.e. to each subgraph G_i, a function

$$F_i : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

For nonnegative integers x and y with $x + y \leq n_i$, $x \leq n_i - 1$, the function F_i is defined in the following way:

$$\begin{aligned} F_i(x, y) = \max\{\delta(H) \mid\, & H \text{ can be obtained from } G_i \text{ by} \\ & \text{contracting the edges of a contraction-set of size } x \\ & \text{and then deleting } y \text{ vertices and their incident edges}\} \end{aligned}$$

A table T_i with $O(n_i^2)$ entries can be used to store the values of F_i. The table contains a tuple for every possible x-y-combination. Each tuple consists of three integers (x, y, z), with $z = F_i(x, y)$.

The motivation behind storing the maximum minimum degree after x edge contractions and y vertex deletions is as follows. When considering a graph G_i, we can contract edges in $E(G_i)$ to increase the minimum degree. We also must consider the possibility that in order to obtain the maximum minimum degree of G, i.e. to solve the contraction degeneracy problem on G, it might be necessary to contract some edges not in $E(G_i)$ but in $E(G_h)$ where h is a 1-node in the cotree that is on the path from i to the root r of the cotree. Deleting a vertex can decrease the minimum degree. However, we do not delete vertices from the graph G_i, we only 'deactivate' them, such that they have no influence on the degrees at this moment. In a later stage, deactivated vertices might be used for contracting cross edges, which results in universal vertices (see Lemma 68). The consequences of the introduction of a universal vertex are easily computable, since the degree of each vertex is increased by one. This is the reason why we first 'deactivate' some vertices and later use them again (not explicitely). In the next lemma, we see that we indeed can compute the contraction degeneracy of a cograph by using function F_i.

Lemma 69. *Given the function F_i defined above for node i of a binary cotree, we can solve the contraction degeneracy problem for G_i in $O(n_i)$ time.*

Proof. To solve the problem on G_i means that we can only use contractions of edges in G_i to increase the minimum degree, and we must not delete any vertices. For every number x of contracted edges in G_i, and every number y of deleted vertices, we have the maximum minimum degree. For the contraction degeneracy problem, we only need these values with $y = 0$. Clearly, we have:

$$\delta C(G_i) = \max_{x=0,\ldots,n_i-1} F_i(x,0)$$

□

Hence, if we have the function F_r for the root r of the cotree, i.e. $G_r = G$, we can compute the contraction degeneracy of G in $O(n)$ additional time. Now we have seen that the functions F_i are sufficient to solve the problem, we look at how to compute these for each node i, using the functions for the children of i. As giving such a function for a leaf node i is trivial (there are just two values $F_i(0,0) = 0$ and $F_i(0,1) = \infty$), we describe the methods to compute the values of these functions for 0-nodes and for 1-nodes recursively.

5.2.1 A Recurrence Relations for 0-nodes

In this section, we give the recurrence relation for a 0-node i with children j_1 and j_2, i.e. we present and prove the correctness of a recursive formula to compute F_i using the already computed values of functions F_{j_1} and F_{j_2}.

Lemma 70. *Let i be a 0-node with children j_1 and j_2, x and y nonnegative integers with $x + y \leq n_i$, $x \leq n_i - 1$. Then we have:*

$$\begin{aligned} F_i(x,y) = \max \{ & \min(F_{j_1}(x_1,y_1), F_{j_2}(x_2,y_2)) \mid \\ & x_1,x_2,y_1,y_2 \in \mathbb{N} \wedge x_1 + x_2 = x \wedge y_1 + y_2 = y \wedge \\ & x_1 \leq n_{j_1} - 1 \wedge x_2 \leq n_{j_2} - 1 \} \end{aligned}$$

Proof. Let x and y be fixed nonnegative integers with $x + y \leq n_i$, $x \leq n_i - 1$. First, we will prove that $F_i(x,y)$ is at least the stated expression.

Claim. Let x_1, y_1, x_2, and y_2 be fixed integers with $x_1 + x_2 = x$, $y_1 + y_2 = y$, $x_1 \leq n_{j_1} - 1$ and $x_2 \leq n_{j_2} - 1$. Then $F_i(x,y) \geq \min(F_{j_1}(x_1,y_1), F_{j_2}(x_2,y_2))$.

Proof. Let E_1 be the contraction-set with $|E_1| = x_1$, and let V_1 be the vertex set with $|V_1| = y_1$, such that $F_{j_1}(x_1,y_1) = \delta((G_{j_1}/E_1) \setminus V_1)$. E_2 and V_2 are defined similarly.

Now we can contract in G_i the contraction-set $E_1 \cup E_2$, and then delete all vertices in $V_1 \cup V_2$. Since $F_i(x,y)$ is defined to be a maximum value, and $(G_i/(E_1 \cup E_2)) \setminus (V_1 \cup V_2)$ is a possible minor (see the definition of F_i), we have: $F_i(x,y) \geq \delta((G_i/(E_1 \cup E_2)) \setminus (V_1 \cup V_2)) = \min(F_{j_1}(x_1,y_1), F_{j_2}(x_2,y_2))$. ◇

As the claim holds for all nonnegative integers x_1, y_1, x_2 and y_2 with $x_1 + x_2 = x$, $y_1 + y_2 = y$, $x_1 \leq n_{j_1} - 1$ and $x_2 \leq n_{j_2} - 1$, we have:

$$\begin{aligned} F_i(x,y) \geq \max \{ & \min(F_{j_1}(x_1,y_1), F_{j_2}(x_2,y_2)) \mid \\ & x_1,x_2,y_1,y_2 \in \mathbb{N} \wedge x_1 + x_2 = x \wedge y_1 + y_2 = y \\ & x_1 \leq n_{j_1} - 1 \wedge x_2 \leq n_{j_2} - 1 \} \end{aligned}$$

Now we show that $F_i(x,y)$ is at most the stated expression. Consider a contraction-set $E' \subseteq E$ with $|E'| = x$, and a vertex set $V' \subseteq V$ with $|V'| = y$, such that

$$F_i(x,y) = \delta((G_i/E') \setminus V')$$

Note that i is a 0-node, and hence, each edge in E_i belongs to E_{j_1} or E_{j_2}. E' and V' can be partitioned in the following way:

$$\begin{array}{llll} E_1 := E_{j_1} \cap E' & E_2 := E_{j_2} \cap E' & V_1 := V_{j_1} \cap V' & V_2 := V_{j_2} \cap V' \\ x_1 := |E_1| & x_2 := |E_2| & y_1 := |V_1| & y_2 := |V_2| \end{array}$$

Claim. It holds that: $x_1 + x_2 = x$, $y_1 + y_2 = y$, $x_1 \leq n_{j_1} - 1$ and $x_2 \leq n_{j_2} - 1$.

Proof. From the partition of E' and V', it directly follows that $x_1 + x_2 = x$ and $y_1 + y_2 = y$. Each contraction-set is a forest, and hence, each contraction-set has at most $n - 1$ edges, where n is the number of vertices in the considered graph. E_1 is a contraction-set, since $E_1 \subseteq E'$. Therefore, E_1 is a contraction-set for G_{j_1}. Thus, $x_1 \leq n_{j_1} - 1$ and $x_2 \leq n_{j_2} - 1$. ◇

We have $G_i = G_{j_1} \cup G_{j_2}$ and by definition of E_1, E_2, V_1 and V_2, the following is easy to see. $(G_i/E') \setminus V' = (G_{j_1}/E_1) \setminus V_1 \cup (G_{j_2}/E_2) \setminus V_2$, and therefore, we have:

$$F_i(x,y) = \delta((G_i/E') \setminus V') = \min(\delta((G_{j_1}/E_1) \setminus V_1), \delta((G_{j_2}/E_2) \setminus V_2))$$

So, $F_i(x,y) \leq \delta((G_{j_1}/E_1) \setminus V_1) \leq F_{j_1}(x_1,y_1)$. Similarly, $F_i(x,y) \leq F_{j_2}(x_2,y_2)$, and hence, $F_i(x,y) \leq \min(F_{j_1}(x_1,y_1), F_{j_2}(x_2,y_2))$. □

5.2.2 A Recurrence Relations for 1-nodes

This section is devoted to the recurrence relation for F_i for a 1-node i with children j_1 and j_2. However, before we present and prove the correctness of the recursive formula to compute F_i (using the already computed values of functions F_{j_1} and F_{j_2}), we introduce a modified version F'_i of F_i. The function F'_i is defined especially for 1-nodes. The advantage is that it is easier and faster to compute. Later, we will show that we can compute F_i using function F'_i. Let i be a 1-node with associated graph $G_i = (V_i, E_i)$, and let $E^c \subseteq E_i$ be the set of cross edges of G_i. For nonnegative integers x and y with $x + y \leq n_i$ and $x \leq n_i - 1$, the function $F'_i : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ is defined in the following way:

$$\begin{aligned} F'_i(x,y) = \max\{\delta(H) \mid\ & H \text{ can be obtained from } G_i \text{ by} \\ & \text{contracting the edges of a contraction-set } E' \text{ of size } x \\ & \text{and then deleting } y \text{ vertices and their incident edges,} \\ & \text{where for all } e,f \in E^c \cap E' : e \cap f = \emptyset \ \} \end{aligned}$$

The last requirement in the definition of F'_i says that all cross edges in E' are pairwise disjoint, i.e. any two edges in E' have no endpoint in common. Hence, every cross edge in E' creates a unique universal vertex. We first give a number of lemmas which will be used later. For Lemmas 71, 72 and 73, suppose the following is given. The graph $G_i = G_{j_1} \times G_{j_2}$ is the product of G_{j_1} and G_{j_2}. Furthermore, let us be given contraction-sets $E' \subseteq E_i$, $|E'| = x$, $E_1 \subseteq E_{j_1}$ and $E_2 \subseteq E_{j_2}$, vertex sets $V' \subseteq V_i$, $|V'| = y$, $V_1 \subseteq V_{j_1}$, $V_2 \subseteq V_{j_2}$, with $E_1 = E' \cap E_{j_1}$, $|E_1| = x_1$, $E_2 = E' \cap E_{j_2}$, $|E_2| = x_2$, $E^* = E' \setminus (E_1 \cup E_2)$, $|E^*| = x^*$, and E^* only contains pairwise disjoint edges, i.e. $\forall e,f \in E^* : e \cap f = \emptyset$. Furthermore, let $V^* = \bigcup_{e \in E^*} e$, $V_1 = V_{j_1} \cap (V' \cup V^*)$, $|V_1| = y_1$, $V_2 = V_{j_2} \cap (V' \cup V^*)$, $|V_2| = y_2$ and $y_1 \geq x^*$, $y_2 \geq x^*$. Note that E^* is the set of cross edges that are contracted, and V^* is the set of endpoints of edges in E^*.

Lemma 71. *Each of the following holds:*

1. $|V(G_i/E' \setminus V')| = n_i - x - y$.
2. $x + y = n_i \Longrightarrow F_i(x,y) = \infty$.
3. $x + y < n_i \Longrightarrow F_i(x,y) < n_i - x - y$.

Proof. (1.) Note that G_i has n_i vertices. Since E' is a contraction-set, every contraction of an edge in E' results in a decrease of the number of vertices by one. Hence, $|V(G_i/E')| = n_i - x$. Now, we have to delete y vertices in G_i/E'. Clearly, $|V(G_i/E' \setminus V')| = n_i - x - y$.

(2.) From *(1.)*, we can conclude that $G_i/E' \setminus V'$ is the null graph (the empty graph with 0 vertices). Hence, the minimum degree of a vertex over an empty vertex set is the minimum of an empty set of integers, i.e. it is ∞.

(3.) Since $G_i/E' \setminus V'$ has $n_i - x - y$ vertices, the maximum degree of a vertex in $G_i/E' \setminus V'$ is at most $n_i - x - y - 1$. As this holds for every E' and V' with $|E'| = x$, $|V'| = y$, we have $F_i(x, y) \leq n_i - x - y - 1$. □

Note that we explicitly excluded the null graph as a minor and as a subgraph of a graph. However, we will use the minimum degree of the null graph in our recurrence relations, i.e. we need statement *(2.)* of Lemma 71.

Lemma 72. *Let $v_1 \in V(G_{j_1}/E_1 \setminus V_1)$. Then we have:*

$$d_{G_i/E' \setminus V'}(v_1) = d_{G_{j_1}/E_1 \setminus V_1}(v_1) + n_{j_2} - x_2 - y_2 + x^*$$

Proof. Since E_1 and V_1 only change the internal structure and the number of vertices in G_{j_1}, we have: $d_{(G_{j_1} \times G_{j_2})/E_1 \setminus V_1}(v_1) = d_{G_{j_1}/E_1 \setminus V_1}(v_1) + n_{j_2}$. With the same argument that E_2 and V_2 only affect G_{j_2}, we have: $d_{(G_{j_1} \times G_{j_2})/(E_1 \cup E_2) \setminus (V_1 \cup V_2)}(v_1) = d_{G_{j_1}/E_1 \setminus V_1}(v_1) + n_{j_2} - x_2 - y_2$ (see *(1.)* in Lemma 71). However, instead of deleting all vertices in $V_1 \cup V_2$, we contract x^* disjoint edges with one endpoint in V_1 and the other endpoint in V_2, i.e. we contract all cross edges in E^*. It is easy to see that this results in x^* additional universal vertices (see Lemma 68). Hence, we have:

$$d_{(G_{j_1} \times G_{j_2})/(E_1 \cup E_2 \cup E^*) \setminus [(V_1 \cup V_2) \setminus V^*]}(v_1) = d_{G_{j_1}/E_1 \setminus V_1}(v_1) + n_{j_2} - x_2 - y_2 + x^*$$

□

Lemma 73. *Let $v_1, v_2 \in V(G_{j_1}/E_1 \setminus V_1)$. Then we have:*

$$d_{G_{j_1}/E_1 \setminus V_1}(v_1) \leq d_{G_{j_1}/E_1 \setminus V_1}(v_2) \iff d_{G_i/E' \setminus V'}(v_1) \leq d_{G_i/E' \setminus V'}(v_2)$$

Proof. This follows directly from Lemma 72. □

Lemma 74. *If $x + y = n_i$, then $F'_i(x, y) = \infty$.*

Proof. This is similar to Lemma 71*(2.)*. □

Now in Lemmas 75 and 76, we give the recurrence relation for 1-nodes.

Lemma 75. *Let i be a 1-node with children j_1 and j_2. Let x and y be nonnegative integers with $x + y \leq n_i$, $x \leq n_i - 1$. Then, we have:*

$$\begin{aligned} F'_i(x, y) = \max\{ \min(& F_{j_1}(x_1, y_1) + n_{j_2} - x_2 - y_2 + x^*, \\ & F_{j_2}(x_2, y_2) + n_{j_1} - x_1 - y_1 + x^*, \\ & n_i - x - y - 1) \mid \\ & x_1, x_2, y_1, y_2, x^* \in \mathbb{N} \wedge \\ & x_1 + x_2 + x^* = x \wedge y_1 + y_2 = y + 2 \cdot x^* \wedge \\ & y_1 \geq x^* \wedge y_2 \geq x^* \wedge \\ & x_1 + y_1 \leq n_{j_1} \wedge x_2 + y_2 \leq n_{j_2} \} \end{aligned}$$

Proof. Let x and y be fixed nonnegative integers with $x + y \leq n_i$, $x \leq n_i - 1$. We will first prove that $F'_i(x, y)$ is at least the stated expression. Therefore, let x_1, x_2, y_1, y_2 and x^* be fixed nonnegative integers fulfilling the requirements stated in the lemma. We will first prove:

$$\begin{aligned} F'_i(x,y) \geq \min(\ & F_{j_1}(x_1,y_1) + n_{j_2} - x_2 - y_2 + x^*, \\ & F_{j_2}(x_2,y_2) + n_{j_1} - x_1 - y_1 + x^*, \\ & n_i - x - y - 1\ ) \end{aligned}$$

Let E_1 and E_2 be contraction-sets with $|E_1| = x_1$ and $|E_2| = x_2$ and V_1 and V_2 be sets of vertices with $|V_1| = y_1$ and $|V_2| = y_2$, such that $F_{j_1}(x_1, y_1) = \delta(G_{j_1}/E_1 \setminus V_1)$ and $F_{j_2}(x_2, y_2) = \delta(G_{j_2}/E_2 \setminus V_2)$. Let E^* be a set of x^* pairwise disjoint cross edges, each of them having one endpoint in V_1 and the other in V_2, and let $V^* = \bigcup_{e \in E^*} e$. As $|V_1| = y_1 \geq x^*$ and $|V_2| = y_2 \geq x^*$, and each vertex in V_1 is adjacent to each vertex in V_2, we can choose such a set E^*. We define: $E' := E_1 \cup E_2 \cup E^*$ and $V' := (V_1 \cup V_2) \setminus V^*$.

Claim. $|E'| = x$ and $|V'| = y$.

Proof. This follows from the definition of the corresponding sets. Note that E_1, E_2 and E^* are pairwise disjoint. Furthermore, observe that V_1 and V_2 are disjoint and $V^* \subseteq (V_1 \cup V_2)$. Hence, $|V'| = |V_1| + |V_2| - |V^*| = y_1 + y_2 - 2 \cdot x^*$. $\diamond$

Note that $V(G_i/E' \setminus V')$ can be partitioned into three disjoint sets: $W_1 := V(G_{j_1}/E_1 \setminus V_1)$, $W_2 := V(G_{j_2}/E_2 \setminus V_2)$, and the set of vertices that result from contracting a cross edge, $W^* := V(G_i/E' \setminus V') \setminus (W_1 \cup W_2)$.

If W_1 is empty, then $x_1 + y_1 = n_{j_1}$ and $F_{j_1}(x_1, y_1) = \infty$. Otherwise, let v_1 be a vertex in $G_{j_1}/E_1 \setminus V_1$ of minimum degree, i.e. $d_{G_{j_1}/E_1 \setminus V_1}(v_1) = F_{j_1}(x_1, y_1)$. From Lemma 73, we know that all other vertices in $G_{j_1}/E_1 \setminus V_1$ have degree in $G_i/E' \setminus V'$ as least as large as the degree of v_1 in $G_i/E' \setminus V'$. So, by Lemma 72 all vertices in W_1 have degree at least $d_{G_{j_1}/E_1 \setminus V_1}(v_1) + n_{j_2} - x_2 - y_2 + x^* \geq F_{j_1}(x_1, y_1) + n_{j_2} - x_2 - y_2 + x^*$. Similarly, either W_2 is empty, in which case $F_{j_2}(x_2, y_2) = \infty$, or all vertices in W_2 have degree at least $F_{j_2}(x_2, y_2) + n_{j_1} - x_1 - y_1 + x^*$.

Vertices that are the result of a contraction of a cross edge are universal in $G_i/E' \setminus V'$, and hence have degree $n_i - x - y - 1$.

We can conclude that

$$\begin{aligned} F'_i(x,y) &\geq \delta(G_i/E' \setminus V') \\ &\geq \min(\ F_{j_1}(x_1,y_1) + n_{j_2} - x_2 - y_2 + x^*, \\ &\qquad\quad F_{j_2}(x_2,y_2) + n_{j_1} - x_1 - y_1 + x^*, \\ &\qquad\quad n_i - x - y - 1) \end{aligned}$$

Since this holds for all x_1, x_2, y_1, y_2 and x^*, fulfilling the conditions as in the lemma, we can conclude:

$$\begin{aligned} F'_i(x,y) \geq \max\{ \min(\ & F_{j_1}(x_1,y_1) + n_{j_2} - x_2 - y_2 + x^*, \\ & F_{j_2}(x_2,y_2) + n_{j_1} - x_1 - y_1 + x^*, \\ & n_i - x - y - 1) \mid \\ & x_1, x_2, y_1, y_2, x^* \in \mathbb{N} \wedge \\ & x_1 + x_2 + x^* = x \wedge y_1 + y_2 = y + 2 \cdot x^* \wedge \\ & y_1 \geq x^* \wedge y_2 \geq x^* \wedge \\ & x_1 + y_1 \leq n_{j_1} \wedge x_2 + y_2 \leq n_{j_2} \wedge\ \} \end{aligned}$$

We now show the equality, i.e. we show that $F_i'(x,y)$ is at most the given expression. Let E' be a contraction-set of size x containing pairwise disjoint cross edges, and V' be a vertex set of size y such that $F_i'(x,y) = \delta(G_i/E' \setminus V')$. E' and V' can be partitioned in the following way:

$$\begin{array}{lll}
E_1 := E_{j_1} \cap E' & E_2 := E_{j_2} \cap E' & E^* := E' \setminus (E_1 \cup E_2) \\
V_1 := V_{j_1} \cap (V' \cup V^*) & V_2 := V_{j_2} \cap (V' \cup V^*) & V^* := \bigcup_{e \in E^*} e \\
x_1 := |E_1| & x_2 := |E_2| & x^* := |E^*| \\
y_1 := |V_1| & y_2 := |V_2| &
\end{array}$$

Claim. $x = x_1 + x_2 + x^*$, $y_1 + y_2 = y + 2 \cdot x^*$, $y_1 \geq x^*$, $y_2 \geq x^*$, $x_1 + y_1 \leq n_{j_1}$, $x_2 + y_2 \leq n_{j_2}$, $x_1 \leq n_{j_1}$, $x_2 \leq n_{j_2}$.

Proof. The first equality follows directly from the corresponding definitions.

Note that E^* is a contraction-set containing all cross edges of E' (they are pairwise disjoint). Hence, $|V^*| = 2 \cdot x^*$. Furthermore, observe that V_{j_1} and V_{j_2} are disjoint and $V_{j_1} \cup V_{j_2} = V_i \supseteq (V' \cup V^*)$. Therefore, $y_1 + y_2 = |V_1| + |V_2| = |V_{j_1} \cap (V' \cup V^*) \cup V_{j_2} \cap (V' \cup V^*)| = |V_i \cap (V' \cup V^*)| = |V' \cup V^*| = y + 2 \cdot x^*$.

The inequality $y_1 \geq x^*$ follows from $y_1 = |V_1| = |V_{j_1} \cap (V' \cup V^*)| \geq |V_{j_1} \cap V^*| = |E^*| = x^*$. The equality $|V_{j_1} \cap V^*| = |E^*|$ can be seen from the fact that E^* contains exactly x^* pairwise disjoint cross edges that have one endpoint in V_{j_1} and the other endpoint in V_{j_2}.

Since E_1 is the subset of E', restricted to G_{j_1}, E_1 is a contraction-set of G_{j_1}, and V_1 is a subset of vertices in G_{j_1}/E_1, since V' is a subset of vertices in G_i/E'. Hence, $x_1 + y_1 \leq n_{j_1}$ and $x_1 \leq n_{j_1}$.

In the same way, we can conclude $y_2 \geq x^*$, $x_2 + y_2 \leq n_{j_2}$ and $x_2 \leq n_{j_2}$. ◇

Similar as above, we write $W_1 = V(G_{j_1}/E_1 \setminus V_1)$, $W_2 = V(G_{j_2}/E_2 \setminus V_2)$ and $W^* = V(G_i/E' \setminus V') \setminus (W_1 \cup W_2)$ for the set of vertices resulting from contracting a cross edge. Note that W_1, W_2 and W^* partition $V(G_i/E' \setminus V')$. We consider four different cases.

Case 1 '$W_1 = \emptyset$ and $W_2 = \emptyset$': In this case, $G_i/E' \setminus V'$ is a clique with x^* vertices, as each vertex in this graph is the result of a contraction of a cross edge. So, $\delta(G_i/E' \setminus V') = x^* - 1 = n_i - x - y - 1$. As $W_1 = \emptyset$, we must have that $n_{j_1} - x - y - 1$, and hence $F_{j_1}(x_1, y_1) = \infty$. Similarly, $F_{j_2}(x_2, y_2) = \infty$. So,

$$\begin{aligned}
F_i'(x,y) &= \delta(G_i/E' \setminus V') \\
&= \min(\ F_{j_1}(x_1,y_1) + n_{j_2} - x_2 - y_2 + x^*, \\
&\qquad\quad F_{j_2}(x_2,y_2) + n_{j_1} - x_1 - y_1 + x^*, \\
&\qquad\quad n_i - x - y - 1\ )
\end{aligned}$$

Case 2 '$W_1 \neq \emptyset$ and $W_2 = \emptyset$': First note that $W_2 = \emptyset$ implies that $F_{j_2}(x_2, y_2) = \infty$, similar to the previous case. Take a vertex $v_1 \in W_1$ which has minimum degree in $G_i/E' \setminus V'$. Note that vertices in W^* are universal in $G_i/E' \setminus V'$, hence have a degree that is at least the degree of v_1 in this graph. So, we have (use Lemmas 72 and 73)

$$\begin{aligned}
F_i'(x,y) &= \delta(G_i/E' \setminus V') \\
&= d_{G_i/E' \setminus V'}(v_1) \\
&= d_{G_{j_1}/E_1 \setminus V_1}(v_1) + n_{j_2} - x_2 - y_2 + x^* \\
&\leq F_{j_1}(x_1,y_1) + n_{j_2} - x_2 - y_2 + x^* \\
&= \min(\ F_{j_1}(x_1,y_1) + n_{j_2} - x_2 - y_2 + x^*, \\
&\qquad\quad F_{j_2}(x_2,y_2) + n_{j_1} - x_1 - y_1 + x^*, \\
&\qquad\quad n_i - x - y - 1)
\end{aligned}$$

The last step follows by using that $F_{j_2}(x_2, y_2) = \infty$, and noting that $F_{j_1}(x_1, y_1)+n_{j_2}-x_2-y_2+x^* \leq n_{j_1} - x_1 - y_1 - 1 + n_{j_2} - x_2 - y_2 + x^* = n_i - x - y - 1$.

Case 3 '$W_1 = \emptyset$ and $W_2 \neq \emptyset$': Similar to the previous Case 2, with the roles of W_1 and W_2 exchanged.

Case 4 '$W_1 \neq \emptyset$ and $W_2 \neq \emptyset$': As in Case 2, $F_{j_1}(x_1, y_1) + n_{j_2} - x_2 - y_2 + x^* \leq n_i - x - y - 1$, and similar $F_{j_2}(x_2, y_2) + n_{j_1} - x_1 - y_1 + x^* \leq n_i - x - y - 1$. Take a vertex $v_1 \in W_1$ that has minimum degree in $G_i/E' \setminus V'$ among all vertices in W_1, and similarly take a vertex $v_2 \in W_2$ with minimum degree in $G_i/E' \setminus V'$. Again, vertices in W^* have a degree that is at least the degree of v_1 (or v_2) in $G_i/E' \setminus V'$. So, we have using Lemmas 72 and 73:

$$\begin{aligned}
F'_i(x, y) &= \delta(G_i/E' \setminus V') \\
&= \min(\, d_{G_i/E'\setminus V'}(v_1), \\
&\qquad\quad d_{G_i/E'\setminus V'}(v_2)) \\
&= \min(\, d_{G_{j_1}/E_1\setminus V_1}(v_1) + n_{j_2} - x_2 - y_2 + x^*, \\
&\qquad\quad d_{G_{j_2}/E_2\setminus V_2}(v_2) + n_{j_1} - x_1 - y_1 + x^*) \\
&\leq \min(\, F_{j_1}(x_1, y_1) + n_{j_2} - x_2 - y_2 + x^*, \\
&\qquad\quad F_{j_2}(x_2, y_2) + n_{j_1} - x_1 - y_1 + x^*) \\
&= \min(\, F_{j_1}(x_1, y_1) + n_{j_2} - x_2 - y_2 + x^*, \\
&\qquad\quad F_{j_2}(x_2, y_2) + n_{j_1} - x_1 - y_1 + x^*, \\
&\qquad\quad n_i - x - y - 1)
\end{aligned}$$

This ends the last case of the proof, and hence we can conclude Lemma 75. □

Now we have seen that our recurrence relation for F'_i is correct, we will show how we can compute F_i given F'_i.

Lemma 76.

$$F_i(x, y) = \max_{0 \leq z \leq x} F'_i(x - z, y + z)$$

Proof. We will first prove that $F_i(x, y) \geq \max_{0\leq z\leq x} F'_i(x - z, y + z)$.

Claim. $F_i(x, y) \geq F'_i(x - 1, y + 1)$.

Proof. Let E' be a contraction-set of size $x - 1$ containing only pairwise disjoint cross edges, and let V' be a vertex set of size $y + 1$ such that: $F'_i(x - 1, y + 1) = \delta(G_i/E' \setminus V')$. Let $v \in V'$ be a vertex that we deleted in $G_i/E' \setminus V'$, i.e. $v \in V'$. Instead of deleting it, we can contract it using a cross edge e incident to v. Since v is not contained in $G_i/E' \setminus V'$, contracting it via a cross edge (no matter whether e has nonempty intersection with another cross edge in E'), will not decrease any vertex degree. Note that unless v is the only vertex left, a cross edge incident to v always exists, since G_{j_1} and G_{j_2} are graphs with at least one vertex each. Hence, we have a contraction-set $E' \cup \{e\}$ of x edges and a vertex set $V' \setminus \{v\}$ of y vertices, such that:

$$F_i(x, y) \geq \delta(G_i/(E' \cup \{e\}) \setminus (V' \setminus \{v\})) \geq F'_i(x - 1, y + 1)$$

◇

A similar argument can be used when $2 \leq z \leq x$. In this case, we contract z vertices with cross edges incident to them instead of deleting them. Thus, for all z, $0 \leq z \leq x$, we have $F_i(x,y) \geq F'_i(x-z, y+z)$, and hence

$$F_i(x,y) \geq \max_{0 \leq z \leq x} F'_i(x-z, y+z) \tag{5.1}$$

We will now show that $F_i(x,y) \leq \max_{0 \leq z \leq x} F'_i(x-z,y+z)$. Let E' be a contraction-set of size x, and let V' be a vertex set of size y, such that $F_i(x,y) = \delta(G_i/E' \setminus V')$. Let $E^* \subseteq E'$ be the set of all cross edges in E', and let be $V^* := \bigcup_{e \in E^*} e$. Observe that (V^*, E^*) is a forest without isolated vertices. Let c be the number of connected components in (V^*, E^*). We modify E^* in the following way to obtain E^{**}. In each connected component of (V^*, E^*), we delete all but one edge, resulting in the forest (V^*, E^{**}). Let be $V^{**} := \bigcup_{e \in E^{**}} e$. Clearly, E^{**} only contains pairwise disjoint cross edges and $|E^{**}| = c$, $|V^{**}| = 2c$. We define:

$$E'' := (E' \setminus E^*) \cup E^{**} \quad \text{and} \quad V'' := V' \cup (V^* \setminus V^{**}) \quad \text{and} \quad z := |E'| - |E''|$$

Claim. $|E''| = |E'| - z = x - z$ and $|V''| = |V'| + z = y + z$.

Proof. $|E''| = |E'| - z$ follows directly from the definition of z. To see the other equality, note that the vertex sets are disjoint or contained in each other when applying basic operations. Furthermore, note that in a forest, the number of edges plus the number of connected components equals the number of vertices. We therefore have:

$$\begin{aligned}
|V''| &= |V' \cup (V^* \setminus V^{**})| = |V'| + |V^*| - |V^{**}| = y + |E^*| + c - 2c \\
&= y + |E^*| - c = y + |E^*| - |E^{**}| = y + |E'| - |E'| + |E^*| - |E^{**}| \\
&= y + |E'| - (|E'| - |E^*| + |E^{**}|) = y + |E'| - (|E' \setminus E^*| + |E^{**}|) \\
&= y + |E'| - |(E' \setminus E^*) \cup E^{**}| = y + |E'| - |E''| = y + z
\end{aligned}$$

◇

Claim. Let E' contain two nondisjoint cross edges. $e = \{u_1, u_2\}$ and $f = \{u_2, w\}$, such that there is no other edge in E' than e with non-empty intersection with f. Let v be a vertex in $G_i/(E' \setminus \{f\}) \setminus V'$. Then it holds that: $d_{G_i/E' \setminus V'}(v) = d_{G_i/(E' \setminus \{f\}) \setminus (V' \cup \{w\})}(v)$.

Proof. If w is not adjacent to v then the claim follows easily. So suppose $\{v, w\}$ is an edge in $G_i/(E' \setminus \{f\}) \setminus V'$. Since e was (w.l.o.g.) contracted before f, the vertex created by contracting e is a universal vertex a_e. Now, we can either delete vertex w, resulting in decreasing the degree of v by one, or we can contract edge f to the universal vertex a_e, which also results in ‘losing’ one edge incident to v, since $\{a_e, v\}$ is already present in $G_i/(E' \setminus \{f\}) \setminus V'$. Hence, the degree of v is the same in both cases. ◇

Applying the last claim iteratively, we can conclude: $\delta(G_i/E' \setminus V') = \delta(G_i/E'' \setminus V'')$. Hence, there is a z, such that $F_i(x,y) = F'_i(x-z, y+z)$. From this fact and Equation 5.1, the lemma follows. □

5.2.3 The Dynamic-Programming Method

Now, we are able to formulate our main result. So far, we only presented the recurrence relations necessary for dynamic programming. It is an easy task to transform these relations into an algorithm for computing the contraction degeneracy of a cograph..

Theorem 77. *There is an $O(n^6)$ time algorithm to compute the contraction degeneracy of a given cograph G with n vertices.*

Proof. Given a cograph G, we first build the cotree in linear time [38, 27], from which the binary cotree can be easily derived. We then compute in bottom-up order for each node i in the cotree the relevant values of F_i. For 1-nodes, we first compute F'_i. These computations are as dictated by Lemmas 70, 74, 75 and 76. After the root values are computed, we use Lemma 69 to compute the contraction degeneracy of G.

To estimate the running time, consider the formula for computing F'_i on 1-nodes, since the computation time needed for this formula dominates the others. We can implement this formula by five nested loops, for x_1, x_2, y_1, y_2 and x^*, ranging over the appropriate domains (at most $\{0, ..., n_i\}$). In the innermost loop body, we check the restrictions, compute x, y and the minimum as given in the formula, and we update $F'_i(x, y)$ if we have found a larger value. This loop takes $O(n^5)$ time, and since we have $O(n)$ internal nodes in the cotree, the theorem follows. □

It is also possible to obtain in $O(n^6)$ time the set of contractions that achieve the contraction degeneracy, using standard techniques for transforming a dynamic-programming-decision algorithm into one that also constructs solutions.

5.3 Concluding Remarks

The contraction degeneracy of a graph appears to be an interesting graph parameter, not only as a lower bound for treewidth. So far, little is known for it. Its strong relation to the well-understood minimum degree δ and degeneracy δD of a graph, and its elementary nature make it a worthwhile object of study. In this chapter, we have presented a dynamic-programming method for computing the contraction degeneracy of a cograph. The running time of our algorithm is polynomial but surprisingly high, since cographs are a very restricted class of graphs with a tree representation, that usually enables faster algorithms. This might be a consequence of the possible inherent hardness of computing δC of a graph. However, it might be possible to obtain a slightly more efficient time-bound by a more careful analysis of our algorithm. It therefore remains an interesting topic for further research to decrease the running time for cographs or to develop algorithms for other special graph classes.

6

Network Reliability: Model, Problems and Complexity

In a world with a growing need for communication and data exchange, computer and telecommunication networks play a decisive role. In most applications, it is of great importance that connections between the communicating parties remain intact. Unfortunately, hardware and software are not infallible. For this reason modern networks are designed such that two communication sites can have more than one possibility of information exchange. Furthermore, one can often observe that some sites in a network are of greater importance than others. For instance, connections between servers may be more important than between clients. It is important to compute the reliability of a network not only during the construction process but also for improving and determining the quality of existing networks.

In this chapter, we look at two classical network reliability problems (Section 6.1) and briefly describe the model they are based on. We define our graph-theoretical model in Section 6.2. Also in this section, we explain how edge failures can be simulated by vertex failures, and we give a list of relevant network reliability problems. The related complexity issues are the topic of Section 6.3, where we first give some definitions and background information and then prove the $\#P'$-membership and -hardness of our problems. This chapter is an adaptation from joint work with Hans L. Bodlaender (see [21]).

6.1 The Classical Network Reliability Problems

The two classical network reliability problems which are considered here, are based on the following model, which is used e.g. in [4]. Let an undirected, simple graph G be given with a number $p_e \in [0, 1]$ associated to each edge e of G. This number p_e is the reliability of edge e. In this classical model only links between sites can break down independently of each other, i.e. some edges between vertices might not be present in the surviving subgraph. The surviving subgraph is the subgraph consisting of all elements corresponding to objects that are not broken down. Therefore, p_e is the probability that the edge e is present in the surviving subgraph. We can now consider the following classical problems:

- $[2 \leftrightarrow t]_e$ (Two Terminal Network Reliability Problem): What is the probability that there is a path between two distinguished vertices s and t, given the reliabilities p_e for the edges?
- $[A \leftrightarrow t]_e$ (All Terminal Network Reliability Problem): What is the probability that all vertices remain connected, given the reliabilities p_e for the edges?

We will use the abbreviations with brackets as the name of the respective problems when we refer to network reliability problems (e.g. in formulas or figures). The subscript e indicates that only edges can fail in the underlying model.

6.2 Our Network Reliability Problems

Our model generalises the classical model in a way that allows more complicated questions about the connectivity of two distinguished sets of vertices. For instance, in the situation where we have a set of clients and a set of servers, we could ask the following questions: 'What is the probability that all clients are connected to at least one server?' or 'What is the expected number of components of the graph induced by the non-failed elements that contain a server?' Before we have a look at more examples of network reliability problems that we will prove to be $\#P'$-complete (see Section 6.3), we introduce the graph-theoretical model which the problems are based on. We will restrict ourselves to vertex-failures and assume that edges are perfectly reliable. However, we will see that this does not restrict the expressiveness of our model, because edge failures can be modelled by vertex failures.

6.2.1 Our Graph-theoretical Model

In contrast to the classical model, where only edges can fail, we have a model where only vertices can break down. For all our network reliability problems, we are given an undirected, simple graph $G = (V, E)$ in which to each vertex $v \in V$ a rational number $p(v) \in [0, 1]$ is associated:

$$p(v) = \frac{a_v}{b_v}, \;\; a_v, b_v \in \mathbb{N}, \;\; a_v \leq b_v, \;\; b_v \neq 0, \text{ and}$$
$$a_v, b_v \text{ have no common divisor greater than } 1$$

For each $v \in V$, the number $p(v)$ is the *reliability* of v. It models the probability that the network element represented by v is properly functioning. We are interested in the subgraph of G obtained as follows: each $v \in V$ is in the subgraph with probability $p(v)$, and each edge is in the subgraph if and only if both its endpoints are in the subgraph. Such a model is often called a *stochastic graph*. The subgraph resulting from this experiment is called the *surviving subgraph*. The vertices in the surviving subgraph are said to be *up*, while the others not in the surviving subgraph are said to be *down*. We assume that the occurrence of failures of elements are statistically independent.

Furthermore, we are given a set $S \subseteq V$ of *servers* and a set $L \subseteq V$ of *clients*, $n_S = |S|$ and $n_L = |L|$. S and L might have vertices in common. In addition, we also have other vertices that serve simply to build connections.

The problems we consider, ask for the probabilities that there are certain connections between vertices in S, and also that there are certain connections between vertices of S and vertices of L. These problems are listed in Section 6.2.3.

6.2.2 Edge Failures vs. Vertex Failures

In the classical network reliability problems only links between sites can fail; or translated into graph theory: only edges of the graph can be non-present. In our model, however, we consider network reliability problems where only vertices can go down (and hence their incident edges with them), and edges between two present vertices will always be present in the graph (i.e. existing edges are perfectly reliable).

This model with only vertex failures is not a restriction, since edge failures can be simulated by vertex failures in the following way. Subdivide each edge $e = \{u, w\}$ of the original graph with a new vertex v_e (see Figure 2.5 in Section 2.5). The reliability of v_e is set to be equal to the reliability p_e of the edge $e = \{u, w\}$: $p(v_e) = p_e$. All edges of the resulting graph will be defined to be perfectly reliable. This construction will at most polynomially increase the size of the graph. However, the size of graphs of bounded treewidth will increase only linearly, and the treewidth also does not increase with this construction (see Section 2.5 for more details).

6.2.3 A List of Network Reliability Problems

In this section, we consider some examples of network reliability problems that can be modelled in our model. It is not difficult to give more examples, and hence, the list of problems below is not complete. It contains only those problems that are proven to be $\#P'$-complete later in this chapter. As for the classical network reliability problems, we will use the abbreviation with brackets to refer to a problem.

- $[2 \leftrightarrow t]$: What is the probability that there is a path in the surviving subgraph between two distinguished vertices u and v?
- $[S \leftrightarrow t]$: What is the probability that all servers (S) are connected in the surviving subgraph?
- $[L \leftrightarrow \geq 1S]$: What is the probability that each client (L) is connected to at least one server (S) in the surviving subgraph?
- $[\geq xS]$: What is the probability that at least x servers are connected to each other in the surviving subgraph?
- $[Ec \geq 1S]$: What is the expected number of components of the surviving subgraph with at least one server?
- $[\geq x_1 L \not\leftrightarrow \geq 1S]$: What is the probability that at least x_1 clients are not connected to a server in the surviving subgraph?
- $[\leq x_2 L \leftrightarrow \geq 1S]$: What is the probability that at most x_2 clients are connected to at least one server in the surviving subgraph?
- $[< x_3 L \not\leftrightarrow \geq 1S]$: What is the probability that less than x_3 clients are not connected to a server in the surviving subgraph?
- $[> x_4 L \leftrightarrow \geq 1S]$: What is the probability that more than x_4 clients are connected to at least one server in the surviving subgraph?
- $[1 - [< x_3 L \not\leftrightarrow \geq 1S]]$: What is the probability that it is not the case that less than x_3 clients are not connected to a server in the surviving subgraph?
- $[\leq yS \not\leftrightarrow L]$: What is the probability that at most y servers are not connected to a client in the surviving subgraph?
- $[\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$: What is the probability that in the surviving subgraph at least x clients are connected to at least one server, while at least y servers are connected to at least one client?
- $[\geq 1L \leftrightarrow \geq xS]$: What is the probability that in the surviving subgraph at least one client is connected to at least x servers?

6.3 Complexity of Network Reliability Problems

More than twenty-five years ago, Rosenthal [86] showed that the classical network reliability problems $[2 \leftrightarrow t]_e$ and $[A \leftrightarrow t]_e$ are NP-hard to approximate. Provan and Ball [75] and Valiant [94] showed that these classical network reliability problems are $\#P$-hard. A similar result was shown independently by Jerrum [59].

More precisely, Valiant [94] shows that computing the number of subgraphs or induced subgraphs of a given graph G such that there is a path between two given vertices s and t is $\#P$-complete. Provan and Ball show in [75] the $\#P$-hardness for the All Terminal Network Reliability Problem. They state that the problem is $\#P$-complete but do not give a proof of the membership of the problem in $\#P$. They also show that it is $\#P$-hard to approximate the two problems, even for the case when all edges have the same reliability. Jerrum [59] showed that the probability that the surviving network is connected, viewed as a polynomial in the edge failure probabilities, is complete for Valiant's class of P-definable polynomials with respect to P-projections.

The complexity class $\#P$ was first defined by Valiant [94] and contains only integer-valued functions. However, notions related to $\#P$ (e.g. $\#P$-completeness) were also used for functions that have rational values (e.g. solutions of the All Terminal Network Reliability Problem). To avoid confusion and formal inadequacies, we will define another class $\#P'$ that is an extension of $\#P$ to include rational-valued functions. $\#P'$ was called $\#P$ in [75]. We use the different notation $\#P'$ to avoid confusion with the original class $\#P$. Valiant's result [94] then implies the $\#P'$-completeness of the Two Terminal Network Reliability Problem for the case when all edges, or all vertices (except possibly s and t), respectively, have the same reliability.

Where NP, NP-hardness and NP-completeness deal, roughly said, with verifying and finding *one* solution to a combinatorial problem, $\#P$, $\#P$-hardness and $\#P$-completeness deal, again roughly said, with establishing the *number* of solutions to a combinational problem. As the problem of counting the number of solutions to a problem is at least as hard as determining if there is at least one solution, $\#P$-complete problems are 'as least as hard', but possibly harder than NP-complete problems.

In this chapter, we prove the problems presented in the list in Section 6.2.3 to be $\#P'$-complete on general graphs. These proofs consist, as usual, of two parts: proofs of membership in $\#P'$ and transformations from problems known to be $\#P'$-hard. Unlike as is common for NP-completeness proofs, the membership part is not entirely trivial, although it is not very complex. And since, as mentioned before, the proofs of membership in $\#P'$ of the network reliability problems are not available in currently easily accessible scientific literature, we give such a proof here (Lemma 82 and Lemma 83) for two of the considered problems; the other problems can be showed to belong to $\#P'$ in the same way.

6.3.1 Complexity Theory Background

Valiant introduced in [94] the notions $\#P$ and $\#P$-completeness to express the hardness of problems that 'count the number of solutions'. More precisely, $\#P$ is the set of integer-valued functions that express the number of accepting computations of a nondeterministic Turing machine of polynomial-time complexity. Let FP represent the class of polynomial-time computable functions. Furthermore, let Σ be the finite alphabet of the input and output of the Turing machines considered in this chapter. The next definition that defines $\#P$ equivalently to Valiant's definition, can be found in [48], which also contains an extensive overview of the theory of counting complexity.

Definition 78 (see [48]). *The complexity class $\#P$ consists of the functions $f : \Sigma^* \Rightarrow \mathbb{N}$ such that there exists a nondeterministic polynomial-time Turing machine M such that for all inputs $x \in \Sigma^*$, $f(x)$ is the number of accepting paths of $M(x)$.*

Provan and Ball [75] extended Valiant's notion of $\#P$ and $\#P$-completeness to deal with functions with a range of multiple or rational values. As problems like the network reliability problems are functions to the rational numbers in $[0, 1]$ and not to the integers, we cannot say, using Valiant's definition, that they belong to $\#P$.

Definition 79. *A language (function) g is* polynomially transformable *to a language (function) f, written as $g \leq f$, if there exists a Turing machine that for any input x, computes $g(x)$ with a polynomial (in the size of x) number of elementary operations and evaluations of language (function) f. (The size of x is the number of bits needed to denote x.) A function f is* $\#P$-hard, *if every function g in $\#P$ is polynomially transformable to f, and f is* $\#P$-complete, *if f belongs to $\#P$ and f is $\#P$-hard.*

If we would use the terminology of [75], we would indeed be able to prove that our network reliability problems are $\#P$-complete. In order not to depart from most of the current complexity theory literature, we restrict our use of the notion of $\#P$ and $\#P$-completeness to the original definition (see Definition 78), and denote the version we use for dealing with another type of functions as $\#P'$. We define $\#P'$ as the class of functions $h : \Sigma^* \Rightarrow \Sigma^*$, that can be computed in polynomial time from input x and a value $f(x)$ for some $f \in \#P$. Whenever needed, we implicitly convert numbers into strings, i.e. we interpret elements in $\mathbb{N}$ or $\mathbb{Q}$ as strings over a fixed alphabet Σ.

Definition 80. *The class $\#P'$ is defined as follows.*

$$\begin{aligned}\#P' = \{\, h \mid\ & h : \Sigma^* \Rightarrow \Sigma^*, \\ & \exists f \in \#P,\ f : \Sigma^* \Rightarrow \mathbb{N}, \\ & \exists g \in FP,\ g : \mathbb{N} \times \Sigma^* \Rightarrow \Sigma^*, \\ & \forall x \in \Sigma^* : h(x) = g(f(x), x) \,\}\end{aligned}$$

Hardness and completeness for this class are defined as usual.

Lemma 81. *Let $h' : \Sigma^* \Rightarrow \Sigma^*$ be a function.*

1. *$\#P \subseteq \#P'$.*
2. *h' is $\#P$-hard $\Longrightarrow$ h' is $\#P'$-hard.*
3. *h' is $\#P$-complete $\Longrightarrow$ h' is $\#P'$-complete.*

Proof. 1. This follows trivially from the definition by choosing g to be the projection on the first argument.

2. For all $f \in \#P$, we have: $f \le h'$, since h' is $\#P$-hard. Hence, there is a deterministic Turing machine M that when given input $x \in \Sigma^*$ computes $f(x)$, using polynomial time and a polynomial number of evaluations of h'. Let $h \in \#P'$, i.e. $h(x) = g(f(x), x)$ for $f \in \#P$, $g \in FP$ and all $x \in \Sigma^*$. We describe a deterministic Turing machine M' that for input $x \in \Sigma^*$ computes $h(x)$, using polynomial time and a polynomial number of evaluations of h'. At input x, M' computes $f(x)$ by simulating machine M. After that, M' computes deterministically in polynomial time $h(x) = g(f(x), x)$, since $g \in FP$. Hence we have: $h \le h'$, for all $h \in \#P'$.

3. This follows directly from 1. and 2. □

6.3.2 $\#P'$-membership

As already mentioned, for showing the completeness of a problem for a complexity class, we have to show the hardness of the problem and the membership in this complexity class. Using Valiant's original definition of $\#P$ [94], we cannot prove the membership of our network reliability problems. Instead, we prove that these problems belong to $\#P'$ by constructing $\#P'$-Turing machines. However, we will only have a close look at the two problems $[2 \leftrightarrow t]$ and $[Ec \ge 1S]$, since the description of the machines is very similar for the other problems.

Often, a nondeterministic Turing machine is described as a machine that can branch in only 2 (or any other constant) computation paths at each step. However, to ease the description of our machines, we will use machines that branch in b_v computation paths at step for vertex v. In our case this is allowed, because such a branch in b_v paths can be simulated by $\lceil \log b_v \rceil$ binary branches, see Figure 6.1. Note that we need $\lceil \log b_v \rceil$ bits to represent b_v in the input x. Hence, we see easily that the first n steps of our machines can be simulated by $\sum_{v \in V} \lceil \log b_v \rceil$ steps of a machine with only binary branches. This is linear in the input size. If b_v is not a power of 2, in some branches, we stop branching at an earlier time. Thus, the overall length of a computation path after the 'blowup' due to a restriction to binary branches is still bounded by a polynomial.

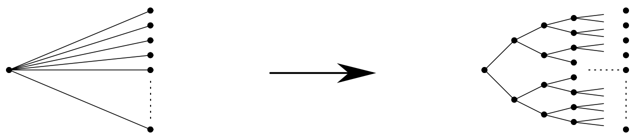

Figure 6.1. Simulating one b_v-branch by $\lceil \log b_v \rceil$ binary branches.

Lemma 82. $[2 \leftrightarrow t] \in \#P'$

Proof. To prove the membership in $\#P'$, we specify a function $h \in \#P'$ by describing functions $f \in \#P$ and $g \in FP$ with $h(x) = g(f(x), x)$ for all $x \in \Sigma^*$. The description of h can be considered as a description of a $\#P'$ Turing machine.

To describe f, we give a $\#P$ Turing machine M with $f(x) = \#M(x)$, where $\#M(x)$ is the number of accepting computation paths of machine M with input x. M is constructed as follows. During the first $n = |V|$ steps, M considers each vertex of G, one at a time. In the step for vertex v, M branches b_v times, where in a_v branches the vertex v is present in the graph G and in $b_v - a_v$ branches, vertex v is not present in G. (The numbers a_v and b_v are nonnegative integers specifying the probability $p(v) = \frac{a_v}{b_v}$ as in Section 6.2.1.) After considering each vertex, M decides deterministically on each computation path, whether the desired property is fulfilled. In our case checking the desired property is checking whether u and w are connected in the surviving subgraph corresponding to the considered computation path of M. If this is the case, M accepts on this path, otherwise it rejects.

The probability that a vertex v is present in the graph is encoded in the total number of branches b_v at the step for vertex v, of which in exactly a_v branches vertex v is present in the graph. After fixing the state of each vertex, i.e. after n steps, each path represents exactly one surviving subgraph. Note, however, that each surviving subgraph G' might be represented by more than one such path, depending on the occurrence probability of G'. Hence, it is easy to see, that the probability $h(x)$ that u and w are connected is:

$$h(x) = \frac{\#M(x)}{\text{total number of paths of } M(x)}$$

The number of accepting paths $\#M(x)$ is given as $f(x)$, but the total number of paths of M is not given. However, this number can be computed in polynomial time by function g as the product over all vertices v of b_v:

$$h(x) = g(f(x), x) = \frac{f(x)}{\prod_{v \in V} b_v}$$

□

The method used above works also for all network reliability problems considered in this chapter that have the form: 'What is the probability that the surviving subgraph has property X?' However, problem $[Ec \geq 1S]$ has a different form: 'What is the expected number of components of the surviving subgraph that have property Y?' and hence, we use a slightly different Turing machine and different functions to prove its $\#P'$-membership.

Lemma 83. $[Ec \geq 1S] \in \#P'$

Proof. Similar to the proof of Lemma 82, we give functions f (i.e. a Turing machine M, with $f(x) = \#M(x)$) and g.

M is constructed as follows. The first n steps are the same as of the machine described in the proof of Lemma 82. However, after these steps, (i.e. after the surviving subgraph on each computation path so far, is completely defined), machine M works differently. Note that for the expected number $h(x)$ of components of the surviving subgraph with at least one server we have:

$$1 \leq h(x) \leq n = |V|$$

Furthermore, note that for each fixed surviving subgraph G', the number $c(G')$ of components with at least one server is a number in $\mathbb{N}$. After the first n steps M branches on each computation path into exactly $c(G')$ accepting paths, where G' is the surviving subgraph defined by the corresponding computation path. If $c(G') = 0$ then M does not branch any further but rejects on this path. We use the same function g as in the proof of Lemma 82.

$$h(x) = g(f(x), x) = \frac{f(x)}{\prod_{v \in V} b_v}$$

We can see in the following way that this is indeed correct. When we consider the computation tree of M as a probability distribution, we see that after the first n steps every path has the same probability. The different probabilities of the surviving subgraphs G' are expressed by an appropriate number of paths that define G'. This automatically assigns the correct weights to $c(G')$ for all G', when computing the expected number of components with at least one server. □

Due to the last two lemmas and the similarity of the problems in the list in Section 6.2.3, the correctness of the following corollary is easy to see.

Corollary 84. *The following network reliability problems are in $\#P'$:* $[2 \leftrightarrow t]$, $[S \leftrightarrow t]$, $[L \leftrightarrow \geq 1S]$, $[\geq xS]$, $[Ec \geq 1S]$, $[\geq x_1L \not\leftrightarrow \geq 1S]$, $[\leq x_2L \leftrightarrow \geq 1S]$, $[< x_3L \not\leftrightarrow \geq 1S]$, $[> x_4L \leftrightarrow \geq 1S]$, $[1 - [< x_3L \not\leftrightarrow \geq 1S]]$, $[\leq yS \not\leftrightarrow L]$, $[\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$ *and* $[\geq 1L \leftrightarrow \geq xS]$.

When we want to show membership in $\#P'$ for network reliability problems where *edges* can fail, the proof of Lemma 82 is changed as follows: The first $n = |V|$ branching steps which make ‘decisions’ for every vertex, would be replaced by $m = |E|$ branching steps making the decisions whether this edge is ‘up’ or ‘down’ for every edge on a certain computation path.

6.3.3 $\#P'$-hardness

In this section, we prove the $\#P'$-hardness of the problems given in the list of network reliability problems in Section 6.2.3. For each problem we give a simple transformation from a known $\#P$-hard problem, thus proving the considered problem to be $\#P$-hard. The $\#P'$-hardness of the problems then follows directly from Lemma 81.

Lemma 85. *The following network reliability problems are $\#P$-hard:* $[2 \leftrightarrow t]$, $[S \leftrightarrow t]$, $[L \leftrightarrow \geq 1S]$, $[\geq xS]$, $[Ec \geq 1S]$, $[\geq x_1L \not\leftrightarrow \geq 1S]$, $[\leq x_2L \leftrightarrow \geq 1S]$, $[< x_3L \not\leftrightarrow \geq 1S]$, $[> x_4L \leftrightarrow \geq 1S]$, $[1 - [< x_3L \not\leftrightarrow \geq 1S]]$, $[\leq yS \not\leftrightarrow L]$, $[\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$ *and* $[\geq 1L \leftrightarrow \geq xS]$.

Proof. For each of the problems, we give a transformation from a problem that is known to be $\#P$-hard.

$[2 \leftrightarrow t]_e \leq [2 \leftrightarrow t]$:

Given an instance of $[2 \leftrightarrow t]_e$, we construct an instance of $[2 \leftrightarrow t]$ by simulating edges as described in Section 6.2.2. All vertices v that do not simulate an edge are perfectly reliable, i.e. they have $p(v) = 1$. It is easy to see that this is a correct transformation. The same transformation can be used to show that $[2 \leftrightarrow t]$ with the additional requirement that both distinguished vertices are perfectly reliable is $\#P$-hard.

$[2 \leftrightarrow t] \leq [S \leftrightarrow t]$:

Suppose we are given an instance of $[2 \leftrightarrow t]$ with two distinguished vertices u and v. We turn this into an instance of $[S \leftrightarrow t]$ by simply letting u and v be the only servers. Again, as the transformation does not change the value of the requested probability, we have a correct transformation from $[2 \leftrightarrow t]$ to $[S \leftrightarrow t]$.

$[A \leftrightarrow t]_e \leq [S \leftrightarrow t]$:

Given an instance of $[A \leftrightarrow t]_e$, we construct a $[S \leftrightarrow t]$-instance by simulating edges as described in Section 6.2.2. We let the set of servers be the set of vertices that do not simulate an edge. Note that these are perfectly reliable. This transformation does not change the value of the requested probability.

$[S \leftrightarrow t] \leq [L \leftrightarrow \geq 1S]$:

Let an instance of $[S \leftrightarrow t]$ be given. We assume that we do not have clients, and if we have, we delete their 'client'-flag, turning them to non-clients. Now, we choose an arbitrary server. This is again a server in the new to be formed instance. All other servers in the $[S \leftrightarrow t]$-instance become clients in the new instance. Thus, we have formed an instance of $[L \leftrightarrow \geq 1S]$. Now, one can note that all servers are connected in the $[S \leftrightarrow t]$-instance, if and only if each client is connected to at least one server in the $[L \leftrightarrow \geq 1S]$-instance. Hence, this transformation does not change the value of the requested probability.

$[2 \leftrightarrow t] \leq [\geq xS]$:

Suppose we are given an instance of $[2 \leftrightarrow t]$. Taking the two distinguished vertices as the only two servers and setting $x = 2$ gives an instance of $[\geq xS]$ with the same probability value.

$[2 \leftrightarrow t] \leq [Ec \geq 1S]$:

More precisely, we give a transformation from the version of $[2 \leftrightarrow t]$ where we assume that both distinguished vertices are perfectly reliable. We have argued above that this version is also $\#P$-hard, so this is not a restriction. Let a $[2 \leftrightarrow t]$-instance G be given, where the two distinguished vertices are perfectly reliable. Mark the two distinguished vertices u and v as the only servers, which gives us a $[Ec \geq 1S]$-instance G'. Let P be the probability that there is a path between u and v in the surviving subgraph $\bar{G}$ of G, and let E be the expected number of components with at least one server in the surviving subgraph $\bar{G}'$ of G'. E is the sum of the probability that the two servers are in one component in the surviving subgraph and two times the probability that the two servers are in two different components in the surviving subgraph. This equals the sum of the probability that there is a path between u and v in the surviving subgraph and two times the probability that there is no path between u and v in the surviving subgraph. Thus, we have $E = 2 - P$, and the transformation is easy to see.

$[\geq x_1 L \not\leftrightarrow \geq 1S] \equiv [\leq x_2 L \leftrightarrow \geq 1S]$:

This is easy to see, because at least x_1 clients are not connected to a server, if and only if at most $x_2 := n_c - x_1$ clients are connected to at least one server in the surviving subgraph.

$[< x_3 L \not\leftrightarrow \geq 1S] \equiv [> x_4 L \leftrightarrow \geq 1S]$:

This is also easy to see, since less than x_3 clients are not connected to a server, if and only if more than $x_4 := n_c - x_3$ clients are connected to at least one server in the surviving subgraph.

In the previous two transformations, the symbol '$\equiv$' does not only mean equivalence concerning the hardness of the problems, but it could also be understood as equality of the corresponding solutions, i.e. the solutions of the equivalent problems are equal, respectively, and hence, the transformations follow trivially. In the next transformation, we use '$\equiv$' in the same way, however with a complementary problem:

For a problem $[Z]$ in our list, we define the complementary problem $[1 - [Z]]$. For all instances σ of $[Z]$, we have $[Z]\sigma = 1 - [1 - [Z]]\sigma$, i.e. $[1 - [Z]]$ maps an instance to 1 minus the value $[Z]$ maps the instance to. So, when $[Z]$ gives us the probability that a certain property holds for the surviving subgraph, $[1 - [Z]]$ gives the probability that this property does *not* hold for the surviving subgraph. Concerning the complexity of the problems, clearly $[Z] \equiv [1 - [Z]]$.

$[\geq x_1 L \not\leftrightarrow \geq 1S] \equiv [1 - [< x_3 L \not\leftrightarrow \geq 1S]]$:

This is again not difficult to see: At least x_1 clients are not connected to at least one server, if and only if it is not the case that less than $x_3 := x_1$ clients are not connected to at least one server in the surviving subgraph.

$[L \leftrightarrow \geq 1S] \leq [> x_4 L \leftrightarrow \geq 1S]$:

This transformation is easy, since we do not have to modify the instance. We simply solve the $[> x_4 L \leftrightarrow \geq 1S]$-problem for $x_4 := n_c - 1$ and get immediately an answer for the $[L \leftrightarrow \geq 1S]$-problem.

$[< x_3 L \not\leftrightarrow \geq 1S] \leq [\leq yS \not\leftrightarrow L]$:

We interchange the roles of servers and clients and we choose $y := x_3 - 1$.

$[L \leftrightarrow \geq 1S] \leq [\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$:

The $[\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$-problem contains the $[L \leftrightarrow \geq 1S]$-problem as a special case: $x := n_c$ and $y = 0$.

$[\geq xS] \leq [\geq 1L \leftrightarrow \geq xS]$:

Given a $[\geq xS]$-instance without any clients, we add to each server a pendant which is a private client for this server. This results in a $[\geq 1L \leftrightarrow \geq xS]$-instance. Now, it is easy to see, that this is a correct transformation. □

6.3.4 Computing the Numerators of Probabilities

In this section, we take a different look at the problems from our list that ask to compute the value of a probability, (i.e. all problems except $[Ec \geq 1S]$). In the previous sections, we have defined the class $\#P'$ and showed our problems to be $\#P'$-complete. Instead of defining a new complexity class, we will modify the problems at hand in this section, and we will show them to be $\#P$-complete.

The problems considered here compute a probability, and each of these probabilities is a rational number, which can be expressed as a fraction with an easily computable denominator and a numerator that is $\#P$-complete to compute. As before, we assume that each vertex $v \in V$ has a reliability $p(v) = a_v/b_v$, with a_v a nonnegative integer, and b_v a positive integer. Let

$$D = \prod_{v \in V} b_v$$

be the product of all denominators of the reliabilities of the vertices. The following lemma can be easily be observed.

Lemma 86. *For each induced subgraph G' of G, the probability that the surviving subgraph equals G' is an integer multiple of $1/D$.*

Given a function $[X]$ (assumed to be one of the considered network reliability problems), we define the function $[D * [X]]$ by taking for each input ϕ, $[D * [X]]\phi = D * [X]\phi$. For instance, $[D * [S \leftrightarrow t]]$ asks for D times the probability that all servers are connected in the surviving subgraph.

Theorem 87. *The following problems are functions to the nonnegative integers and $\#P$-complete: $[D * [2 \leftrightarrow t]]$, $[D * [S \leftrightarrow t]]$, $[D * [L \leftrightarrow \geq 1S]]$, $[D * [\geq xS]]$, $[D * [\geq x_1 L \not\leftrightarrow \geq 1S]]$, $[D * [\leq x_2 L \leftrightarrow \geq 1S]]$, $[D * [< x_3 L \not\leftrightarrow \geq 1S]]$, $[D * [> x_4 L \leftrightarrow \geq 1S]]$, $[D * [1 - [< x_3 L \not\leftrightarrow \geq 1S]]]$, $[D * [\leq yS \not\leftrightarrow L]]$, $[D * [\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]]$ and $[D * [\geq 1L \leftrightarrow \geq xS]]$.*

Proof. Consider some property Y.The probability that the surviving subgraph has this property Y is the sum over all induced subgraphs G' of G with property P of the probability that the surviving subgraph equals G'. Thus, by Lemma 86, the probability of Y is an integer multiple of $1/D$. Hence, we have that $[D * [Y]]$ is a function to the integers.

The proof of membership in $\#P$ of $[D * [2 \leftrightarrow t]]$ is similar, and even slightly easier than that of Lemma 82: note that the requested value precisely equals the number of accepting paths $\#M(x)$. For the other problems, membership in $\#P$ can be derived in the same way.

$\#P$-hardness for the problems can be shown with basically the same proof as for Lemma 85. □

So, we have for each of the considered network reliability problems apart from $[Ec \geq 1S]$ that these are functions to the rational numbers, which can be written as the quotient of a $\#P$-complete function (a $[D * \cdots]$ variant of the problem) and a function in P (the product of all values b_v). The problems of the type $[D * \cdots]$ hence can be seen as *numerator*-versions of the original problems. Note that the size of D is polynomial in the size of the given graph.

6.4 Concluding Remarks

In this chapter, we introduced the classical network reliability problems. We generalised these problems and presented the graph-theoretical model on which the problems are based. We gave a list of some example problems according to our model, and we proved these problems to be $\#P'$-complete.

For this, we have given a formal definition of an extension $\#P'$ of $\#P$ to include rational-valued functions. The list given in Section 6.2.3 is obviously not complete. However, it is not hard to show the same type of results for other network reliability problems of a similar 'flavour'. The following theorem which follows directly from Corollary 84 and Lemma 85, summarises the results of this chapter.

Theorem 88. *The following network reliability problems are $\#P'$-complete:* $[2 \leftrightarrow t]$, $[S \leftrightarrow t]$, $[L \leftrightarrow\geq 1S]$, $[\geq xS]$, $[Ec \geq 1S]$, $[\geq x_1L \not\leftrightarrow\geq 1S]$, $[\leq x_2L \leftrightarrow\geq 1S]$, $[< x_3L \not\leftrightarrow\geq 1S]$, $[> x_4L \leftrightarrow\geq 1S]$, $[1 - [< x_3L \not\leftrightarrow\geq 1S]]$, $[\leq yS \not\leftrightarrow L]$, $[\geq xL \leftrightarrow S\wedge \geq yS \leftrightarrow L]$ *and* $[\geq 1L \leftrightarrow\geq xS]$.

The hardness proofs were obtained through simple transformations from known $\#P$-hard problems. Figure 6.2 gives an overview which transformations were used. A path from left to right corresponds to a chain of transformations and vertical lines represent transformations showing the hardness equivalence of the corresponding problems. The $\#P$-hardness of some network reliability problems

$$\begin{tikzcd}
 & & & {[\geq x_1L \not\leftrightarrow\geq 1S]} \arrow[d, dash] & \\
 & & & {[\leq x_2L \leftrightarrow\geq 1S]} \arrow[d, dash] & \\
 & & {[Ec \geq 1S]} & {[< x_3L \not\leftrightarrow\geq 1S]} \arrow[d, dash] \arrow[r, dash] & {[\leq yS \not\leftrightarrow L]} \\
{[2 \leftrightarrow t]_e} \arrow[r, dash] & {[2 \leftrightarrow t]} \arrow[ur, dash] \arrow[r, dash] \arrow[dr, dash] \arrow[ddr, dash] & {[L \leftrightarrow\geq 1S]} \arrow[r, dash] \arrow[dr, dash] & {[> x_4L \leftrightarrow\geq 1S]} & \\
 & & {[\geq xS]} \arrow[dr, dash] & {[\geq xL \leftrightarrow S\wedge \geq yS \leftrightarrow L]} & \\
{[A \leftrightarrow t]_e} \arrow[rr, dash] & & {[S \leftrightarrow t]} & {[\geq 1L \leftrightarrow\geq xS]} &
\end{tikzcd}$$

Figure 6.2. The transformations in this chapter

on general graphs, motivates us and justifies to look at graphs with a special structure to enable efficient algorithms. Graphs of bounded treewidth have such a special structure, and the study of network reliability problems on these graphs is the topic of the next chapter.

7 Network Reliability: Graphs of Bounded Treewidth

In this chapter, we develop a framework based on dynamic programming for solving network reliability problems on graphs of bounded treewidth. The graph-theoretical model we are using, assumes that only vertices can fail (independently of each other), and that we have two distinguished sets of vertices: S (a set of servers) and L (a set of clients). The problems we are looking at include for example the following questions: ‘What is the probability that all clients are connected to at least one server?’ (problem $[L \leftrightarrow \geq 1S]$ from Chapter 6) and ‘What is the expected number of components of the graph of the non-failed elements that contain a vertex of S?’ (problem $[Ec \geq 1S]$). More generally, the problems ask for the probability that the surviving subgraph has a certain property. Such a property defines the connectivity between vertices of the set S, and also the connectivity between vertices of S and vertices of L. Furthermore, the number of connected components which contain a vertex of S or L can be dealt with. More details about the underlying model, some network reliability problems and proofs of their computational hardness can be found in Chapter 6. In this chapter, many of these questions are shown to be solvable in linear or polynomial time if the input graph G has bounded treewidth.

Much work has been done on algorithms for graphs of bounded treewidth, as well as for special network reliability problems. Arnborg and Proskurowski considered in [4] the case that only edges can fail and that nodes are perfectly reliable. They gave a linear time algorithm for this problem on partial k-trees, i.e. graphs of treewidth at most k. Mata-Montero generalised this to partial k-trees in which edges as well as nodes can fail [72].

We will present a framework on graphs of bounded treewidth for answering a large number of questions, which can be more complicated and more general than the classical network reliability problems. In Section 7.1, we describe the general technique we are using. This technique in general requires exponential time for solving network reliability problems. Its consideration can be seen as a preparation for the method with equivalence classes, which we will consider in Section 7.2. There, we will see how equivalence classes can be used to reduce the number of objects considered by the method at hand. The smaller the number of objects the smaller the running time of the corresponding algorithms. Depending on the problem, we are able to give equivalence relations that are coarse enough to enable linear time algorithms. A closer look at equivalence relations and solvable problems is taken in Section 7.3. Equivalence relations must fulfil certain properties if we would like to use them in our framework. We will consider a way of creating equivalence relations that automatically meet these requirements. Section 7.4 is devoted to an extension of our framework. We consider an additional idea with which we can quickly answer questions that differ from our original questions. We will see that it is possible to modify or extend our framework to answer even more questions or to obtain faster running times. This chapter is based on previous work [96, 97].

7.1 A General Technique

Rosenthal introduced in [86] a decomposition method for computing network reliability measures. This method was applied by Carlier and Lucet to solve the network reliability problems with edge and node failures [29]. Furthermore, with this approach Carlier, Manouvrier and Lucet solved the 2-edge connected reliability problem for graphs of bounded pathwidth [71].

We will generalise both the type of the problems which can be solved, and the method used in [29] and [71]. Instead of looking at graphs of bounded pathwidth, we will deal with the more general class of graphs of bounded treewidth. The dynamic-programming algorithm described in this chapter stores solutions for subproblems in tables. The more information needed to answer a question, the bigger the table and hence, the higher the running time of the algorithm. This starts with linear time and can increase to exponential time. Algorithms that require exponential time are necessary for solving problems in which most or all of the possible information must be preserved. Hence, no decrease of information or running time is possible.

7.1.1 Specific Definitions

A *scenario* f assigns to each vertex v its state of operation: v is up ($f(v) = 1$) or down ($f(v) = 0$). Hence, a scenario describes which vertices are up and which are down in the network. For the scenario f, $W^{f=1}$ is the set of all vertices of W which are up and $W^{f=0} = W \setminus W^{f=1}$ is the set of all vertices of W which are down, for $W \subseteq V$. The probability of a scenario f (for the whole graph G) is:

$$\Pr_V(f) = \prod_{v \in V^{f=1}} p(v) \cdot \prod_{v \in V^{f=0}} 1 - p(v)$$

Clearly, there are $2^{|V|}$ scenarios. The elements in $L \subseteq V$ represent special objects, the *clients*, and the elements of $S \subseteq V$ are the *servers*, where $n_S = |S|$ and $n_L = |L|$. We say $W \subseteq V$ is connected in G, if for any two vertices of W there is a path joining them. One can easily simulate edge failures by introducing a new vertex on each edge, and then using the methods for networks with only node failures. Note that this operation leaves the treewidth invariant. For more details on how to simulate edge failures, see Sections 6.2.2 and 2.5.

Summary of Notations

Let $G = (V, E)$ be a graph.

- $p : V \to [0, 1]$ assigns to each vertex its probability to be up
- $f : V \to \{0, 1\}$ assigns to each vertex its state for this scenario f
- $W^{f=q} = \{v \in W \mid f(v) = q\}$, for $q \in \{0, 1\}$, $W \subseteq V$, $f : V \to \{0, 1\}$
- $L \subseteq V$ is the set of clients of G, $n_L = |L|$
- $S \subseteq V$ is the set of servers of G, $n_S = |S|$

We now give the definitions of blocks and representatives. These notions play an important role for the main methods given in this chapter. We assume to have a nice tree-decomposition $(T, X) = ((I, F), \{X_i \mid i \in I\})$ of $G = (V, E)$. Since T is a binary rooted tree, the ancestor relation of nodes of the tree is well defined. As described in many articles, e.g. [8] and [10], we will use a bottom-up approach using this tree-decomposition. Each node i of the tree can be viewed as the root of a subtree. We will compute partial solutions for the subgraphs corresponding to these subtrees step by step. These partial solutions are contained in tables associated to the roots of the subtrees. To compute this information, we only need the information stored in the children of a node, which is typical for dynamic-programming algorithms using tree-decompositions. The following definition provides us with the terminology for the subgraphs.

Definition 89. *Let $G = (V, E)$ be a graph and $(T = (I, F), X = \{X_i \mid i \in I\})$ a nice tree-decomposition of G. Then:*

- $V_i = \{v \in X_j \mid j = i \text{ or } j \text{ is a descendant of } i\}$
- $G_i = G[V_i]$
- $L_i = L \cap V_i$
- $S_i = S \cap V_i$

Blocks

For a certain subgraph $G_i = G[V_i]$, we consider the scenarios of G restricted to V_i. A scenario f^1 causes G_i to decompose into one or more connected components $O_1, ..., O_q$, i.e. $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$ with $V(O_x) \cap V(O_y) = \emptyset$ for $x \neq y$ and $1 \leq x, y \leq q$. For these components, we consider the intersection with X_i. These intersection sets $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$ are called *blocks* and we use the following notation:

$$blocks_i(f^1) = \{B_1, ..., B_q\}$$

Thus, each scenario f^1 specifies a multiset of blocks. It is a multiset since more than one component may have an empty intersection with X_i. When using the symbol '$\cup$' between multisets (of blocks), it means the 'multi-union' of the sets, i.e. multiple occurrences are not deleted and the result is again a multiset.

We augment the blocks with L-flags and/or S-flags. Since we are looking at reliability problems with a set L of clients and a set S of servers, it is important to store the information whether $O_j \cap L^{f=1} \neq \emptyset$ and/or $O_j \cap S^{f=1} \neq \emptyset$. If either of this is the case we add $|O_j \cap L^{f=1}|$ L-flags and/or $|O_j \cap S^{f=1}|$ S-flags to the block B_j. Hence, the blocks for a scenario f reflect the connections between the vertices of X_i in $G[V_i^{f=1}]$ as well as the number of servers and clients of the components of $G[V_i^{f=1}]$. We use the notation '#Xflags(B)' to refer to the number of X-flags of block B, ($X \in \{S, L\}$).

Definition 90. *A* block *B of the graph G_i for scenario f is a (perhaps empty) subset of X_i extended by x L-flags and y S-flags. For $\{v_1, ..., v_t\} \subseteq X_i$, we write $B = \{v_1, ..., v_t\}^{x \cdot L}_{y \cdot S}$.*

For small x and y instead of $B = \{v_1, ..., v_t\}^{x \cdot L}_{y \cdot S}$, we can also write $B = \{v_1, ..., v_t\}^{L..L}_{S..S}$ where the number of L-flags equals x and the number of S-flags equals y.

Example of Blocks

Let $X_i = \{u, v, w\}$. Depending on G_i, example blocks could be:

- $B_1 = \{uv\}$; u and v are up and connected within one component. This component contains no vertex of S nor of L.
- $B_2 = \{\ \}^{LL}_{S}$; this represents a component with empty intersection with X_i. Furthermore this component contains two clients and one server.
- $B_3 = \{w\}_{SS}$; this represents a component with nonempty intersection with X_i, namely w. This component contains no clients but two servers.

Representatives

For $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$, we have the following blocks $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$, which can be used to represent f^1. A block B of such a representation has x L-flags, if and only if the component O corresponding to B contains exactly x vertices of L. The same is also true for the S-flags.

To each scenario of node i, we associate the multiset of its blocks. We call this representation of a scenario f^1 its *representative* C^1:

$$C^1 = \mathit{blocks}_i(f^1) = \{B_1, ..., B_q\}$$

for $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$ and $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$. To G_i we associate a multiset of representatives, one for each possible scenario. Note that two different scenarios f^1 and f^2 can have identical representatives $C^1 = C^2$, but these will be considered as different objects in the multiset of the representatives of G_i. In Section 7.2.2, equivalence relations on scenarios and hence on representatives are defined. Thus, equivalence classes will be formed that can correspond to more than one scenario. It is then possible and perhaps also sensible to define C^1 and C^2 as equivalent. Furthermore, it will be seen that the algorithmic techniques for computing the collection of representatives of the graphs G_i and their corresponding probabilities will carry over with little modification to algorithms for computing the collections of equivalence classes for many equivalence relations.

Definition 91. *A* representative $C^1 = \{B_1, ..., B_q\}$ *of* G_i *is a multiset of blocks of* G_i*, such that the scenario* f^1 *specifies* $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$*. It must hold that #Lflags(*B_j*)=* $|O_j \cap L|$ *and #Sflags(*B_j*)=* $|O_j \cap S|$*, for* $j = 1, ..., q$*, and* $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$*.*

The number of representatives per node is a crucial point in the algorithm running times as will be demonstrated. As mentioned before, we therefore examine equivalence relations between scenarios in Section 7.2.2 to reduce the number of representatives.

Example of Representatives

Let $X_i = \{u, v, w\}$. Some of the possible representatives are:

- $C^1 = (\{uv\}_S^{LL}\{w\}\{\ \}^L\{\ \}_S)$; all vertices are up, u, v are in one component containing two clients and one server, w is in another component without clients and servers. There are two components with empty intersection with X_i, one of them contains a client and the other a server.
- $C^2 = (\{vw\}\{\ \}\{\ \}\{\ \}_S)$; v and w are up and connected. There are three components with empty intersection with X_i, one of them contains a server.
- $C^3 = (\{vw\}\{\ \}_S)$; v and w are up and connected. There is one component with empty intersection with X_i and this component contains a vertex of S.

Here, the representatives C^2 and C^3 are somehow ‘similar’. They only differ in the number of empty blocks without flags. Such blocks neither give information about clients or servers nor can be used to create additional connections. Hence, it can be sensible to define C^2 and C^3 to be equivalent (see Section 7.2.2).

We already saw that representatives are made of blocks. When considering our method, we will see that we associate a set of representatives to each node of the tree-decomposition. Therefore, we introduce the following notation:

- n_c is the maximum number of representatives for a node.
- n_b is the maximum number of blocks of a representative.

7.1.2 Description of the Technique

After these necessary notations, we now describe our method where tree-decompositions of a network can be used to solve variations of the network reliability problem. As already mentioned, we compute the set of representatives for every node of the nice tree-decomposition using a bottom-up approach. We will not only compute the representatives C, but also their associated probabilities $\Pr(C)$. For this computation, we give algorithms for each of the four types of nodes that are used in a nice tree-decomposition. Before looking at these algorithms, we consider a subroutine to repair the structure of a representative. This subroutine is used in the algorithms for introduce- and join-nodes.

For a representative C^1 of node i, $\Pr(C^1)$ is the probability that G_i is in scenario f^1. To compute $\Pr(C^1)$, we could simply compute $\Pr(f^1)$. We could do this for all representatives C of node i. Nevertheless, Lemma 92 and 94 and similar easy statements for leaf and forget nodes, will make evident the fact that we only need the already computed representatives and their probabilities of the children of the node, insofar as they exist. This is a relevant point for applying the dynamic-programming scheme, described in [8] or [10].

Starting at the leaves of the nice tree-decomposition, we compute for each node all representatives and their probabilities that are possible for this node. In this computation, the representatives of the children of a node play a decisive role. When 'walking up' the tree-decomposition to a node i the representatives will be refined and split into other representatives representing the states of the vertices of X_i and their connections. Later in Section 7.2.2 we will see that representatives are not only refined, while walking up the tree, but at the same time, some representatives will 'collapse' into one equivalence class, since they become equivalent. Once we have the probabilities of all representatives of all nodes, especially of the root, we can easily solve problems by computing the requested probabilities.

7.1.3 Repairing Representatives

Before considering the particular algorithms, we describe a procedure that is a subroutine of the introduce- and join-node-algorithm. It is used to repair the structure of a representative. The procedure REPAIR receives as input a multiset of blocks. This multiset is not necessarily a representative, as some blocks may contain common vertices in X_i. The REPAIR procedure 'repairs' or 'corrects' the multiset by unifying such blocks.

By introducing a new vertex or joining two representatives (see Section 7.1.4 for the algorithms), a representative can contain blocks with nonempty intersection. Between two components O_a and O_b corresponding to such blocks B_a and B_b exists a connection via a vertex of $B_a \cap B_b$. This means that after introducing a new vertex or joining representatives, we have a larger component $O = O_a \cup O_b$, which has to be represented by only one block B. Having two blocks with a nonempty intersection violates the proper structure of a representative. Hence, we *merge* or *unify* such nondisjoint blocks B_a and B_b into one block B. Since blocks are sets of vertices, multiple appearances of a vertex are not possible. The new component O contains all the clients (i.e. vertices $v \in L$) of O_a and O_b. We have to add to B the number of L-flags of blocks B_a and the number of L-flags of B_b. On the other hand, we must subtract the number of clients which belong to both blocks B_a and B_b, because they are counted twice. The same applies to servers. Here is the code of the procedure to repair representative C:

```
REPAIR($C$)

while $\exists B_a, B_b \in C$ with $a \neq b$ and $B_a \cap B_b \neq \emptyset$ do
    $B := B_a \cup B_b$
```

```
        add (#$L$flags($B_a$) + #$L$flags($B_b$) − $|L \cap B_a \cap B_b|$) $L$-flags to $B$;
        add (#$S$flags($B_a$) + #$S$flags($B_b$) − $|S \cap B_a \cap B_b|$) $S$-flags to $B$;
        $C := \{B\} \cup C \setminus \{B_a, B_b\}$
    endwhile
```

The while loop can be implemented to take at most $O(n_b^2)$ iterations, where n_b is the maximum number of blocks in one representative. One block can have at most $k = O(\max_{i \in I} |X_i|)$ elements (k is the treewidth of the considered graph). Thus, the condition and the body of the while loop is executable in $O(k^2)$ steps. The REPAIR algorithm needs altogether $O(k^2 \cdot n_b^2)$ time.

7.1.4 Computation of the Representatives and their Probability

In the following, we use i to denote a node of the tree T. Node i may have $0, 1$ or 2 children, depending on the type of i. These children are referred to as j or j_1, j_2. Accordingly, C_i denotes a representative of i, which is computed using the representative(s) C_j (C_{j_1}, C_{j_2}) of node(s) j (j_1, j_2), respectively. We will have a look at each type for node i. When handling representatives, a lower index refers to the node the representative belongs to, and an upper index is used to differentiate different representatives belonging to the node considered.

Leaf Nodes

For leaves i of T we have a relatively simple algorithm, since $|X_i| = 1$ and leaves have no children. Let $v \in X_i$. So, we have the following possibilities, either v is up or down, and we can have $v \in L$ or $v \notin L$ and $v \in S$ or $v \notin S$. We distinguish these cases:

- v is up: this results in a representative $C_i^1 = \{\ \{v\}\ \}$, with $\Pr(C_i^1) = p(v)$. We add C_i^1 to the set of representatives of node i.
- v is down: here we have another representative $C_i^2 = \{\ \}$, with $\Pr(C_i^2) = 1 - p(v)$. Again, we add it to the set of representatives of node i.

If v is up and $v \in L$ and/or $v \in S$, we must add one L- and/or S-flag to the block. An algorithm performing the appropriate steps is very simple and would clearly need constant time per leaf. Since $|V_i| = 1$, we have only two scenarios.

Introduce Nodes

Suppose i is an introduce node with child j, such that $X_i = X_j \cup \{v\}$. We use the representatives and their probabilities of node j to compute the representatives and their probabilities of node i. For any state of v (up or down, in L or not in L, in S or not in S), we compute the representatives and add them to the set of representatives of node i. This is done by considering the following two cases for each representative C_j of j:

- v is up: we add v to each block $B \in C_j$ which contains a neighbour of v, since there is a connection between v and the component corresponding to B. If v is a client or a server, then we add an L- or S-flag to block B. Now, we apply the algorithm in Section 7.1.3 to repair the structure of a representative. The probability of this new representative C_i^1 is: $\Pr(C_i^1) = p(v) \cdot \Pr(C_j)$.
- v is down: we do not make any changes to the representative C_j to obtain C_i^2, since no new connections can be made and no flags were added. However, we compute the probability: $\Pr(C_i^2) = (1 - p(v)) \cdot \Pr(C_j)$.

The following code formalises this procedure:

```
Introduce-Node-Algorithm

for every representative $C_j$ of node $j$ do
    /* $v$ is up */
    $C_i := C_j$
    for each block $B$ of $C_i$ do
        if $B \cap N(v) \neq \emptyset$
        then $B := B \cup \{v\}$
            if $v \in L$ then add one $L$-flag to $B$ endif
            if $v \in S$ then add one $S$-flag to $B$ endif
        endif
    endfor
    REPAIR($C_i$)
    $\Pr(C_i) := \Pr(C_j) \cdot p(v)$
    add $C_i$ to the set of representatives of node $i$
    /* $v$ is down */
    $C_i := C_j$
    $\Pr(C_i) := \Pr(C_j) \cdot (1 - p(v))$
    add $C_i$ to the set of representatives of node $i$
endfor
```

Lemma 92. *Given all representatives and their probabilities of the child j of an introduce node i, the* `Introduce-Node-Algorithm` *computes the representatives and their probabilities for node i correctly.*

Proof. Note that we can compute the representatives and their probabilities for node i by considering every possible scenario for node i. However, recall that v is the only vertex in $V_i \setminus V_j$. So, each scenario for node i can be obtained by taking a scenario for node j and extending it by specifying whether v is up or down. All possible scenarios for node j are represented by the representatives of j, whose probabilities are known. For that reason, it is sufficient to combine the representatives of j with v to create the representatives of i. To do this we consider both v being up and down for each representative C_j of j and make the appropriate modifications. In this way, we implicitly look at every scenario which is possible for G_i.

It is not hard to see that the algorithm realises the description above. To demonstrate that this is true, we consider the representative C_j of a scenario f of j, hence $C_j = blocks_j(f)$. Two cases: v is up, or v is down, have to be considered:

Case 1 'v is up': We extend f by $f(v) = 1$, i.e. for all $w \in V_j : f(w)$ is unchanged. Since v may have neighbours in blocks of C_j, we must add v to these blocks. There is a connection between v and these neighbours and hence, between all vertices of the component represented by this block. Furthermore, if v is a client or a server then the component containing v has one more client or server, respectively. Thus, we have to add an L- or S-flag to such a block. That is exactly what happens in the inner for-loop. After this we can have nonempty blocks with nonempty intersection, namely v.

We use the REPAIR algorithm to re-establish the structure of the representative. As a final result, we obtain a new representative C_i, which we add to the set of representatives of i. The probability of C_i is:

$$\Pr(C_i) = \Pr_{V_i}(f) = \Pr(C_j) \cdot p(v)$$

Case 2 'v is down': Similar to Case I, we extend scenario f by setting v as down. Hence, no new connections between vertices of blocks or components can be made. Thus, we do not modify C_j to get the new representative C_i. The probability of C_i is:

$$\Pr(C_i) = \Pr(C_j) \cdot (1 - p(v))$$

□

We now analyse the running time of the algorithm. The outer for-loop of the Introduce-Node-Algorithm is executed n_c times, the inner for-loop n_b times. Since we have to check only $O(k)$ neighbours, the condition $B \cap N(v) \neq \emptyset$ can be tested in $O(k^2)$ steps. Hence, the algorithm needs time $O(n_c \cdot n_b^2 \cdot k^2)$.

Forget Nodes

Let node i be a forget node with child j, such that $X_i = X_j \setminus \{v\}$. Note that $V_i = V_j$, thus i and j have the same scenarios. A representative C_i of i differs from the corresponding representative C_j of j only because v does not appear in the blocks of C_i. Thus, we modify every representative C_j of j in the following way to create a representative C_i of node i. We simply delete v from every block of C_j to obtain C_i. As C_i and C_j represent the same scenarios, we have $\Pr(C_i) = \Pr(C_j)$. All this can be done by a rather simple `Forget-Node-Algorithm`. Such an algorithm would need time linear in $n_c \cdot k$ to create all new representatives. (Again, n_c is the maximum number of representatives for a node and k is the treewidth.)

Join Nodes

For a join node i with children j_1 and j_2 we have: $X_i = X_{j_1} = X_{j_2}$. Note that $V_i = V_{j_1} \cup V_{j_2}$, and that $V_{j_1} \cap V_{j_2} = X_i$. Let $up(C)$ denote the set of up vertices of X_i restricted to a representative C of node i.

Definition 93. *Let C be a representative.*

$$up(C) := \{v : v \in B \, for \, B \in C\}$$

Two representatives C^1 and C^2 are compatible *if $up(C^1) = up(C^2)$.*

Each scenario f of node i can be seen as the 'merge' of its restriction f_1 to V_{j_1} and its restriction f_2 to V_{j_2}. Conversely, scenarios f_1 of j_1 and f_2 of j_2 can be merged, if and only if the same vertices in X_i are up, i.e. the corresponding representatives must be compatible.

A representative C_{j_1} of j_1 represents states and connections of vertices of X_{j_1}. The same is true for a representative C_{j_2} of j_2. Note that there is no connection between vertices of a representative, if they belong to two different blocks.

Combining two compatible representatives means checking for new connections, since this translates to combining the corresponding subgraphs. If we have a representative C_{j_1} with two nonempty blocks B_1 and B_2, this means there is no connection in G_{j_1} between vertices of the corresponding components. Note that there is no connection between two blocks of a representative if they are disjoint. However, when combining this representative with a compatible representative C_{j_2}, it is possible that there will be a connection between B_1 and B_2 via a connection of C_{j_2}. Therefore, to combine two representatives, we unite all blocks with nonempty intersection. Specifically this is done by the REPAIR algorithm given in Section 7.1.3.

Before demonstrating that the formula for computing $\Pr(C_i)$ is correct, we introduce the following notations:

$$\Pr_W(f) = \prod_{v \in W^{f=1}} p(v) \cdot \prod_{v \in W^{f=0}} 1 - p(v); \quad \text{for a scenario } f \text{ and } W \subseteq V$$
$$\Pr_{X_i}(C) = \prod_{v \in up(C)} p(v) \cdot \prod_{v \in X_i \setminus up(C)} 1 - p(v); \quad \text{for a set of blocks } C$$

Note that $\Pr_{X_i}(C_i) \neq 0$ and $\Pr_W(f) \neq 0$, because otherwise we would have a vertex v with $p(v) = 0$ being up, or we would have a vertex v with $p(v) = 1$ being down. According to our model, either of this happens with probability 0.

Since the probabilities of the states of vertices of X_i are 'contained' in $\Pr(C_{j_1})$ and in $\Pr(C_{j_2})$ as well, and since the probabilities of the vertices of $V_i \setminus X_i$, being in their appropriate state, are independent, we compute

$$\Pr(C_i) = \frac{\Pr(C_{j_1}) \cdot \Pr(C_{j_2})}{\Pr_{X_i}(C_i)}$$

Again, we give pseudocode to formalise the complete method for join nodes:

```
Join-Node-Algorithm

for every representative $C_{j_1}$ of node $j_1$ do
    for every representative $C_{j_2}$ of node $j_2$ do
        if $up(C_{j_1}) = up(C_{j_2})$
        then $C_i := C_{j_1} \cup C_{j_2}$
             REPAIR($C_i$)
             $\Pr(C_i) = \frac{\Pr(C_{j_1}) \cdot \Pr(C_{j_2})}{\Pr_{X_i}(C_i)}$
             add $C_i$ to the set of representatives of node $i$
        endif
    endfor
endfor
```

Lemma 94. *Given all representatives and their probabilities of the children j_1 and j_2 of a join node i, the* Join-Node-Algorithm *computes the representatives and their probabilities for node i correctly.*

Proof. We consider a representative C_i of node i. C_i represents a scenario f with $C_i = blocks_i(f)$. Such a scenario f determines two scenarios f_1 of G_{j_1} and f_2 of G_{j_2}. These scenarios f_1 and f_2 are represented by representatives C_{j_1} and C_{j_2} of node j_1 or j_2, respectively. C_{j_1} and C_{j_2} are compatible and already computed. Hence, we combine them to get representative C_i. Thus, it is sufficient to combine a representative of j_1 with a compatible one of j_2. By this means, we compute all possible representatives of i.

The search for and processing of compatible representatives is done in the body of the two nested for-loops. Two compatible representatives C_{j_1} and C_{j_2} represent a scenario f of G_i or its restrictions f_1 and f_2 to G_{j_1} and G_{j_2}, respectively. They correspond to components of G_{j_1} and G_{j_2}. If we find one block B_1 in C_{j_1} and another block B_2 in C_{j_2} with $B_1 \cap B_2 \neq \emptyset$, this means that there is a path from a vertex of the component represented by B_1 to a vertex of the component represented by B_2, via a vertex of $B_1 \cap B_2$. In the new representative for i these two components are connected and hence make up one component, with the number of servers and clients added together. Thus, we must join both blocks B_1 and B_2 and modify the number of flags. This is done by using the REPAIR algorithm, described in Section 7.1.3.

We look at representative C_i of node i which is the combination of representative C_{j_1} of j_1 and C_{j_2} of j_2. Since $\Pr_{X_i}(C_i) = \Pr_{X_i}(f)$ with $C_i = \mathit{blocks}_i(f)$, we have:

$$\begin{aligned}
\Pr(C_i) = \Pr_{V_i}(f) &= \Pr_{V_{j_1}}(f_1) \cdot \Pr_{V_{j_2}}(f_2) \cdot \Pr_{X_i}(C_i) \\
&= \frac{1}{\Pr_{X_i}(C_i)} \cdot \Pr_{V_{j_1}}(f_1) \cdot \Pr_{X_i}(C_i) \cdot \Pr_{V_{j_2}}(f_2) \cdot \Pr_{X_i}(C_i) \\
&= \frac{1}{\Pr_{X_i}(C_i)} \cdot \Pr_{V_i}(f_1) \cdot \Pr_{V_i}(f_2) \\
&= \frac{1}{\Pr_{X_i}(C_i)} \cdot \Pr(C_{j_1}) \cdot \Pr(C_{j_2})
\end{aligned}$$

□

The two nested for-loops have $O(n_c^2)$ iterations. At most k^2 steps are required for the if-condition and $O(k^2 \cdot n_b^2)$ for the REPAIR algorithm. Altogether, we have $O(n_c^2 \cdot n_b^2 \cdot k^2)$ steps (n_c is the maximum number of representatives for a node, n_b is the maximum number of blocks of a representative and k is the treewidth).

7.2 The Framework and Equivalence Classes

The network reliability problem $[S \leftrightarrow t]$ (which is sometimes called S-terminal reliability problem) can be expressed as the question for the probability that the surviving subgraph has a certain property. In this example, the property would be the connection between all pairs of vertices of S. The framework given in the previous section can handle more complex properties, and hence can answer the appropriate question for the probability of such more complex properties. In this section, we take a close look at properties that can be handled by the given framework. We also consider equivalence relations between scenarios (i.e. their representatives). With the help of equivalence relations, it is possible to speed up the entire method. Of course, such relations must meet certain requirements for a given problem, which will also be discussed.

7.2.1 Properties of Solvable Network Reliability Problems

The results which can be obtained with our method, can be divided into two groups. The first group does not use an equivalence relation between scenarios. Its consideration is a preparation for the second group which uses equivalence relations between scenarios (see Section 7.2.2).

So far, a representative C of a node i represents only one scenario f of G_i, and $\Pr(C)$ is the probability that this scenario occurs for G_i. $C = blocks_i(f)$ contains information about the connectivity of vertices of S and L. More precisely, the blocks of C reflect the components and the flags the number of vertices of S and L in each component of the graph $G[V_i^{f=1}]$. Since we are dealing with network reliability problems, the term 'property of the graph' should refer to a property concerning the components and connectivity of vertices of S and L. Hence, any question which asks for the probability of such a property can be answered. Clearly, only graphs can have such properties, but we will generalise this to scenarios and representatives.

Definition 95. *A scenario f for node i* has *property Y, if $G[V_i^{f=1}]$ has property Y. A representative C of node i* has *property Y, if the scenario f with $C = blocks_i(f)$ has property Y.*

Now, we know that representatives can have properties. If we want to use them for solving problems, it is also very important to be able to make the decision whether a (representative of a) scenario has a certain property just by considering the information given by the blocks of the representative. If this information is sufficient to determine that a representative has a property, we say that it shows this property.

Definition 96. *A representative C of node i* shows *property Y, if the information given by the blocks of C is sufficient to determine that C has Y.*

The following definition describes all the properties that can be handled or checked with our approach. Therefore, an equivalence between 'showing' and 'having' is necessary.

Definition 97. *A possible property Y of G_i can be* checked *by* using representatives *of i, if for all representatives C of i, it holds that:*

$$C \text{ has property } Y \Longleftrightarrow C \text{ shows property } Y$$

Note that from Definition 96, it already follows that the implication from right to left always holds. After having computed all representatives for node i and their associated probabilities, we can use them to easily solve several problems for G_i.

Lemma 98. *Let Y be a possible property of G_i that can be checked by using the representatives of i. The probability that G_i has property Y equals the sum over the probabilities of the representatives of node i which show the property Y.*

Proof. This can easily be seen, since obviously we have to collect the scenarios with properties Y. Such scenarios are represented by representatives which also have this property Y. Hence, we have to add together the probabilities of representatives showing Y. □

When considering representatives of the root r of the tree-decomposition, we can solve problems for the entire graph G. Because there are $O(2^{|V_i|})$ scenarios for graph G_i, this results in a method with exponential running time, since the number of representatives is a crucial point for the running time of the algorithm. Thus, in the next section, we consider how to reduce the number of representatives. However, the more representatives per node we 'allow', the more information we conserve during the bottom-up process and hence, the more problems we can solve.

7.2.2 Reducing the Number of Representatives

In this section we will have a global look at the strategy. The correctness and restrictions follow in subsequent sections.

By using an equivalence relation between scenarios it is possible to reduce the number of representatives or, in other words: it is possible to reduce the number of objects that have to be considered during the algorithms. For this, we join several scenarios to one equivalence class. We extend this formalism to representatives and say that two representatives (of equivalence classes) are equivalent if the corresponding scenarios (belonging to these classes) are equivalent. Equivalent scenarios, however, must be 'similar' with regard to the way they are processed by the algorithms. Additionally, it is reasonable to consider the equivalence of scenarios of one node at a time. We have to take care that the equivalence relation R we have chosen is suitable to solve our problem. Further, we have to show that R is 'preserved' by the algorithms, i.e. if we have two equivalent scenarios of a node as input to an algorithm, then the resulting scenarios are also equivalent. (This is described in detail in Section 7.2.4.) Then, we use the algorithms given in Section 7.1.4 with the equivalence classes (or better: with their representatives) instead of using the representatives of only one scenario. For that reason, it is only reasonable if we define classes to be equivalent, which are treated in 'the same way' by the algorithms. We have to modify the algorithms slightly, since we have classes now and we have to check after processing each node, if (the scenarios of) two classes became equivalent. If this is the case, we join them into one equivalence class with new probability equal to the sum of the probabilities of the joined representatives (of equivalence classes). The following list of steps summarises what we have to do.

- Step 1: We have to define an equivalence relation R which is fine enough to solve our problem. On the other hand it should be as coarse as possible to reduce the number of classes (i.e. the number of representatives) and hence the running time as much as possible. In Section 7.2.3, we give more details.
- Step 2: We must modify our algorithms to handle (representatives of) classes now and to maintain them, i.e. it is necessary to check for equivalent representatives/classes after processing each node. If we find such two classes among all classes of a node, we keep only one of them with accumulated probability. For doing this, we add as a last step the following code to the algorithms:

```
while there exist representatives $C_a, C_b$ of $i$ with $a \neq b$ and $C_a \equiv C_b$ do
    $\Pr(C_a) := \Pr(C_a) + \Pr(C_b)$
    delete $C_b$ for node $i$
endwhile
```

  This code-fragment can be implemented to take $O(n_c^2)$ iterations. We assume that the check for equivalence of two representatives can be performed in $O(n_b^2)$ time. Thus, we need $O(n_c^2 \cdot n_b^2)$ steps, altogether.
- Step 3: We have to show that it is correct to use the class representatives of R instead of scenario representatives, i.e. we have to show that the algorithms preserve R. This is considered in detail in Section 7.2.4.
- Step 4: We analyse the running time by estimating the maximum number of equivalence classes. The maximum number of blocks is therefore very important.

A representative C is a multiset of blocks. When considering equivalence relations with an equivalence class representative C, we use $[C]$ to denote the equivalence class represented by C, i.e. $[C]$ is a set containing all scenarios represented by C.

7.2.3 Information Content of Equivalence Classes

In Section 7.2.1, we introduced some notations for scenarios. Here, we give analogous notations for equivalence classes. With these, we want to express the information 'contained' in equivalence classes. Furthermore, we clarify what it means for a relation to be fine enough, and we give a lemma that tells us what we can do with relations that are fine enough.

When using an equivalence relation, we define scenarios to be equivalent which may not be equal. Thus, by doing this, we lose the information that these scenarios were different and also why they were different. However, we reduce the number of class representatives, as intended. Clearly, using no equivalence relation provides as much information as possible with this method. However, for solving the S-terminal reliability problem a rather coarse equivalence relation is sufficient. Hence, we are faced with a trade-off between information content and efficiency. In analogy to the Definitions 95, 96 and 97, we give the following definition for equivalence classes.

Definition 99. *An equivalence class $[C]$ of scenarios for node i* has *property Y, if each scenario belonging to $[C]$ has property Y. An equivalence class $[C]$ of node i* shows *property Y, if its representative C shows Y.*

Definition 100. *An equivalence relation R is fine enough for property Y, if for all nodes i of the tree-decomposition the following holds: For all equivalence classes $[C]$ of node i, it holds that:*

$$(\forall f \in [C] : f \text{ has property } Y) \quad \vee \quad (\forall f \in [C] : f \text{ does not have property } Y)$$

That means that R is fine enough for Y, iff there is no equivalence class which contains a scenario that has property Y and another scenario which does not have property Y. In the same flavour as Lemma 98, the next one tells us that under certain conditions, we can use equivalence classes to solve problems for a graph G, after computing all equivalence classes of all nodes until we reached the root r.

Lemma 101. *Let R be an equivalence relation which is fine enough for property Y that can be checked using representatives. The probability that G_i has property Y equals the sum over the probabilities of the equivalence classes of node i which show this property Y.*

Proof. This can be proven very similar to the proof of Lemma 98. Again, we have to add together the probabilities of the scenarios with property Y. These scenarios are collected in equivalence classes which also have and show this property Y, because Y can be checked using representatives. Since R is fine enough, it is sufficient to sum up the probabilities of these equivalence classes. □

7.2.4 Conditions for Using Equivalence Classes

In earlier sections, we discussed when an equivalence relation is fine enough. In this section we will see under which conditions we can use (the representatives of) the equivalence classes during the execution of the bottom-up computation in the tree-decomposition by means of the leaf-, introduce-, forget-, and join-node algorithms.

The choice of the equivalence relation R cannot be made arbitrarily. As mentioned in Step 1, R has to be fine enough to provide enough information to solve the problem. Furthermore, R must have some structural properties, i.e. R must be 'respected' or 'preserved' by the algorithms. This targets to the property that if two scenarios are equivalent before being processed by one of the algorithms for the four types of nodes, they are also equivalent after this processing. To capture exactly the notion of 'preserving an equivalence relation', some definitions are required.

Leaf nodes need no consideration, since we have no class representatives as input to the algorithm for leaf nodes. For forget nodes, the situation is rather simple.

Definition 102. *Let i be a forget node with child j. Let C_j^x be a representative of j which results, after processed by the* `Forget-Node-Algorithm`, *in C_i^x of node i. The* `Forget-Node-Algorithm` preserves *the equivalence relation R if the following holds:*

$$\forall C_j^1, C_j^2 : (C_j^1, C_j^2) \in R \Longrightarrow (C_i^1, C_i^2) \in R$$

Now, we look at introduce nodes. The introduce-node-algorithm produces two output-representatives for each input-representative, namely one for the introduced vertex v being up and one for v being down.

Definition 103. *Let i be an introduce node with child j with $X_i = X_j \cup \{v\}$. Let representative C_j^x of node j be the input for the* `Introduce-Node-Algorithm`, *which gives as output $C_i^{x,up}$ and $C_i^{x,down}$ of node i, for v is up and down, respectively. The* `Introduce-Node-Algorithm` preserves *the equivalence relation R if the following holds:*

$$\forall C_j^1, C_j^2 : (C_j^1, C_j^2) \in R \Longrightarrow (C_i^{1,up}, C_i^{2,up}) \in R \wedge (C_i^{1,down}, C_i^{2,down}) \in R$$

For join nodes, we have to consider representatives of two children.

Definition 104. *Let i be a join node with children j_1 and j_2. Furthermore, let $C_{j_1}^x$ and $C_{j_2}^y$ be representatives of node j_1 and j_2, respectively. Let $C_i^{x,y}$ be the result of the* `Join-Node-Algorithm` *for processing $C_{j_1}^x$ and $C_{j_2}^y$. The* `Join-Node-Algorithm` preserves *the equivalence relation R if the following holds:*

$$\forall C_{j_1}^1, C_{j_1}^2, C_{j_2}^1, C_{j_2}^2 :$$
$$(C_{j_1}^1, C_{j_1}^2) \in R \wedge (C_{j_2}^1, C_{j_2}^2) \in R \wedge up(C_{j_1}^1) = up(C_{j_1}^2) = up(C_{j_2}^1) = up(C_{j_2}^2)$$
$$\Longrightarrow$$
$$(C_i^{1,1}, C_i^{1,2}) \in R \wedge (C_i^{1,2}, C_i^{2,1}) \in R \wedge (C_i^{2,1}, C_i^{2,2}) \in R$$

If we use only (the representatives of) the equivalence classes in the algorithms, we can process all scenarios represented by such equivalence classes by only one execution of an algorithm. The results will be, again, equivalence classes. Using an equivalence relation which is preserved by the algorithms is therefore important. That means that we do not have to worry whether an algorithm 'violates' R by creating a result that actually is not an equivalence class of R.

Lemma 105. *When using equivalence classes of R as input for the* `Forget-`, `Introduce-` *or* `Join-Node-Algorithm`, *then the result will also be equivalence classes of R, if R is preserved by the respective algorithm.*

Proof. It follows directly from the corresponding definitions above that all scenarios represented by a resulting 'equivalence class' are indeed equivalent. The algorithm-fragment given in Step 2 in Section 7.2.2 ensures that all equivalent scenarios are pooled together in one equivalence class. □

From the previous definition, we can see that it is important that equivalent scenarios are compatible. Actually, this is not sufficient for being able to apply our framework. The idea is to handle equivalent scenarios with only one computation. The algorithms modify only the nonempty blocks and two equivalent scenarios have to be modified in the same way. Hence, for applying the framework equivalent scenarios must have the same set of nonempty blocks. However, there are reasonable exceptions to this, which will not be considered here in general, to keep the framework and its presentation less technical. The next definition summarises the two conditions discussed above.

Definition 106. *An equivalence relation R is called* proper *if*

- *for all equivalence classes C of R, it holds that: for all scenarios $f, f' \in C$, the sets of nonempty blocks of f and f', respectively, are equal and*
- *R is preserved by the* `Forget-`, `Introduce-` *and* `Join-node-algorithms`.

7.2.5 The Correctness of the Algorithm for Classes

In this section, we look at the correctness of the counterparts of the algorithms given in Section 7.1.4. As described earlier, only a few modifications are needed to get algorithms for (representatives of) equivalence classes rather than (representatives of) scenarios. We can take the algorithms described in Section 7.1.4 and extend them with the while loop given in Step 2 in Section 7.2.2. While the algorithms are straightforward, the proof of their correctness is slightly more complicated.

We use the same terminology as in Section 7.1.4: i denotes a node of the tree T. Node i may have $0, 1$ or 2 children, depending on the type of i. These children are referred to as j or j_1, j_2. C_i denotes a representative of a class of i, which is computed using the representative(s) of class(es) C_j (C_{j_1}, C_{j_2}) of node(s) j (j_1, j_2), respectively. We will have a look at each type for node i.

When handling representatives, a lower index refers to the node the representative belongs to, and an upper index is used to differentiate different representatives belonging to the node considered. In general, the algorithms given in Section 7.1.4 can be used here as well. However, they have to be extended by the code-fragment of Step 2 in Section 7.2.2 to check for new equivalences. It is easy to see, that each of these 'new' algorithms can be performed in time $O(n_c^2 \cdot n_b^2 \cdot k^2)$.

Leaf Nodes

As seen before, for leaves i of T, the algorithm is rather simple, since $|X_i| = 1$. It is very similar to the description for leaves given in Section 7.1.4, because we have two scenarios for a leaf. For proper relations, these scenarios will never be in one equivalence class at a leaf, since their sets of nonempty blocks are not equal. Hence, we have two equivalence classes for a leaf. The remaining argumentation is very similar to the part in Section 7.1.4 concerning leaf nodes.

Introduce Nodes

Suppose i is an introduce node with child j, such that $X_i = X_j \cup \{v\}$. We use the representatives of the classes and their probabilities of node j to compute the representatives and their probabilities of node i. For any state of v (up or down, in L or not in L, in S or not in S) we compute the representatives and add them to the set of representatives of node i. This is done in an analogous way as described in Section 7.1.4.

Lemma 107. *Given the representatives of all equivalence classes of a proper equivalence relation and their probabilities of the child j of an introduce node i, the* `Introduce-Node-Algorithm` *computes (the representatives of) all equivalence classes and their probabilities for node i correctly.*

Proof. We can compute the representatives and their probabilities for node i by considering every possible scenario for node i and creating equivalence classes according to the equivalence relation.

However, recall that v is the only vertex in $V_i \setminus V_j$. So, each scenario for node i can be obtained by taking a scenario for node j and extending it by specifying whether v is up or down. All possible scenarios for node j are represented by the representatives of equivalence classes of j, whose probabilities are known. Since the considered equivalence relation is proper, two scenarios which are equivalent at node j will result in equivalent scenarios of node i (for v being fixed to up or down). For that reason, it is sufficient to combine the representatives of classes of j with v to create the representatives of classes of i. To do this we consider both v being up and down for each class representative C_j of j and make the appropriate modifications. By doing this, we implicitly look at every scenario which is possible for G_i. For the correctness, we consider for a class representative C_j of j two cases: v is up or v is down:

Case 1 'v is up': For all $f \in [C_j]$, we extend f by $f(v) = 1$, i.e. for all $w \in V_j : f(w)$ is unchanged. The argumentation is very similar to the proof of Lemma 92. We have to use only one more argument. Since the relation is proper, all scenarios $f \in [C_j]$ have the same set of nonempty blocks. And hence, all these scenarios would be handled in the same way when treated individually. That is why we can handle them by a single pass of the outer loop of the algorithm when using the representative C_j. The probability of the new class with representative C_i is:

$$\Pr(C_i) = \sum_{f \in [C_i]} \Pr(f) = \sum_{f \in [C_j]} \Pr(f) \cdot p(v) = \Pr(C_j) \cdot p(v)$$

Here, $[C_i]$ contains scenarios, which are scenarios belonging to $[C_j]$ extended by $f(v) = 1$.

Case 2 'v is down': Similar to Case I, we extend all scenarios f by 'v is down'. Hence, no new connections between vertices of blocks or components can be made. Thus, we do not modify C_j to get the new class representative C_i. The probability of C_i is:

$$\Pr(C_i) = \Pr(C_j) \cdot (1 - p(v))$$

□

Forget Nodes

For a forget node i with child j, let $X_i = X_j \setminus \{v\}$. Note that $V_i = V_j$, thus i and j have the same scenarios. However, they might have different representatives. Let C_j and C_i be two representatives of the same scenario of node j and i, respectively. Because v does not appear in the blocks of C_i, C_j and C_i are different or equal, depending on the state of vertex v (up or down). Each scenario belongs to an equivalence class. Thus, we modify every class representative C_j of j in the following way to create a class representative C_i of node i. We simply delete v from every block of C_j containing v to obtain C_i. For a proper equivalence relation, all scenarios belonging to an equivalence class $[C_j]$ have the same set of nonempty blocks. Hence, by deleting v from the blocks of C_j, we implicitly delete v from all scenarios represented by C_j. This results in the new equivalence class C_i for node i. Because C_i and C_j represent the same scenarios, we have $\Pr(C_i) = \Pr(C_j)$.

Join Nodes

For join nodes i with children j_1 and j_2 we have: $X_i = X_{j_1} = X_{j_2}$. Note that $V_i = V_{j_1} \cup V_{j_2}$ and $V_{j_1} \cap V_{j_2} = X_i$.

We proceed as described in Section 7.1.4. We combine compatible class representatives of nodes j_1 and j_2. After combining, we repair them with the procedure given in Section 7.1.3. By doing this, we get equivalence classes of node i.

Lemma 108. *Given all the representatives of equivalence classes of the children j_1 and j_2 of a join node i and the probabilities of these equivalence classes with a proper underlying equivalence relation, the* `Join-Node-Algorithm` *computes the representatives and the probabilities of classes of node i correctly.*

Proof. We consider an equivalence class representative C_i of node i. By restricting a scenario $f \in [C_i]$ to V_{j_1} and V_{j_2}, respectively, it determines two scenarios f_1 of G_{j_1} and f_2 of G_{j_2}. Those two scenarios f_1 and f_2 belong to equivalence classes C_{j_1} and C_{j_2} of nodes j_1 and j_2, respectively. The representatives of these classes and their probabilities are already computed in our bottom-up approach. Note that these representatives are compatible. Furthermore, we have due to equivalence (and hence compatibility) that any scenario in $[C_{j_1}]$ combined with any scenario in $[C_{j_2}]$ results in a scenario in $[C_i]$. This yields the Cartesian product of $[C_{j_1}]$ and $[C_{j_2}]$ with $|[C_{j_1}]| \cdot |[C_{j_2}]|$ combinations. Fortunately, since any such combination results in a scenario in $[C_i]$, we can compute all combinations by a single pass of the algorithm by using the representatives of $[C_{j_1}]$ and $[C_{j_2}]$. Hence, simply combining the representatives C_{j_1} and C_{j_2} is sufficient to handle all scenarios represented by them and to create (a subset of) $[C_i]$.

To create the entire set of scenarios $[C_i]$, we have to process all equivalence classes emerging from restrictions of scenarios of $[C_i]$ to V_{j_1} and V_{j_2}. However, the same argumentation as above holds for these as well, and the added code-fragment of Step 2 in Section 7.2.2 will check for equivalent classes and produce the final class $[C_i]$.

Let C_{j_1} and C_{j_2} be two compatible equivalence class representatives of node j_1 and j_2, respectively. $[C_{j_1}]$ and $[C_{j_2}]$ are combined to get class $[C_i]$ of node i. Then we have for the probability of C_i of node i:

$$\Pr(C_i) = \sum_{f \in [C_i]} \Pr_{V_i}(f)$$

This product can be split into factors with regard to the children of node i. However, the set X_i is contained in both sets V_{j_1} and V_{j_2}. Hence, this product would contain a factor for the state probabilities of vertices of X_i twice, which we correct by an appropriate division:

$$\Pr(C_i) = \sum_{f \in [C_i]} \Pr_{V_{j_1}}(f) \cdot \Pr_{V_{j_2}}(f) \cdot \frac{1}{\Pr_{X_i}(f)}$$

Since the latter factor is constant for all $f \in [C_i]$ for a proper equivalence relation, we have:

$$\Pr(C_i) = \frac{1}{\Pr_{X_i}(f)} \cdot \sum_{f \in [C_i]} \Pr_{V_{j_1}}(f) \cdot \Pr_{V_{j_2}}(f)$$

Because the equivalence class $[C_i]$ contains all scenarios, which result from the Cartesian product of compatible classes $[C_{j_1}]$ and $[C_{j_2}]$ of j_1 and j_2, respectively, we have:

$$\sum_{f\in[C_i]} \Pr_{V_{j_1}}(f)\cdot \Pr_{V_{j_2}}(f) = \sum_{f\in[C_i]} \Pr_{V_{j_1}}(f)\cdot \sum_{f\in[C_i]} \Pr_{V_{j_2}}(f) = \sum_{f\in[C_{j_1}]} \Pr_{V_{j_1}}(f)\cdot \sum_{f\in[C_{j_2}]} \Pr_{V_{j_2}}(f)$$

And hence, this results in:

$$\begin{aligned}\Pr(C_i) &= \frac{1}{\Pr_{X_i}(f)}\cdot \sum_{f\in[C_{j_1}]} \Pr_{V_{j_1}}(f)\cdot \sum_{f\in[C_{j_2}]} \Pr_{V_{j_2}}(f)\\ &= \frac{1}{\Pr_{X_i}(f)}\cdot \Pr(C_{j_1})\cdot \Pr(C_{j_2})\end{aligned}$$

After computing the new classes of node i, which contain scenarios corresponding to the Cartesian product of scenarios of compatible classes of j_1 and j_2, we have to check for new equivalences. The correctness of the procedure we use for this is easy to see. □

We are now ready to bring everything together.

Theorem 109. *Let R be a proper equivalence relation of scenarios that is fine enough for property Y which can be checked using representatives. Then we can use our framework to compute the probability that G_i has property Y.*

Proof. The correctness of the computation of the representatives and their probabilities follows from the correctness of the particular algorithms for the four types of nodes. Lemma 107 and 108 show this for introduce- and join-nodes. The situation is trivial for leaf- and forget-nodes. The theorem follows now from Lemma 101 and 105. □

The running time depends on the relation R. For a fixed size of X_i, if R has a finite number of equivalence classes, then the algorithm is linear time. If the number of equivalence classes is bounded by a polynomial in the number of vertices of G, then the algorithm uses polynomial time. For further discussion, see Section 7.5.

7.3 Problems that Fit into this Framework

In this section, we look at examples of solvable problems which yield different running times. Later in this section, we look at relations (and the resulting running times) that meet all conditions described in the previous section, because they are generated by specific operations or because they are a combination of relations.

7.3.1 Solvable Problems

It is not possible to give an exhaustive list of problems that can be handled with this approach, because many equivalence relations can be used, and very often, many different questions can be handled with a single relation. Note that we assume in this entire chapter that a tree-decomposition of bounded width of the graph is given. Theorem 109 gives information about solvable problems. We simply can say that every problem that asks for the probability of a property Y of G can be solved, if Y can be checked using classes. Therefore we should find an equivalence relation R as coarse as possible, but still fine enough for Y. Furthermore, to apply the framework described in this paper, R must be preserved by the algorithms. In the following, we list some example properties.

To obtain relations that allow algorithms with linear running time, we can restrict the maximum number of blocks and the number of flags per block to be constant. With such relations we can answer questions like: 'What is the probability that all clients are connected to at least one server?' (problem $[L \leftrightarrow \geq 1S]$) or 'What is the probability that all servers are useful, i.e. have a client connected to them?' ($[S \leftrightarrow \geq 1L]$). We can also use only one kind of special vertices, e.g. only servers. With such a relation we are able to give an answer to 'What is the probability that all servers are connected?' ($[S \leftrightarrow t]$), which is the classical S-terminal reliability problem. With additional ideas and modifications, it is also possible to answer the following question in linear time: 'What is the expected number of components that contain at least one vertex of S (of L; of S and L)?' ($[Ec \geq 1S]$, $[Ec \geq 1L]$ or $[Ec \geq 1(S \wedge L)]$, respectively)

A possible assumption would be to consider the number of servers n_S to be small. We can use a different S-flag for each server. In this case, a relation which does not conflate or fuse empty blocks with S-flags, but only with L-flags, can be utilised. Further, each block can have multiple flags of each type. Unfortunately, this relation leads to a maximum number of classes bounded exponentially in n_S. With this relation we can answer the question with which probability certain servers are connected, and with what probability server x has at least y clients connected to it (x, y are integers). We can also determine the 'most useless' server, i.e. the server with smallest expected number of clients connected to it. Of course, this relation enables us to compute the expected number of components with at least one server.

If we use relations that bound n_b by a constant, and the number of flags per block is only bounded by n_L and n_S, then our algorithm will run in a time polynomial in n_L and n_S. Such relations enable properties like: at least x clients are not connected to a server, or at most y server are not connected to a client, as well as at least x clients are connected to at least one server while at least y servers are connected to at least one client.

7.3.2 Generating Relations

The relation which is the finest we can have, is the relation $\hat{R}$ which assigns to each scenario its own equivalence class. This relation is clearly preserved by the algorithms. We show below, that if we modify $\hat{R}$ by applying a sequence of specific operations to get a relation R, then R is preserved by the algorithms as well. The goal is to 'construct' relations which are 'automatically' preserved.

As a reminder, a representative is a multiset of blocks, not necessarily representing a scenario. With an operation $h : \mathcal{C} \rightarrow \mathcal{C}$ (where $\mathcal{C}$ is the set of multisets of blocks), which takes the representation, i.e. the set of blocks of an equivalence class as input, we can define a new equivalence relation R_h. We give a list of a few basic operations, with which many useful relations can be created. Therefore, we can start with the finest relation $\hat{R}$ which equivalence classes represent exactly one scenario. The general form of an operation is: $h^{block-sel}_{action}$, whereby *block-sel* selects the blocks for the *action*, which is one of the following:

- 'del': We *del*ete all blocks that are selected by *block-sel*.
- '$dmulS$': Here, we *d*rop *mul*tiple S-flags and keep only one. That means each block selected by *block-sel* is replaced by the same block at which we delete multiple S-flags and we keep only one of them.
- '$dmulL$': The same as $dmulS$, but here we *d*rop *mul*tiple L-flags.
- 'con': We *con*flate all selected blocks and *con*catenate their lists of flags. That means all blocks selected by *block-sel* are united to one block with as many S- and L- flags as those of the united blocks together.

As already mentioned, blocks are selected by *block-sel*, which is a sequence of the following symbols:

- '$=\emptyset$' or '$\neq\emptyset$', which determines that either empty blocks or nonempty blocks are selected, respectively.
- 'S' or '$\not{S}$', which determines that either blocks that must have at least one S-flag or blocks that must not have any S-flag are selected.
- 'L' or '$\not{L}$', which is the same as above for L-flags.

We give the set of those block-selectors that are considered in this section.

$$BSel = \{\neq\emptyset, \neq\emptyset\not{S}\not{L}, \neq\emptyset S\not{L}, \neq\emptyset\not{S}L, \neq\emptyset SL, =\emptyset, =\emptyset\not{S}\not{L}, =\emptyset S\not{L}, =\emptyset\not{S}L, =\emptyset SL\}$$

For the sake of clarity, we will look at some examples:

- $h_{del}^{=\emptyset\not{S}\not{L}}(C)=$ the multiset of blocks consisting of exactly all blocks of C but without empty blocks without any flag:
$$\begin{array}{rl} h_{del}^{=\emptyset\not{S}\not{L}} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}\ \} \end{array}$$
- $h_{dmulS}^{=\emptyset SL}(C)=$ the multiset of blocks consisting of exactly all blocks of C but for each empty block with at least one S-flag and at least one L-flag, we drop (or delete) all but one S-flag:
$$\begin{array}{rl} h_{dmulS}^{=\emptyset SL} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{S}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ \} \end{array}$$
- $h_{dmulL}^{\neq\emptyset\not{S}L}(C)=$ the multiset of blocks consisting of exactly all blocks of C but each nonempty block $\{...\}_{L...L}$ with at least one L-flag and no S-flag is replaced by the same block $\{...\}_{L}$ with only one L-flag:
$$\begin{array}{rl} h_{dmulL}^{\neq\emptyset\not{S}L} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{L}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ \} \end{array}$$
- $h_{con}^{=\emptyset S\not{L}}(C)=$ the multiset of blocks consisting of exactly all blocks of C but all empty blocks with at least one S-flag and no L-flag are replaced by one empty block with as many S-flags as those of the replaced blocks together:
$$\begin{array}{rl} h_{con}^{=\emptyset S\not{L}} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SSS}, \{\ \}\ \} \end{array}$$

Compiling all operations which we will consider, we get the set $\mathcal{H}$:

$$\mathcal{H} = \{h_{del}^{=\emptyset\not{S}\not{L}}, h_{del}^{=\emptyset S\not{L}}, h_{del}^{=\emptyset\not{S}L}, h_{del}^{=\emptyset SL}, h_{dmulS}^{=\emptyset S\not{L}}, h_{dmulS}^{=\emptyset SL}, h_{dmulS}^{\neq\emptyset S\not{L}}, h_{dmulS}^{\neq\emptyset SL},$$
$$h_{dmulL}^{=\emptyset\not{S}L}, h_{dmulL}^{=\emptyset SL}, h_{dmulL}^{\neq\emptyset\not{S}L}, h_{dmulL}^{\neq\emptyset SL}, h_{con}^{=\emptyset S\not{L}}, h_{con}^{=\emptyset\not{S}L}, h_{con}^{=\emptyset SL}\}$$

Note that in this list the only operations mapping nonempty sets are operations that drop multiple flags, i.e. that do not change any vertex of any block.

So far, h is an operation which takes a multiset of blocks as input and gives a multiset of blocks as output. In the following definition, we extend the applicability of h to an equivalence relation R. Note that the relation $\hat{R}$ is the finest relation possible. It assigns to each scenario its own equivalence class.

Definition 110. *Let $\mathcal{R}_0 = \{\hat{R}\}$ and $h \in \mathcal{H}$. For $R \in \mathcal{R}_i$ we define a new equivalence relation $h(R)$ in the following way: Let C^1 and C^2 be two representatives of two equivalence classes $[C^1]$ and $[C^2]$ of R, respectively. Then*

$$(C^1, C^2) \in h(R) \iff h(C^1) = h(C^2)$$

Furthermore, we have:

$$\mathcal{R}_{i+1} = \{R' | R' = h(R) \text{ for } R \in \mathcal{R}_i \text{ and } h \in \mathcal{H}\}$$

and

$$\mathcal{R} = \bigcup_{i=0}^{\infty} \mathcal{R}_i$$

If $(C^1, C^2) \in h(R)$, it might be convenient to choose the representative C of the new equivalence class $[C]$ with $C^1, C^2 \in [C]$ as follows: $C = h(C^1) = h(C^2)$. However, the algorithms have to be modified to maintain this. As an example, if the `Forget-Node-Algorithm` creates a new empty block without any flags and if we use the operation $h_{del}^{=\emptyset\not S\not L}$ to define the new relation, then we have to modify the `Forget-Node-Algorithm` to delete the new empty block.

Thus, applying an operation to the multiset of blocks of the representation of an equivalence class immediately results in the multiset of blocks of an equivalence class of the new relation. Now, the idea is to consider operations $h \in \mathcal{H}$ and prove that $h(R)$ is preserved by the algorithms if $h(R)$ is obtained by applying a sequence of operations of $\mathcal{H}$ to $\hat{R}$, i.e. we will show that each $R \in \mathcal{R}$ is preserved by the algorithms. To prove this, we require some lemmas.

In the upcoming proofs, we use the following terms: For A a multiset of blocks, $blocks^{\alpha}(A)$ is the multiset of all blocks of A selected by $\alpha \in BSel$. We call these sets *categories*. Here are a few examples:

- $blocks^{\neq\emptyset}(A)$ is the set of all nonempty blocks of A.
- $blocks^{\neq\emptyset\not S L}(A)$ is the set of all nonempty blocks of A with at least one L-flag and no S-flags.
- $blocks^{=\emptyset\not S\not L}(A)$ is the multiset of all empty blocks of A with no flags.

Lemma 111. *Let h be a sequence of operations of $\mathcal{H}$, A be a multiset of blocks, and $\alpha \in BSel$.*

$$blocks^{\alpha}(h(A)) = h(blocks^{\alpha}(A))$$

Proof. This is easy to see for a single operation $h \in \mathcal{H}$. By applying this iteratively for each operation in a sequence h, we obtain the lemma. □

As a reminder, the symbol '∪' between multisets denotes the multiunion. (See the paragraph about blocks in Section 7.1.1 for more details.)

Lemma 112. *Let A be a multiset of blocks and let h be a sequence of operations of $\mathcal{H}$.*

$$\begin{aligned} h(A) = {} & h(blocks^{\neq\emptyset\not S\not L}(A)) \cup h(blocks^{=\emptyset\not S\not L}(A)) \\ & \cup h(blocks^{\neq\emptyset S\not L}(A)) \cup h(blocks^{=\emptyset S\not L}(A)) \\ & \cup h(blocks^{\neq\emptyset\not S L}(A)) \cup h(blocks^{=\emptyset\not S L}(A)) \\ & \cup h(blocks^{\neq\emptyset SL}(A)) \cup h(blocks^{=\emptyset SL}(A)) \end{aligned}$$

Proof. The multiset A of blocks can be partitioned into:

$$\begin{aligned} A = {} & \mathit{blocks}^{\neq\emptyset\not{S}\not{L}}(A) \cup \mathit{blocks}^{=\emptyset\not{S}\not{L}}(A) \\ & \cup \mathit{blocks}^{\neq\emptyset S\not{L}}(A) \cup \mathit{blocks}^{=\emptyset S\not{L}}(A) \\ & \cup \mathit{blocks}^{\neq\emptyset\not{S}L}(A) \cup \mathit{blocks}^{=\emptyset\not{S}L}(A) \\ & \cup \mathit{blocks}^{\neq\emptyset SL}(A) \cup \mathit{blocks}^{=\emptyset SL}(A) \end{aligned}$$

It follows from the definition of the categories $\mathit{blocks}^{\alpha}(A)$ above (for $\alpha \in \mathit{BSel} \setminus \{\neq\emptyset, =\emptyset\}$) that this is indeed a partition. Each single operation handles only the blocks of one category and leaves the other categories untouched. Therefore, it is easy to see that the claim is true for h being a single operation. Now, the result follows from Lemma 111. □

Lemma 113. *Let A and B be multisets of blocks and let h be a sequence of operations of $\mathcal{H}$.*

$$h(A) = h(B) \iff$$

$$\forall\alpha \in \mathit{BSel} : h(\mathit{blocks}^{\alpha}(A)) = h(\mathit{blocks}^{\alpha}(B))$$

Proof. The '$\Leftarrow$' direction follows from Lemma 112. To see the '$\Rightarrow$' direction, let $D \in h(A)$. Clearly, $D \in h(B)$ and there is an $\alpha \in \mathit{BSel} \setminus \{\neq\emptyset, =\emptyset\}$, with $D \in \mathit{blocks}^{\alpha}(h(A))$. Since $h(A) = h(B)$, we have $D \in \mathit{blocks}^{\alpha}(h(B))$. Because this is true for all $D \in \mathit{blocks}^{\alpha}(h(A))$, we have: $\mathit{blocks}^{\alpha}(h(A)) \subseteq \mathit{blocks}^{\alpha}(h(B))$, and by symmetry: $\mathit{blocks}^{\alpha}(h(A)) = \mathit{blocks}^{\alpha}(h(B))$. By Lemma 111, we conclude: $h(\mathit{blocks}^{\alpha}(A)) = h(\mathit{blocks}^{\alpha}(B))$. Since D was chosen arbitrarily, α was chosen arbitrarily as well, and we see that the last equation is true for all $\alpha \in \mathit{BSel}$. □

Lemma 114. *Let A and B be multisets of blocks. Furthermore, let $h \in \mathcal{H} \setminus \{h_{con}^{=\emptyset S\not{L}}, h_{con}^{=\emptyset\not{S}L}, h_{con}^{=\emptyset SL}\}$. Then it holds that:*

$$h(A \cup B) = h(A) \cup h(B)$$

Proof. This is easy to see, because such an h modifies a multiset of blocks locally, namely block by block, which means that each block is exchanged by another one of the same category or deleted. □

It is not possible to give an easy-to-prove statement as in Lemma 114 if at least one of $h_{con}^{=\emptyset S\not{L}}$, $h_{con}^{=\emptyset\not{S}L}$ or $h_{con}^{=\emptyset SL}$ is involved. In this case, it is possible that several blocks are exchanged by one other block. We will have a closer look at a sequence containing $h_{con}^{=\emptyset S\not{L}}$. Operation $h_{con}^{=\emptyset S\not{L}}$ only affects empty blocks with at least one S-flag and no L-flags. As we have seen above, there is no interference between different categories. Thus, in the next lemma, we restrict ourself to this category and only to operations which have an effect to this category.

Lemma 115. *Let A, B, C, D be multisets of empty blocks with at least one S-flag and no L-flags. For h a sequence of operations of $\{h_{del}^{=\emptyset S\not{L}}, h_{dmulS}^{=\emptyset S\not{L}}, h_{con}^{=\emptyset S\not{L}}\}$, we have:*

$$h(A) = h(B) \ \wedge\ h(C) = h(D) \implies h(A \cup C) = h(B \cup D)$$

Proof. We consider a number of cases:

Case 1 '$h_{con}^{=\emptyset S\not{L}}$ is not in the sequence h': Then the result follows directly from Lemma 114.

Case 2 'there is any $h_{del}^{=\emptyset S\not{L}}$ in the sequence h': Then we have the situation that there are no empty blocks with at least one S-flag and no L-flags anymore, and the result trivially holds.

Hence, we suppose in the following cases that $h_{con}^{=\emptyset S \not L} \in h$ and $h_{del}^{=\emptyset S \not L} \notin h$.

Case 3 '$h(X) = ...h_{dmulS}^{=\emptyset S \not L}(...h_{con}^{=\emptyset S \not L}(...(X)...)...)...$*, i.e. first we apply* $h_{con}^{=\emptyset S \not L}$ *and then* $h_{dmulS}^{=\emptyset S \not L}$*'*: Sequence h will turn all empty blocks with at least one S-flag and no L-flags into a single empty block with a single S-flag. Then we know that $h(A) = h(B) \in \{\, \{\,\}^S, \emptyset \,\}$ and $h(C) = h(D) \in \{\, \{\,\}^S, \emptyset \,\}$. (We have $h(A) = h(B) = \emptyset$ iff A and B do not contain any empty block with at least one S-flag and no L-flags. The same also holds for sets C and D.) If $A \cup C$ contains an empty block with only S-flags, then $B \cup D$ as well, and we have: $h(A \cup C) = h(B \cup D) = \{\{\,\}^S\}$ otherwise we have: $h(A \cup C) = h(B \cup D) = \emptyset$.

Case 4 '$h(X) = ...h_{con}^{=\emptyset S \not L}(...h_{dmulS}^{=\emptyset S \not L}(...(X)...)...)...$*, i.e. first we apply* $h_{dmulS}^{=\emptyset S \not L}$ *and then* $h_{con}^{=\emptyset S \not L}$*'*: The result of $h(X)$ in this case is $\{\{\,\}^{S...S}\}$, where the number of S-flags equals the number of empty blocks in X with at least one S-flag and no L-flags. Let a be the number of empty blocks of A with at least one S-flag and no L-flags. We define b, c, d in the same way. Then we trivially have: $a = b \wedge c = d \Longrightarrow a + c = b + d$, which implies $h(A \cup C) = h(B \cup D)$.

Case 5 'only $h_{con}^{=\emptyset S \not L}$ *and no* $h_{dmulS}^{=\emptyset S \not L}$ *is applied'*: Let a be the total number of all S-flags of all empty blocks of A with at least one S-flag and no L-flags. We define b, c, d in the same way. Since $h(A)$ is exactly one empty block with a S-flags and no L-flags, we have: $a = b \wedge c = d \Longrightarrow a + c = b + d$. Hence $h(A \cup C) = h(B \cup D)$.

Case 6 '$h(X) = ...h_{con}^{=\emptyset S \not L}(...h_{dmulS}^{=\emptyset S \not L}(...h_{con}^{=\emptyset S \not L}(...(X)...)...)...)...$*, i.e. first we apply* $h_{con}^{=\emptyset S \not L}$*, then* $h_{dmulS}^{=\emptyset S \not L}$ *and then* $h_{con}^{=\emptyset S \not L}$ *again'*: Here the outermost $h_{con}^{=\emptyset S \not L}$ has no effect and hence we have already considered this case above.

Case 7 '$h(X) = ...h_{dmulS}^{=\emptyset S \not L}(...h_{con}^{=\emptyset S \not L}(...h_{dmulS}^{=\emptyset S \not L}(...(X)...)...)...)...$*, i.e. first we apply* $h_{dmulS}^{=\emptyset S \not L}$*, then* $h_{con}^{=\emptyset S \not L}$ *and then* $h_{dmulS}^{=\emptyset S \not L}$ *again'*: In this case the innermost $h_{dmulS}^{=\emptyset S \not L}$ has no effect and thus is already handled above. □

We can easily see that lemmas similar to the previous one are true, when considering blocks with at least one L-flag and no S-flags (or for blocks with at least one S-flag and at least one L-flag). The proofs are very similar and hence we omit them. We generalise the previous lemma which is the last intermediate step.

Lemma 116. *Let A, B, C, D be multisets of blocks. For h a sequence of operations of $\mathcal{H}$, we have:*

$$h(A) = h(B) \;\wedge\; h(C) = h(D) \;\Longrightarrow\; h(A \cup C) = h(B \cup D)$$

Proof. As discussed above, operations for different categories of blocks do not affect each other, and hence we can consider each category separately. Thus, the result follows easily from Lemma 112, 113, 115 and the variants of Lemma 115, discussed above. □

Lemma 117. *Let R be an equivalence relation with $R \in \mathcal{R}$. R is preserved by the* `Introduce-`*,* `Forget-` *and* `Join-Node-Algorithms`.

Proof. Let h be the sequence of operations with $h(\hat{R}) = R$. In many equations below, we make use of Lemmas 115 and 116. We consider each algorithm separately:

- The `Forget-Node-Algorithm`:
  Node i is a forget node with child j. Let C_j^1, C_j^2 be representatives of node j and let C_i^1, C_i^2 be representatives of node i which are the results of the `Forget-Node-Algorithm` of representatives C_j^1, C_j^2, respectively. We have:

$$\begin{aligned} h(C_j^1) = h(C_j^2) \Longrightarrow\ & \mathit{blocks}^{\neq\emptyset}(C_j^1) = \mathit{blocks}^{\neq\emptyset}(C_j^2)\ \wedge \\ & \mathit{blocks}^{\neq\emptyset}(C_i^1) = \mathit{blocks}^{\neq\emptyset}(C_i^2)\ \wedge \\ & h(\mathit{blocks}^{=\emptyset}(C_j^1)) = h(\mathit{blocks}^{=\emptyset}(C_j^2)) \end{aligned} \tag{7.1}$$

  Now, we can distinguish two cases:
  Case 1 'no new empty block arises': Then we have:

$$\mathit{blocks}^{=\emptyset}(C_j^1) = \mathit{blocks}^{=\emptyset}(C_i^1)\ \wedge\ \mathit{blocks}^{=\emptyset}(C_j^2) = \mathit{blocks}^{=\emptyset}(C_i^2)$$

  Together with equation 7.1, we see:

$$\begin{aligned} h(C_j^1) = h(C_j^2) &\Longrightarrow h(\mathit{blocks}^{=\emptyset}(C_j^1)) = h(\mathit{blocks}^{=\emptyset}(C_j^2)) \\ &\Longrightarrow h(\mathit{blocks}^{=\emptyset}(C_i^1)) = h(\mathit{blocks}^{=\emptyset}(C_i^2)) \\ &\Longrightarrow h(C_i^1) = h(C_i^2) \end{aligned}$$

  Case 2 'a new empty block B^* arises': Here, we have:

$$\mathit{blocks}^{=\emptyset}(C_i^1) = \mathit{blocks}^{=\emptyset}(C_j^1) \cup B^*\ \wedge\ \mathit{blocks}^{=\emptyset}(C_i^2) = \mathit{blocks}^{=\emptyset}(C_j^2) \cup B^*$$

  Applying Lemma 116 with $A = \mathit{blocks}^{=\emptyset}(C_j^1), B = \mathit{blocks}^{=\emptyset}(C_j^2)$ and $C = D = B^*$, we have:

$$\begin{aligned} & h(\mathit{blocks}^{=\emptyset}(C_j^1) \cup B^*) = h(\mathit{blocks}^{=\emptyset}(C_j^2) \cup B^*) \\ \Longrightarrow\ & h(\mathit{blocks}^{=\emptyset}(C_i^1)) = h(\mathit{blocks}^{=\emptyset}(C_i^1)) \\ \Longrightarrow\ & h(C_i^1) = h(C_i^2). \end{aligned}$$

  In both cases, we have $h(C_i^1) = h(C_i^2)$ and hence R is preserved by the forget-node-algorithm.

- The `Introduce-Node-Algorithm`:
  Node i is an introduce node with child j. Let C_j^1, C_j^2 be representatives of node j. Also, let $C_i^{1,up}, C_i^{2,up}, C_i^{1,down}, C_i^{2,down}$ be representatives of node i which are the results of the `Introduce-Node-Algorithm` of representatives C_j^1, C_j^2, with respect to newly introduced vertex being up or down.

$$\begin{aligned} h(C_j^1) = h(C_j^2) &\Longrightarrow \mathit{blocks}^{\neq\emptyset}(C_j^1) = \mathit{blocks}^{\neq\emptyset}(C_j^2) \\ &\Longrightarrow \mathit{blocks}^{\neq\emptyset}(C_i^{1,up}) = \mathit{blocks}^{\neq\emptyset}(C_i^{2,up})\ \wedge \\ &\qquad \mathit{blocks}^{\neq\emptyset}(C_i^{1,down}) = \mathit{blocks}^{\neq\emptyset}(C_i^{2,down}) \end{aligned} \tag{7.2}$$

  Furthermore, the algorithm does not modify the empty blocks of the classes:

$$\mathit{blocks}^{=\emptyset}(C_j^1) = \mathit{blocks}^{=\emptyset}(C_i^{1,up}) = \mathit{blocks}^{=\emptyset}(C_i^{1,down})$$

$$\mathit{blocks}^{=\emptyset}(C_j^2) = \mathit{blocks}^{=\emptyset}(C_i^{2,up}) = \mathit{blocks}^{=\emptyset}(C_i^{2,down})$$

Thus, we have:

$$\begin{aligned} h(C_j^1) = h(C_j^2) &\Longrightarrow h(\mathit{blocks}^{=\emptyset}(C_j^1)) = h(\mathit{blocks}^{=\emptyset}(C_j^2)) \\ &\Longrightarrow h(\mathit{blocks}^{=\emptyset}(C_i^{1,up})) = h(\mathit{blocks}^{=\emptyset}(C_i^{2,up})) \end{aligned}$$

Using this and equation 7.2, we see:

$$\begin{aligned} h(C_i^{1,up}) &= h(\mathit{blocks}^{=\emptyset}(C_i^{1,up}) \cup \mathit{blocks}^{\neq\emptyset}(C_i^{1,up})) \\ &= h(\mathit{blocks}^{=\emptyset}(C_i^{1,up})) \cup \mathit{blocks}^{\neq\emptyset}(C_i^{1,up}) \\ &= h(\mathit{blocks}^{=\emptyset}(C_i^{2,up})) \cup \mathit{blocks}^{\neq\emptyset}(C_i^{2,up}) \\ &= h(C_i^{2,up}) \end{aligned}$$

The statement $h(C_i^{1,down}) = h(C_i^{2,down})$ follows directly from:

$$\mathit{blocks}^{\neq\emptyset}(C_j^1) = \mathit{blocks}^{\neq\emptyset}(C_i^{1,down}) \;\wedge\; \mathit{blocks}^{\neq\emptyset}(C_j^2) = \mathit{blocks}^{\neq\emptyset}(C_i^{2,down})$$

We conclude that R is preserved by the `Introduce-Node-Algorithm`.

- The `Join-Node-Algorithm`:
  Node i is a join node with children j_1, j_2. Let $C_{j_1}^1$, $C_{j_1}^2$, $C_{j_2}^3$, $C_{j_2}^4$ be compatible classes of nodes j_1 and j_2, respectively. Let C_i^{13} be the result of the `Join-Node-Algorithm` when processing $C_{j_1}^1$ and $C_{j_2}^3$. C_i^{14}, C_i^{23} and C_i^{24} are defined in the same way. We have:

$$\begin{aligned} h(C_{j_1}^1) = h(C_{j_1}^2) \Longrightarrow\ & \mathit{blocks}^{\neq\emptyset}(C_{j_1}^1) = \mathit{blocks}^{\neq\emptyset}(C_{j_1}^2) \;\wedge \\ & h(\mathit{blocks}^{=\emptyset}(C_{j_1}^1)) = h(\mathit{blocks}^{=\emptyset}(C_{j_1}^2)) \end{aligned}$$

An analogous implication can be obtained with $h(C_{j_2}^3) = h(C_{j_2}^4)$. Let C be a multiset of blocks, then $REPAIR(C)$ is the result of the REPAIR-algorithm of Section 7.1.2 applied to C. Then we can see:

$$\begin{aligned} \mathit{blocks}^{\neq\emptyset}(C_i^{13}) &= REPAIR(\mathit{blocks}^{\neq\emptyset}(C_{j_1}^1) \cup \mathit{blocks}^{\neq\emptyset}(C_{j_2}^3)) \\ &= REPAIR(\mathit{blocks}^{\neq\emptyset}(C_{j_1}^2) \cup \mathit{blocks}^{\neq\emptyset}(C_{j_2}^4)) \\ &= \mathit{blocks}^{\neq\emptyset}(C_i^{24}) \end{aligned} \tag{7.3}$$

Applying Lemma 116 with $A = \mathit{blocks}^{=\emptyset}(C_{j_1}^1)$, $B = \mathit{blocks}^{=\emptyset}(C_{j_1}^2)$, $C = \mathit{blocks}^{=\emptyset}(C_{j_2}^3)$ and $D = \mathit{blocks}^{=\emptyset}(C_{j_2}^4)$, we get:

$$\begin{aligned} h(\mathit{blocks}^{=\emptyset}(C_i^{13})) &= h(\mathit{blocks}^{=\emptyset}(C_{j_1}^1) \cup \mathit{blocks}^{=\emptyset}(C_{j_2}^3)) \\ &= h(\mathit{blocks}^{=\emptyset}(C_{j_1}^2) \cup \mathit{blocks}^{=\emptyset}(C_{j_2}^4)) \\ &= h(\mathit{blocks}^{=\emptyset}(C_i^{24})) \end{aligned} \tag{7.4}$$

From equations 7.3 and 7.4 it follows that $h(C_i^{13}) = h(C_i^{24})$. We can see $h(C_i^{13}) = h(C_i^{24}) = h(C_i^{23}) = h(C_i^{14})$ by using symmetric arguments. Now it follows that R is preserved by the join-node-algorithm. □

We conclude that all relations of $\mathcal{R}$ are preserved by the algorithms. As an example, we consider $R = h_{con}^{=\emptyset SL}(h_{con}^{=\emptyset \not{S} L}(h_{con}^{=\emptyset S \not{L}}(h_{del}^{=\emptyset \not{S} \not{L}}(\hat{R}))))$. With this relation $R \in \mathcal{R}$, we can answer the questions which ask for the probabilities of (among others) the following properties: 'at least half of the clients are connected to at least one server' or 'at most x servers are not connected to a client'. Another example is the classical S-terminal reliability problem, which can be solved with, e.g. the following relation: $h_{con}^{=\emptyset SL}(h_{con}^{=\emptyset S \not{L}}(h_{dmulS}^{\neq\emptyset SL}(h_{dmulS}^{\neq\emptyset S \not{L}}(h_{dmulS}^{=\emptyset SL}(h_{dmulS}^{=\emptyset S \not{L}}(h_{del}^{=\emptyset \not{S} L}(h_{del}^{=\emptyset \not{S} \not{L}}(\hat{R}))))))))$.

The next theorem summarises this section.

Theorem 118. *Let $R \in \mathcal{R}$ be an equivalence relation which is fine enough for property Y. There is an $O(n \cdot n_c^2 \cdot n_b^2 \cdot k)$ time algorithm for computing the probability that graph G has Y.*

Proof. The correctness follows directly from Theorem 109, Lemma 117 and the discussion of the global running time in Section 7.5. Also note that for all generated relations, we have that the set of non-empty blocks of any two equivalent scenarios are equal. □

Because of their special structure, we will analyse the running times of generated relations in the next section. We will see that applying certain operations bounds the number of classes n_c polynomially or by a constant.

7.3.3 Running Times of Generated Relations

In general, the running times of the procedures heavily depend on the chosen equivalence relations. The choice of the relation is a crucial point. Hence, it is very hard to make a statement about running times in general. For our generated or constructed relations, however, we are in a slightly better situation. The relation that can be used for the largest collection of properties $\hat{R}$ has exponentially many equivalence classes and therefore needs exponential time. By applying our operations in $\mathcal{H}$, we reduce the number of equivalence classes by a 'roughly' determinable amount.

Instead of giving a list of all possible combinations of operations, we consider a few cases which appear especially useful. As an example, when using operation $h_{dmulS}^{=\emptyset S \not{L}}$, it can be reasonable to use operation $h_{dmulS}^{\neq\emptyset S \not{L}}$ (and $h_{dmulS}^{=\emptyset SL}$) as well. Furthermore, operation $h_{del}^{=\emptyset \not{S} \not{L}}$ can almost always be used, since empty blocks without flags cannot be used to make further connections, and they do not give any information about servers or clients.

When giving upper bounds for the number of representatives, we can distinguish between the number of possibilities due to the nonempty blocks and due to the empty blocks. For now, we only consider the number of possibilities due to the nonempty blocks, with no respect to flags. This number of possibilities due to the nonempty blocks is a constant and hence is not influenced by other input parameters apart from the treewidth k. Therefore, we have to consider at most $k+1$ vertices per node which can be taken to form blocks and classes. A rough upper bound for this number is

$$\Psi := \sum_{i=1}^{k+1} \binom{k+1}{i} \sum_{j=1}^{i} \left\{ {i \atop j} \right\}$$

This can be shown as follows. The rightmost term gives the number of partitions of i elements into j nonempty subsets. It is known as the Stirling number of the second kind. See [52] for notations and details. With the inner summation, we get the number of partitions of i elements into at most i nonempty subsets. The binomial coefficient chooses i elements for partitioning among $k+1$ elements, which are the vertices of one node of the tree-decomposition. Since not all nodes must have exactly

$k+1$ vertices and not all vertices have to be up, we have to use the left summation. Although, this number can be rather large, it is a constant (which we refer to as Ψ) when k considered a constant. After this preparatory work, we consider in the following cases some examples of sequences h of operations in $\mathcal{H}$.

Case 1 ‘the sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{dmulS}^{=\emptyset S \not L}, h_{dmulS}^{=\emptyset SL}, h_{dmulS}^{\neq\emptyset S \not L}, h_{dmulS}^{\neq\emptyset SL}$’:

Any nonempty block can have 0 or 1 S-flags and between 0 and n_L L-flags. These give at most $2 \cdot (n_L+1)$ possibilities per nonempty block. When working with a tree-decomposition of width k, there can be at most $k+1$ nonempty blocks in any representative. For $k+1$ blocks, this results in at most $(2 \cdot (n_L+1))^{k+1}$ possibilities.

Now, we look at the number of possible equivalence classes caused by the empty blocks. We denote the number of possibilities of empty blocks with at least one L-flag and no S-flag by Ψ'. This number only depends on n_L. The number of possibilities with exactly i blocks with at least one L-flag and no S-flag is at most the number of ways to choose a $i-1$-element subset from an n_L-1-element set, because we can choose $i-1$ positions for placing separating lines in a sequence with n_L flags. These separating lines then define exactly i segments of the sequence of n_L flags. Therefore, we have (see e.g. [52] for notations and the last inequality):

$$\Psi' \leq \sum_{i=1}^{n_L} \binom{n_L-1}{i-1} = \sum_{i=0}^{n_L-1} \binom{n_L-1}{i} \leq 2^{n_L-1}$$

Each of these blocks can have 0 or 1 S-flags and since we have at most n_L such blocks, this results in at most 2^{n_L} possibilities. Furthermore, we can have between 0 and n_S empty blocks with exactly one S-flag and no L-flags. (There are no empty blocks without flags.) Altogether, we have that the number of equivalence classes is bounded by:

$$\Psi \cdot (2 \cdot (n_L+1))^{k+1} \quad \cdot \quad \Psi' \cdot 2^{n_L} \cdot (n_S+1)$$

Example 119. With the relation

$$R = h_{del}^{=\emptyset \not S \not L}(h_{dmulS}^{=\emptyset S \not L}(h_{dmulS}^{=\emptyset SL}(h_{dmulS}^{\neq\emptyset S \not L}(h_{dmulS}^{\neq\emptyset SL}(\hat{R})))))$$

we can solve among others the following problem for every x: ‘What is the probability that there are at least x clients connected to each server?’ (we call this problem $[\geq xL \leftrightarrow \forall S]$).

Corollary 120. *If we do not allow empty blocks without any flags and if we bound the number of S-flags per block to be at most one, we have $O(n_S \cdot 2^{n_L})$ classes per node, i.e. linear in n_S, but exponential in n_L.*

Case 2 ‘the sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{dmulS}^{=\emptyset S \not L}, h_{dmulL}^{=\emptyset \not S L}, h_{dmulS}^{=\emptyset SL}, h_{dmulL}^{=\emptyset SL}, h_{dmulS}^{\neq\emptyset S \not L}, h_{dmulL}^{\neq\emptyset \not S L}, h_{dmulS}^{\neq\emptyset SL}, h_{dmulL}^{\neq\emptyset SL}$’:

For a nonempty block, there are 4 possibilities: no flag, one S-flag, one L-flag, one of each flags. Hence, we have at most 4^{k+1} possibilities for at most $k+1$ nonempty blocks. One equivalence class can have at most n_S empty blocks with exactly only one S-flag. The situation is similar for blocks with only L-flags, and the number of empty blocks with one S-flag and one L-flag is bounded by $\min(n_S, n_L)$. In this case, the number of equivalence classes is bounded by:

$$\Psi \cdot 4^{k+1} \quad \cdot \quad (n_S+1) \cdot (n_L+1) \cdot (\min(n_S, n_L)+1)$$

Example 121. With the relation

$$R = h_{del}^{=\emptyset \not S \not L}(h_{dmulS}^{=\emptyset S \not L}(h_{dmulL}^{=\emptyset \not S L}(h_{dmulS}^{=\emptyset S L}(\\ h_{dmulL}^{=\emptyset S L}(h_{dmulS}^{\neq\emptyset S \not L}(h_{dmulL}^{\neq\emptyset \not S L}(h_{dmulS}^{\neq\emptyset S L}(h_{dmulL}^{\neq\emptyset S L}(\hat{R})))))))))$$

we can solve among others the following problem for every x and y: 'What is the probability that there are x connected components with at least one client and no server, while there are at least y connected components containing at least one server and no client?'

Corollary 122. *If we do not allow empty blocks without any flags and if we bound the number of S-flags and L-flags per block to be at most one, respectively, we have $O(n_S \cdot n_L \cdot \min(n_S, n_L))$ classes per node.*

Case 3 'the sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{con}^{=\emptyset S \not L}, h_{con}^{=\emptyset \not S L}, h_{con}^{=\emptyset S L}$':

For one nonempty block, we have at most $(n_S + 1) \cdot (n_L + 1)$ possibilities for the number of L- and S-flags. Considering $k+1$ nonempty blocks results in $((n_S+1) \cdot (n_L+1))^{k+1}$ possibilities. We have at most one empty block with S-flags and no L-flags, which gives $n_S + 1$ possibilities. The situation is similar for a possible block with only L-flags, and if we have a block with both types of flags, then there are $n_S \cdot n_L$ possible flag distributions. Multiplying these bounds results in an upper bound for the number of equivalence classes:

$$\Psi \cdot ((n_S + 1) \cdot (n_L + 1))^{k+1} \quad \cdot \quad (n_S + 1) \cdot (n_L + 1) \cdot (n_S \cdot n_L + 1)$$

Example 123. With the relation

$$R = h_{del}^{=\emptyset \not S \not L}(h_{con}^{=\emptyset S \not L}(h_{con}^{=\emptyset \not S L}(h_{con}^{=\emptyset S L}(\hat{R}))))$$

we can solve among others the following problem: 'What is the probability that at least x clients are not connected to a server while at least y servers have no client connected to it?' ($[\geq xL \not\leftrightarrow S \wedge \geq yS \not\leftrightarrow L]$).

Corollary 124. *If we do not allow empty blocks without any flags and if we have at most one empty block with S-flags, at most one empty block with L-flags and at most one empty block with S- and L-flags, then we have $O(n_S^{k+3} \cdot n_L^{k+3})$ classes per node, i.e. polynomial in n_S and n_L.*

Case 4 'the sequence h contains: $h_{del}^{=\emptyset \not S \not L}$, $h_{con}^{=\emptyset S \not L}$, $h_{con}^{=\emptyset \not S L}$, $h_{con}^{=\emptyset S L}$, $h_{dmulS}^{=\emptyset S \not L}$, $h_{dmulL}^{=\emptyset \not S L}$, $h_{dmulS}^{=\emptyset S L}$, $h_{dmulL}^{=\emptyset S L}$, $h_{dmulS}^{\neq\emptyset S \not L}$, $h_{dmulL}^{\neq\emptyset \not S L}$, $h_{dmulS}^{\neq\emptyset S L}$, $h_{dmulL}^{\neq\emptyset S L}$ and the $h_{con}^{\cdots}$ operations are applied earlier than the $h_{dmulS}^{\cdots}$ and $h_{dmulL}^{\cdots}$ operations':

This means, that after applying h each block can have at most one flag of each type. As in Case 2, we have for the nonempty blocks 4^{k+1} possibilities. There are only 8 possibilities for the empty blocks, since we only can have the following three blocks: $\{\ \}^S, \{\ \}_L^S, \{\ \}_L$, which may or may not exist in an equivalence class. Hence, we have 8 possibilities. Altogether, the number of equivalence classes is bounded by:

$$\Psi \cdot 4^{k+1} \quad \cdot \quad 8$$

Example 125. With the relation

$$R = h_{dmulS}^{=\emptyset S \not L}(h_{dmulL}^{=\emptyset \not S L}(h_{dmulS}^{=\emptyset SL}(h_{dmulL}^{=\emptyset SL}(h_{dmulS}^{\neq\emptyset S \not L}(h_{dmulL}^{\neq\emptyset \not S L}(h_{dmulS}^{\neq\emptyset SL}(h_{dmulL}^{\neq\emptyset SL}($$

$$h_{del}^{=\emptyset \not S \not L}(h_{con}^{=\emptyset S \not L}(h_{con}^{=\emptyset \not S L}(h_{con}^{=\emptyset SL}(\hat{R})))))))))))$$

we can solve among others the following problem: ‘What is the probability that each client is connected to a server while each server has a client connected to it?’ ($[L \leftrightarrow \geq 1S \wedge S \leftrightarrow \geq 1L]$).

Corollary 126. *If we do not allow empty blocks without any flags and if we bound the number of S- and L-flags per block to be at most one, respectively, and if we bound the number of empty blocks to be constant, then we have a constant number of classes per node.*

These four corollaries show, how applying operations from $\mathcal{H}$ can help to reduce the number of classes. Recall that this yields algorithms that compute the corresponding probabilities of the properties for graphs, given with a tree-decomposition of width at most k. The running time of such an algorithm is $O(n)$ times the number of equivalence classes (i.e. representatives) per node. In the next section, we consider answering questions, which ask for the probability that the surviving subgraph has property Y_1 and property Y_2. For such a combination of properties, we can use a combination of equivalence relations.

7.3.4 Combining Relations

Once we have relations for solving problems, we can combine them to create relations for ‘combined problems’. We know that many relations allow us to solve more than one problem. Assume that we are faced with a problem like ‘What is the probability that the surviving subgraph has properties A and B?’ (e.g. problem $[\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$) Let us assume further that we already have relations R^A and R^B for computing the probabilities of A and B, respectively. We will see that we can use R^A and R^B to create a new relation for property $(A \wedge B)$.

Lemma 127. *Let R^A and R^B be two relations, that are proper and fine enough for property A and B, respectively. Let $R^{A\wedge B}$ be the relation, such that for all C^1 and C^2 that are representatives of scenarios f_1 and f_2, respectively, we have:*

$$(C^1, C^2) \in R^{A\wedge B} \iff (C^1, C^2) \in R^A \wedge (C^1, C^2) \in R^B$$

Then $R^{A\wedge B}$ is proper and fine enough for property $(A \wedge B)$.

Proof. We have to show three properties: $blocks^{\neq\emptyset}(C^1) = blocks^{\neq\emptyset}(C^2)$, the preservation by the algorithms and the fineness of $R^{A\wedge B}$.

The first property is easy to see, because two scenario representatives C^1 and C^2 can only be equivalent under $R^{A\wedge B}$ if they are equivalent under R^A and R^B. Since R^A and R^B are proper, we have: $blocks^{\neq\emptyset}(C^1) = blocks^{\neq\emptyset}(C^2)$.

To see the preservation by the algorithms, we look at a join node i with children j_1 and j_2. We use the notation introduced in Section 7.1.4 and utilised in Section 7.2.5. Then we have:

$$\begin{array}{rccc}
& (C^1_{j_1}, C^2_{j_1}) \in R^{A\wedge B} & \wedge & (C^1_{j_2}, C^2_{j_2}) \in R^{A\wedge B} \\
\Longrightarrow & \begin{bmatrix} (C^1_{j_1}, C^2_{j_1}) \in R^{A} \\ \wedge \\ (C^1_{j_1}, C^2_{j_1}) \in R^{B} \end{bmatrix} & \wedge & \begin{bmatrix} (C^1_{j_2}, C^2_{j_2}) \in R^{A} \\ \wedge \\ (C^1_{j_2}, C^2_{j_2}) \in R^{B} \end{bmatrix} \\
\Longrightarrow & \begin{bmatrix} (C^1_{j_1}, C^2_{j_1}) \in R^{A} \\ \wedge \\ (C^1_{j_2}, C^2_{j_2}) \in R^{A} \end{bmatrix} & \wedge & \begin{bmatrix} (C^1_{j_1}, C^2_{j_1}) \in R^{B} \\ \wedge \\ (C^1_{j_2}, C^2_{j_2}) \in R^{B} \end{bmatrix} \\
\Longrightarrow & \begin{bmatrix} (C^{11}_i, C^{12}_i) \in R^{A} \wedge \\ (C^{12}_i, C^{21}_i) \in R^{A} \wedge \\ (C^{21}_i, C^{22}_i) \in R^{A} \end{bmatrix} & \wedge & \begin{bmatrix} (C^{11}_i, C^{12}_i) \in R^{B} \wedge \\ (C^{12}_i, C^{21}_i) \in R^{B} \wedge \\ (C^{21}_i, C^{22}_i) \in R^{B} \end{bmatrix} \\
\Longrightarrow & \begin{bmatrix} (C^{11}_i, C^{12}_i) \in R^{A\wedge B} \wedge \\ (C^{12}_i, C^{21}_i) \in R^{A\wedge B} \wedge \\ (C^{21}_i, C^{22}_i) \in R^{A\wedge B} \end{bmatrix} & &
\end{array}$$

Hence, $R^{A\wedge B}$ is preserved by the `Join-Node-Algorithm`. The preservation by the `Introduce-` and `Forget-Node-Algorithm` can be proven in an analogous way.

Next, we prove the fineness of the relations. We look at a node i of T. There we select an equivalence class representative C of $R^{A\wedge B}$. We will derive a contradiction: Assume we have two scenario representatives C^1 and C^2 with: C^1 has property $(A \wedge B)$ and C^2 does not have $(A \wedge B)$ and $C^1, C^2 \in C$. That means 'C^1 has A and C^1 has B' and 'C^2 does not have A or C^2 does not have B' and:

$$\begin{array}{ccc}
(C^1, C^2) \in R^{A\wedge B} & \Longrightarrow & \\
(C^1, C^2) \in R^{A} & \wedge & (C^1, C^2) \in R^{B} \\
& \Longrightarrow & \\
\begin{bmatrix} [C^1 \text{ has } A \;\wedge\; C^2 \text{ has } A] \\ \vee \\ [C^1 \text{ has } \neg A \wedge C^2 \text{ has } \neg A] \end{bmatrix} & \wedge & \begin{bmatrix} [C^1 \text{ has } B \;\wedge\; C^2 \text{ has } B] \\ \vee \\ [C^1 \text{ has } \neg B \wedge C^2 \text{ has } \neg B] \end{bmatrix}
\end{array}$$

From the assumption 'C^1 has $(A \wedge B)$', we have:

$$C^1 \text{ has } A \wedge C^1 \text{ has } B \Longrightarrow C^2 \text{ has } A \wedge C^2 \text{ has } B,$$

which is a contradiction to 'C^2 has $\neg(A \wedge B)$', and hence $R^{A\wedge B}$ is fine enough for $(A \wedge B)$, which completes the proof. □

Other Combinations.

The previous lemma only considers the combination of two relations with the logical 'and'. A natural question to ask is whether combining relations can also be applied to the logical 'not' and the logical 'or'.

Given a graph property A and a relation R^A to compute the probability p^A that the surviving subgraph has property A. To compute the probability $p^{\neg A}$ that the surviving subgraph does not have property A, i.e. it has $\neg A$, we can simply compute $p^{\neg A} = 1 - p^A$ using R^A. Hence, the relations R^A and $R^{\neg A}$ for property A and $\neg A$, respectively, are identical.

We assume now, that we have two relations R^A and R^B for properties A and B, respectively. Defining $R^{A \vee B}$ in the way of Lemma 127 will not result in an equivalence relation, because due to the transitivity of such a relation, two scenarios would be equivalent under $R^{A \vee B}$ even if they are neither equivalent under R^A nor under R^B. However, using basic logical rules, we still can compute the probability $p^{A \vee B}$ that the surviving subgraph has property A or property B (whereat $p^{\neg A \wedge B}$ is the probability that the surviving subgraph does not have property A, but it has property B; $p^{A \wedge B}, p^{A \wedge \neg B}, p^{\neg A \wedge \neg B}$ are defined accordingly):

$$p^{A \vee B} = p^{A \wedge B} + p^{\neg A \wedge B} + p^{A \wedge \neg B} = 1 - p^{\neg A \wedge \neg B}$$

The two previous Lemmas 117 and 127 give us powerful tools for generating and combining relations. For these new relations, it is not necessary to prove the properness and fineness, respectively, as long as the original relations have these properties. We will look at examples of questions that can be answered with the framework:

- What is the probability that in the surviving subgraph not every client is connected to a server? (problem $[\geq 1L \not\leftrightarrow S]$)
- What is the probability that in the surviving subgraph each client is connected to a server and that each server has a client connected to it? ($[L \leftrightarrow \geq 1S \wedge S \leftrightarrow \geq 1L]$)
- What is the probability that in the surviving subgraph all servers are connected or that each server is connected to at least 5 clients? ($[S \leftrightarrow t \vee S \leftrightarrow \geq 5L]$)

For these, we need equivalence relations for the single properties in each question. These single properties, however, fit into our framework and hence, due to Lemma 127, the three combined properties of the questions above fit as well.

7.4 Example of an Extension – The Expected Number of Components with a Certain Property

We give a description of an example relation R, which cannot be generated by the methods of the previous sections. The goal is to answer the questions: ‘What is the expected number of components with at least one server and no clients (at least one client, no servers; at least one client and at least one server)?’ ($[Ec \geq 1(S \wedge \not{L})]$). This question differs from ‘What is the probability that G has property Y?’ Even though, our framework can be used to answer this question, since the representatives of the equivalence classes ‘show enough information’. However, some modifications or extensions are necessary.

Definition of R

We consider two scenario representatives C^1 and C^2 of node i. The scenarios represented by C^1 and C^2 are defined to be equivalent, if they are equal on their nonempty blocks, i.e.:

$$(C^1, C^2) \in R \Longleftrightarrow$$

$$h_{del}^{=\emptyset\not S\not L}(h_{del}^{=\emptyset S\not L}(h_{del}^{=\emptyset\not SL}(h_{del}^{=\emptyset SL}(C^1)))) = h_{del}^{=\emptyset\not S\not L}(h_{del}^{=\emptyset S\not L}(h_{del}^{=\emptyset\not SL}(h_{del}^{=\emptyset SL}(C^2))))$$

Even though we can use the operations of Section 7.3.2 to define the considered equivalence relation, we are not able to use our framework without further modifications. This is because the representative C of an equivalence class $[C]$ of node i will have a special structure. All empty blocks will be replaced by three special blocks: $\{e_S(C)\}^S, \{e_L(C)\}_L$ and $\{e_{SL}(C)\}_L^S$. The nonempty blocks will be the same as the ones of the represented scenarios and $e_S(C), e_L(C)$ and $e_{SL}(C)$ are real numbers. We will only go into details for $e_S(C)$, since for $e_L(C)$ and $e_{SL}(C)$ it is analogue. The number $e_S(C)$ is the expected number of components of G_i, with empty intersection with X_i, with at least one server and no client given the fact that G_i is in one of the scenarios represented by C. The number of components of $G[V_i^{f=1}]$ with at least one server and no client and with empty intersection with X_i is $|blocks^{=\emptyset S\not L}(blocks_i(f))|$, which is the number of empty blocks with at least one S-flag and no L-flags of the representative for scenario f. Then we have:

$$e_S(C) = \sum_{f \in C} \Pr(f) \cdot |blocks^{=\emptyset S\not L}(blocks_i(f))|$$

Fineness and Preservation of R

The preservation is easy to see, since the algorithms only affect the nonempty blocks. Thus, in each case equivalent scenarios are treated in the same way and, hence, are equivalent after processing by the algorithms. A formal proof uses the same ideas and would be similar to the proof of Lemma 117.

The fineness of a relation is defined using the fact that scenarios can or cannot have specific properties. On the other hand, 'the expected number of components with at least on server and no client' is a number and not a property. That is why the term 'fine enough' is not really applicable for our question. However, we can use the equivalence classes of R in the following way. We consider a node i and all its equivalence classes $[C]$. Then, the expected number of components of G_i with at least one server and no client is:

$$\sum_{[C]:[C] \text{ is eq.class of } i} \Pr([C]) \cdot (e_S(C) + |blocks^{\neq\emptyset S\not L}(C)|)$$

Modification of the Algorithms

Since we have a differently structured problem, we modify our algorithms slightly. These modifications maintain e_S from node to node (and also e_L and e_{SL}). Introduce-nodes need no special treatment, because the set of empty blocks is not changed. This is different with forget-nodes. We consider an equivalence class $[C]$. If we create a new empty block with only S-flags, then we delete it and increase $e_S(C)$ by one. The correctness of this is easy to see, since all scenarios of $[C]$ would now have one more empty block with only S-flags. For a join-node i with children j_1 and j_2, we add the two e_S numbers of the 'source' representatives together to obtain $e_S(C)$ of the resulting class C. This can be understood by bringing into mind that an equivalence class C of i represents scenarios of G_i which contains as subgraphs G_{j_1} and G_{j_2}. Furthermore, the empty components of G_{j_1} and G_{j_2} are not influencing each other.

There is one thing left we have to do: the modification of the code-fragment in Step 2 in Section 7.2.2. If two equivalence classes $[C^1]$ and $[C^2]$ become equivalent, we have to modify $e_S(C)$ of the resulting equivalence class $[C]$, using $e_S(C^1)$ and $e_S(C^2)$ in the following way, which is not hard to see.

$$e_S(C) = \frac{e_S(C^1) \cdot \Pr(C^1) + e_S(C^2) \cdot \Pr(C^2)}{\Pr(C)}$$

(Note that $\Pr(C) \neq 0$, since otherwise there would be a vertex v being up and $p(v) = 0$, or there would be a vertex v being down and $p(v) = 1$.) The numbers $e_L(C)$ and $e_{SL}(C)$ are computed in the same way.

Running Time

The overall running time of the algorithms is $O(n_c^2 \cdot n_b^2 \cdot k^2)$. For our relation R we have $n_b \leq k+1+3$, since we have at most $k+1$ nonempty blocks and 3 special blocks, which actually do not play a role in class distinction. The analysis of the maximum number of equivalence classes is similar to Case 4 in Section 7.3.3. Hence, n_c is a constant.

Conclusion

The problem 'What is the expected number of components with at least one server and no clients?' does not fit into the framework as described in the previous sections. However, we have seen that with additional modifications of the framework, we can obtain an algorithm to solve this problem in linear time on graphs with given tree-decomposition of width at most k.

7.5 Concluding Remarks

We presented a framework for a variety of network reliability problems for graphs of bounded treewidth. The method itself is general enough for extensions which might be necessary to enable faster running times and/or additional properties to check. Those properties that can be checked with classes, i.e. those properties for which our framework is applicable, concern connections between sets of vertices. Examples of network reliability problems were considered throughout the last two chapters. Some of them can be solved in linear time on graphs of bounded treewidth, while others need polynomial time, and some problems even require time that is exponential in certain input-parameters (e.g. the number of clients or servers, n_L or n_S, respectively). The next theorem summarises the computational complexity of the problems in Section 6.2.3 on graphs of bounded treewidth.

Theorem 128. *Let G be a graph with n vertices, and let $tw(G)$ be bounded by some constant k.*

1. *The problems $[2 \leftrightarrow t]$, $[S \leftrightarrow t]$ and $[L \leftrightarrow \geq 1S]$ can be solved in $O(n)$ time.*
2. *The problem $[\geq xS]$ can be solved in $O(n \cdot 2^{O(n_S)})$ time.*
3. *The problem $[Ec \geq 1S]$ can be solved in $O(n)$ time.*
4. *The problems $[\geq x_1L \not\leftrightarrow \geq 1S]$, $[\leq x_2L \leftrightarrow \geq 1S]$, $[< x_3L \not\leftrightarrow \geq 1S]$, $[> x_4L \leftrightarrow \geq 1S]$ and $[1 - [< x_3L \not\leftrightarrow \geq 1S]]$ can be solved in time $O(n \cdot n_S \cdot 2^{O(n_L)})$.*
5. *The problems $[\leq yS \not\leftrightarrow L]$ and $[\geq 1L \leftrightarrow \geq xS]$ can be solved in time $O(n \cdot n_S \cdot 2^{O(n_L)})$.*
6. *The problem $[\geq xL \leftrightarrow S \wedge \geq yS \leftrightarrow L]$ can be solved in polynomial time.*

Proof. To prove the theorem, we use Theorem 118 and the analysis in the cases given in Section 7.3.3. Furthermore, see the discussion in Section 7.5 for remarks on the overall running time.

(1) We can use the relation given in Example 125. It is easy to see that this relation is fine enough for the considered problems. See Corollary 126 for the running time.

(2) We can ignore all L-flags, since they are not important for the problem. At the end, we need scenarios that contain a block with at least x S-flags. A similar relation to the one in Example 119 can be used for this. The analysis of that relation shows the running time.

(3) This problem is dealt with in Section 7.4, where we see that it can be solved in the claimed time bound.

(4) Using the relation defined in Example 119, we can solve the problem in the stated time (see Corollary 120).

(5) See (4) with the roles of L and S exchanged.

(6) We can use the relation in Example 123. The running time is stated in Corollary 124. □

It remains a further research topic, to develop a more powerful framework that can handle further properties, like 'What is the probability that the surviving subgraph has a dominating set of size $\leq k$?'

Treewidth vs. Pathwidth

In Section 7.2.4, we required a relation R to be preserved by all three algorithms. We can refrain from this demand, if we look at graphs of bounded pathwidth. A path-decomposition of G is a tree-decomposition (T, X) of G whereat T is a path. Hence, join-nodes do not appear and thus, R does not have to be preserved by the `Join-Node-Algorithm`. This may enable more relations, but on the other hand we have to use a path-decomposition.

Running Times

The running time for our algorithm heavily depends on the chosen relation R. We can see it is very important to choose a relation 'as coarse as possible'. All considered algorithms run in time $O(n_c^2 \cdot n_b^2 \cdot k^2)$ per node. In [61], Kloks shows that there are nice tree-decompositions with at most $4 \cdot n$ nodes (thus linear in $n = |V(G)|$). Hence, we have the global running time $O(n \cdot n_c^2 \cdot n_b^2 \cdot k^2)$.

Furthermore, the framework provides enough possibilities for using additional tricks, ideas and modifications. It may be necessary to use them to get the best running time, as shown in Section 7.4. One general possibility to decrease the running time would be to delete classes as soon as possible, if they can never have the required property. Another possibility is not to delete whole classes, but single blocks that are not needed for determining the required probability. Fewer blocks result in fewer classes, since we have less objects that can make a difference in class distinction. However, the performance gain of these approaches is not easy to analyse.

References

1. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.
2. S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12:308–340, 1991.
3. S. Arnborg and A. Proskurowski. Characterization and recognition of partial 3-trees. *SIAM J. Alg. Disc. Meth.*, 7:305–314, 1986.
4. S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Disc. Appl. Math.*, 23:11–24, 1989.
5. E. Bachoore and H. L. Bodlaender. New upper bound heuristics for treewidth. Technical Report UU-CS-2004-036, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.
6. M. Behzad, G. Chartrand, and L. Lesniak-Foster. *Graphs and Digraphs*. Pindle, Weber & Schmidt, Boston, 1979.
7. J. R. S. Blair and B. Peyton. An introduction to chordal graphs and clique trees. In A. George, J. R. Gilbert, and J. H. U. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–29, New York, 1993. Springer-Verlag.
8. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.
9. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25:1305–1317, 1996.
10. H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS'97*, pages 19–36, Berlin, 1997. Springer-Verlag, Lecture Notes in Computer Science, vol. 1295.
11. H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.
12. H. L. Bodlaender. Necessary edges in k-chordalizations of graphs. *Journal of Combinatorial Optimization*, 7:283–290, 2003.
13. H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, frontsize, and minimum elimination tree height. *J. Algorithms*, 18:238–255, 1995.
14. H. L. Bodlaender and A. M. C. A. Koster. On the Maximum Cardinality Search lower bound for treewidth. In J. Hromkovič, M. Nagl, and B. Westfechtel, editors, *Proc. 30th International Workshop on Graph-Theoretic Concepts in Computer Science WG 2004*, pages 81–92. Springer-Verlag, Lecture Notes in Computer Science 3353, 2004.

15. H. L. Bodlaender and A. M. C. A. Koster. Safe separators for treewidth. In *Proceedings 6th Workshop on Algorithm Engineering and Experiments ALENEX04*, pages 70–78, 2004.
16. H. L. Bodlaender, A. M. C. A. Koster, F. v. d. Eijkhof, and L. C. van der Gaag. Pre-processing for triangulation of probabilistic networks. In J. Breese and D. Koller, editors, *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, pages 32–39, San Francisco, 2001. Morgan Kaufmann.
17. H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. In S. Albers and T. Radzik, editors, *Proceedings 12th Annual European Symposium on Algorithms, ESA2004*, pages 628–639. Springer, Lecture Notes in Computer Science, vol. 3221, 2004.
18. H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. Technical Report UU-CS-2004-34, Dept. of Computer Science, Utrecht University, Utrecht, The Netherlands, 2004.
19. H. L. Bodlaender and R. H. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Disc. Math.*, 6:181–188, 1993.
20. H. L. Bodlaender and T. Wolle. Contraction degeneracy on cographs. Technical Report UU-CS-2004-031, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.
21. H. L. Bodlaender and T. Wolle. A note on the complexity of network reliability problems. Technical Report UU-CS-2004-001, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.
22. J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, MacMillan, New York, London, 1976.
23. Boost C++ Libraries. `http://www.boost.org`, 2005-02-21.
24. K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using *pq*-tree algorithms. *J. Comp. Syst. Sc.*, 13:335–379, 1976.
25. R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.
26. A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes – A Survey*. Society for Industrial and Applied Mathematics, Philadelphia, 1999.
27. A. Bretscher, D. Corneil, M. Habib, and C. Paul. A simple linear time LexBFS cograph recognition algorithm. In H. L. Bodlaender, editor, *Proceedings 29th International Workshop on Graph-Theoretic Concepts in Computer Science WG'03*, pages 119–130. Springer Verlag, Lecture Notes in Computer Science, vol. 2880, 2003.
28. B. Burgstaller, J. Blieberger, and B. Scholz. On the tree width of ada programs. In A. Llamosí and A. Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004: 9th Ada-Europe International Conference on Reliable Software Technologies*, pages 78–90. Springer-Verlag, Lecture Notes in Computer Science, vol. 3063, 2004.
29. J. Carlier and C. Lucet. A decomposition algorithm for network reliability evaluation. *Discrete Applied Mathematics*, 65:141–156, 1996.
30. K. Cattell, M. J. Dinneen, R. G. Downey, M. R. Fellows, and M. A. Langston. On computing graph minor obstruction sets. *Theor. Comp. Sc.*, 233:107–127, 2000.
31. F. Clautiaux, S. N. A. Moukrim, and J. Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Oper. Res.*, 38:13–26, 2004.
32. F. Clautiaux, J. Carlier, A. Moukrim, and S. Négre. New lower and upper bounds for graph treewidth. In J. D. P. Rolim, editor, *Proceedings International Workshop on Experimental and Ef-*

ficient Algorithms, WEA 2003, pages 70–80. Springer Verlag, Lecture Notes in Computer Science, vol. 2647, 2003.
33. W. Cook and P. D. Seymour. Tour merging via branch-decomposition. *Informs J. on Computing*, 15(3):233–248, 2003.
34. G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.
35. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., USA, 1989.
36. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms – Second Edition*. MIT Press, Cambridge, Mass., USA, 2001.
37. D. G. Corneil, H. Lerchs, and L. Stewart Burlingham. Complement reducible graphs. *Annals Discrete Math.*, 1:145–162, 1981.
38. D. G. Corneil, Y. Perl, and L. K. Stewart. A linear recognition algorithm for cographs. *SIAM J. Comput.*, 14:926–934, 1985.
39. B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
40. B. Courcelle. The monadic second-order logic of graphs III: Treewidth, forbidden minors and complexity issues. *Informatique Théorique*, 26:257–286, 1992.
41. G. Demoucron, Y. Malgrange, and R. Pertuiset. Graphes planaires: reconnaissance et construction de representations planaires topologiques. *Rev. Francaise Recherche Operationelle*, 8:33–47, 1964.
42. R. Diestel. *Graph Theory*. Springer-Verlag, New York, 2000.
43. R. Diestel. Graph theory – electronic edition 2000. `http://www.math.uni-hamburg.de/home/diestel/books/graph.theory`, 2005-01-26.
44. The second DIMACS implementation challenge: NP-Hard Problems: Maximum Clique, Graph Coloring, and Satisfiability. `http://dimacs.rutgers.edu/Challenges/`, 1992–1993.
45. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1998.
46. F. v. d. Eijkhof and H. L. Bodlaender. Safe reduction rules for weighted treewidth. In L. Kučera, editor, *Proceedings 28th Int. Workshop on Graph Theoretic Concepts in Computer Science, WG'02*, pages 176–185. Springer Verlag, Lecture Notes in Computer Science, vol. 2573, 2002.
47. I. S. Filotti, G. L. Miller, and J. Reif. On determining the genus of a graph in $O(V^{O(g)})$ steps. In *Proceedings of the 11th Annual Symposium on Theory of Computing*, pages 27–37, New York, 1979. ACM Press.
48. L. Fortnow. Counting complexity. In L. A. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 81–107. Springer-Verlag, 1997.
49. M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
50. V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In proceedings UAI'04, Uncertainty in Artificial Intelligence, 2004.
51. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
52. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, Amsterdam, 1989.
53. J. Gross and J. Yellen. *Graph Theory and Its Applications*. New York, CRC Press, 1999.
54. F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.
55. I. V. Hicks. Planar branch decompositions I: The ratcatcher. INFORMS Journal on Computing (to appear, 2005).

56. D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, Boston, 1997.
57. J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.
58. F. V. Jensen. *Bayesian Networks and Decision Graphs*. Statistics for Engineering and Information Science, Springer-Verlag, New York, 2001.
59. M. Jerrum. *On the Complexity of Evaluating Multivariate Polynomials*. PhD thesis, Dept. Computer Science, University Edinburgh, 1981.
60. R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85 – 104. Plenum Press, 1972.
61. T. Kloks. *Treewidth*. PhD thesis, Utrecht University, Utrecht, The Netherlands, 1993.
62. A. M. C. A. Koster. *Frequency Assignment - Models and Algorithms*. PhD thesis, Univ. Maastricht, Maastricht, The Netherlands, 1999.
63. A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. In H. Broersma, U. Faigle, J. Hurink, and S. Pickl, editors, *Electronic Notes in Discrete Mathematics*, volume 8. Elsevier Science Publishers, 2001.
64. A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40:170–180, 2002.
65. A. M. C. A. Koster, T. Wolle, and H. L. Bodlaender. Degree-based treewidth lower bounds. Technical Report UU-CS-2004-050, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.
66. A. M. C. A. Koster, T. Wolle, and H. L. Bodlaender. Degree-based treewidth lower bounds. In *Proceedings 4th International Workshop on Experimental and Efficient Algorithms, WEA 2005*. Springer-Verlag, Lecture Notes in Computer Science, to appear, 2005.
67. S. K. Lando and A. K. Zvonkin. *Graphs on Surfaces and Their Applications*. Springer-Verlag, Heidelberg, 2004.
68. S. J. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological)*, 50:157–224, 1988.
69. H. Lerchs. On cliques and kernels. Technical report, Dept. of Computer Science, University of Toronto, 1971.
70. B. Lucena. A new lower bound for tree-width using maximum cardinality search. *SIAM J. Disc. Math.*, 16:345–353, 2003.
71. C. Lucet, J.-F. Manouvrier, and J. Carlier. Evaluating network reliability and 2-edge-connected reliability in linear time for bounded pathwidth graphs. *Algorithmica*, 27:316–336, 2000.
72. E. Mata-Montero. *Reliability of Partial k-Tree Networks*. PhD thesis, University of Oregon, 1990.
73. B. Mohar and C. Thomassen. *Graphs on Surfaces*. The Johns Hopkins University Press, Baltimore, 2001.
74. A compendium of NP optimization problems. `http://www.nada.kth.se/~viggo/problemlist/compendium.html`, 2005-02-17.
75. J. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a network remains connected. *SIAM J. Comput.*, 12(4):777–788, 1983.
76. S. Ramachandramurthi. *Algorithms for VLSI Layout Based on Graph Width Metrics*. PhD thesis, Computer Science Department, University of Tennessee, Knoxville, Tennessee, USA, 1994.
77. S. Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Disc. Math.*, 10:146–157, 1997.
78. G. Ringel. *Map Color Theorem*. Springer-Verlag, Berlin, 1974.

79. N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *J. Comb. Theory Series B*, 35:39–61, 1983.
80. N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322, 1986.
81. N. Robertson and P. D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory Series B*, 63:65–110, 1995.
82. N. Robertson and P. D. Seymour. Graph minors. XX. Wagner's conjecture. *J. Comb. Theory Series B*, 92:325–357, 2004.
83. N. Robertson, P. D. Seymour, and R. Thomas. Quickly excluding a planar graph. *J. Comb. Theory Series B*, 62:323–348, 1994.
84. H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.
85. D. J. Rose. On simple characterization of k-trees. *Disc. Math.*, 7:317–322, 1974.
86. A. Rosenthal. Computing the reliability of complex networks. *SIAM J. Appl. Math.*, 32:384–393, 1977.
87. P. Scheffler. *Die Baumweite von Graphen als ein Maß für die Kompliziertheit algorithmischer Probleme*. PhD thesis, Akademie der Wissenschaften der DDR, Berlin, 1989.
88. P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.
89. K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proc. National Conference on Artificial Intelligence (AAAI '97)*, pages 185–190. Morgan Kaufmann, 1997.
90. R. E. Tarjan and M. Yannakakis. Simple linear time algorithms to test chordiality of graphs, test acyclicity of graphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13:566–579, 1984.
91. C. Thomassen. The graph genus problem is NP-complete. *J. Algorithms*, 10:568–576, 1989.
92. M. Thorup. Structured programs have small tree-width and good register allocation. *Information and Computation*, 142:159–181, 1998.
93. Treewidthlib. `http://www.cs.uu.nl/people/hansb/treewidthlib`, 2004-03-31.
94. L. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 6:410–421, 1979.
95. L. C. van der Gaag and H. L. Bodlaender. Comparing loop cutsets and clique trees in probabilistic networks. Technical Report UU-CS-1997-042, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 1997.
96. T. Wolle. A framework for network reliability problems on graphs of bounded treewidth. In P. Bose and P. Morin, editors, *Algorithms and Computation, Proceedings of the 13th International Symposium, ISAAC 2002*, pages 137–149, Berlin, 2002. Springer-Verlag, Lecture Notes in Computer Science, vol. 2518.
97. T. Wolle. A framework for network reliability problems on graphs of bounded treewidth. Technical Report UU-CS-2003-026, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2003.
98. T. Wolle and H. L. Bodlaender. A note on edge contraction. Technical Report UU-CS-2004-028, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.
99. T. Wolle, A. M. C. A. Koster, and H. L. Bodlaender. A note on contraction degeneracy. Technical Report UU-CS-2004-042, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.

Acknowledgements

I would like to thank all persons who helped me making this thesis. For very useful scientific discussions and comments, I am grateful to Hans L. Bodlaender, Sergio Cabello, Frank van den Eijkhof, Andreas Eisenblätter, Mark Jerrum, Arie M. C. A. Koster, Dieter Kratsch, Jan van Leeuwen, Peter Lennartz, Gerard Tel, Carsten Thomassen, Peter Verbaan, the reading committee of this thesis and others. I am indebted to Ian Sumner, Pauline Vollmerhaus, Annemarie Kerkhoff, Shay Uzery and Peter Verbaan for proof-reading parts of my work.

I would like to extend special thanks to my supervisors Hans L. Bodlaender and Jan van Leeuwen for their supervision, suggestions, explanations and improvements, and also thanks to Arie M. C. A. Koster. It has been a pleasure to work with them. Arie, thank you very much for inviting me to Berlin, and Hans, thank you very much for all your support, stimulations, insight, guidance and patience.

Samenvatting

Grafen zijn een onderdeel van vele systemen. Vaak kunnen we de structuur van projekten, scenario's en data in het algemeen door grafen representeren en modelleren. Op deze abstracte modellen proberen wetenschappers problemen op te lossen die ook in de praktijk zeer belangrijk zijn, of ze tonen aan dat een probleem op algemene grafen waarschijnlijk erg lastig (bijvoorbeeld NP-moeilijk) is. Bekijken we grafen met een speciale structuur, dan kan het toch mogelijk zijn om juist deze structuur te benutten om in het algemeen lastige problemen efficiënt op te lossen. Grafen met begrensde boombreedte hebben zo'n speciale structuur, en zij vormen het hoofdonderwerp van dit proefschrift.

Grafen met begrensde boombreedte zijn in de laatste jaren steeds belangrijker geworden—zowel in theorie als in de praktijk. Bomen zijn ook een speciale klasse grafen, waarop veel problemen efficiënt oplosbaar zijn. De boombreedte van een graaf geeft aan hoe sterk de graaf met een boom verwant is: hoe kleiner de boombreedte, hoe meer verwantschap. Veel praktisch relevante problemen zijn NP-moeilijk voor algemene grafen, terwijl ze gemakkelijk (soms zelfs triviaal) zijn voor bomen. De gelijkenis van grafen met begrensde boombreedte met bomen maakt het mogelijk dat veel (in het algemeen) NP-moeilijke problemen efficiënt opgelost kunnen worden op deze grafen. Het bepalen van de boombreedte van een graaf is echter zelf ook een NP-moeilijk probleem. In dit proefschrift bekijken we methoden om ondergrenzen voor de boombreedte van grafen te berekenen en hoe men de betrouwbaarheid van netwerken kan berekenen op grafen met begrensde boombreedte.

Na een inleiding in Hoofdstuk 1 geven we in Hoofdstuk 2 algemene definities en begrippen uit de grafentheorie, met name bekijken we het contraheren of samentrekken van een kant. Dit is een bekende operatie die een kant $e = \{u, w\}$ en zijn twee eindpunten u en w vervangt door een nieuwe knoop v_e, zodanig dat v_e met alle knopen die ook met u of w door een kant verbonden waren, door een kant verbonden is. Veel eigenschappen van deze operatie zijn intuïtief, wij geven echter een gedetailleerde beschrijving van de operatie, die verder gaat dan bestaande beschrijvingen. In Hoofdstuk 2 geven we een formalisatie van deze operatie. Verder bekijken we in dat hoofdstuk subgrafen en minoren. Voor minoren is het samentrekken van een kant een essentiël punt. We definiëren ook de begrippen boomdecompositie en boombreedte. Aan het einde van Hoofdstuk 2 bekijken we de operatie van het onderverdelen van een kant en het effect op de boombreedte van een graaf van deze operatie. Een kant onderverdelen betekent een knoop op die kant zetten, of met andere woorden, een kant vervangen door een pad van lengte twee.

In Hoofdstuk 3 onderzoeken we een aantal parameters die allemaal een ondergrens voor de boombreedte van een graaf zijn. Het hoofdidee voor nieuwe parameters daarbij is bekende boombreedte-ondergrenzen over alle subgrafen of minoren van de oorspronkelijke graaf te nemen, want de boombreedte van een subgraaf of minor van een graaf is altijd hooguit de boombreedte van de oorspronkelijke graaf. Een van die parameters is $\delta(G)$—de kleinste graad in G. De parameters $\delta D(G)$ (ook

bekend als ‘degeneracy’) en $\delta C(G)$ zijn gedefinieerd als het maximum over alle subgrafen (bij δC: minoren) G' van G, van $\delta(G')$. Zes andere parameters zijn op dezelfde manier gedefinieerd, maar in plaats van δ wordt δ_2 of γ_R gebruikt. De parameter $\delta_2(G)$ is de graad van de knoop met de op een na kleinste graad in G, en $\gamma_R(G)$ is het minimum over alle paren verschillende knopen u en w die niet door een kant verbonden zijn van het maximum van de graad van u en de graad van w. We bewijzen dat alle parameters ondergrenzen voor boombreedte zijn, waarbij we gebruik maken van het feit dat δ, δ_2 en γ_R al bekend zijn als ondergrenzen voor boombreedte. Verder tonen we verhoudingen tussen deze parameters aan en bewijzen de NP-moeilijkheid van het berekenen van sommige van deze parameters. We laten ook zien dat deze parameters matige resultaten (als boombreedte-ondergenzen) zullen geven op grafen met kleine genus. Twee andere parameters zijn gebaseerd op ‘Maximum Cardinality Search’ (MCS). Ook hier combineren wij een bekende boombreedte-ondergrens (namelijk $MCSLB$) met het samentrekken van kanten, dat wil zeggen, met minoren, en krijgen de ondergrens $MCSLBC$, die ook NP-moeilijk is om te berekenen. Aan het einde van Hoofdstuk 3 beschrijven we een al bekende manier om elke andere boombreedte-ondergrens te verbeteren. Deze methode is gebaseerd op zogenaamde ‘graph improvements’.

Na het theoretische onderzoek in Hoofdstuk 3 gaan we in Hoofdstuk 4 de boombreedte-ondergrenzen experimenteel evalueren. Aan het begin van dat hoofdstuk beschrijven we een datastructuur die gebaseerd is op een ‘adjacency-list’ datastructuur en door een ‘bucket’ structuur wordt uitgebreid. Omdat sommige (voor ons belangrijke) bewerkingen zeer efficiënt kunnen worden uitgevoerd met deze datastructuur, is hij geschikt voor vele algoritmen en heuristieken in dit hoofdstuk. Voor de parameters die in polynomiale tijd exact kunnen worden berekend, ontwikkelen we algoritmen en tonen hun correctheid en looptijd aan (voor zover ze niet triviaal zijn). Voor de parameters die NP-moeilijk zijn om te berekenen stellen we een aantal heuristieken voor. Van sommige van deze heuristieken kunnen we aantonen dat ze in de praktijk redelijk goed werken maar theoretisch willekeurig slecht kunnen zijn. Verder introduceren we nog een andere manier om heuristieken gebaseerd op ‘graph improvements’ met het samentrekken van kanten te combineren. In de experimenten in dit hoofdstuk vergelijken we de behaalde boombreedte-ondergrenzen en ook de daarvoor nodige looptijden van de algoritmen en heuristieken. Een niet verrassend resultaat is dat vaak ingewikkelder algoritmen en heuristieken betere boombreedte-ondergrenzen kunnen geven, hoewel dit ten koste gaat van de looptijd. In het algemeen wordt zeer goed duidelijk dat het combineren van boombreedte-ondergrenzen met het samentrekken van kanten (d.w.z. het nemen van het maximum van die ondergrens over minoren van de oorspronkelijke graaf) heel vaak grote verbeteringen op de ondergrens oplevert.

De ondergrens δC heeft een elementair karakter en is een interessant studie-object—niet alleen als boombreedte-ondergrens. De parameter δC is NP-moeilijk te berekenen. Voor een speciale klasse van grafen, de cografen, bekijken we in Hoofdstuk 5 hoe δC wel efficiënt berekend kan worden. In dit hoofdstuk ontwikkelen wij daarvoor een methode die gebaseerd is op dynamisch programmeren. Cografen kunnen worden gerepresenteerd door een boomstructuur (een zogenaamde ‘cotree’) waarvan ons dynamisch programmeer-algoritme gebruik maakt. We bewijzen dat de looptijd van het algoritme polynomiaal is.

In de Hoofdstukken 6 en 7 bekijken we een ander onderwerp, namelijk het berekenen van de netwerkbetrouwbaarheid (voor grafen met begrensde boombreedte). Aan het begin van Hoofdstuk 6 geven we het klassieke model voor netwerkbetrouwbaarheid. Daarna specificeren we ons eigen model: Voor elke knoop v van het netwerk is er een rationeel getal $p(v)$, dat de betrouwbaarheid van deze knoop aangeeft, oftewel die de kans aangeeft dat deze knoop in het netwerk voorhanden is (niet stukgegaan is). Alle knopen kunnen onafhankelijk van elkaar stuk gaan. We nemen aan dat alle kanten perfect betrouwbaar zijn. Dit is geen beperking, omdat we niet-perfecte kanten kunnen simuleren door niet-perfecte knopen, door deze kanten onder te verdelen. Het netwerk dat uit alle niet-kapotte knopen (en kanten) bestaat noemen we het overlevende deelnetwerk. In ons model hebben we twee verzamelingen S ('servers') en L ('clients') van speciale knopen. Verder geven we in dit hoofdstuk een lijst met interessante netwerkbetrouwbaarheidsproblemen, zoals 'Wat is de kans dat elke client verbonden is aan tenminste één server in het overlevende deelnetwerk?' of 'Wat is het verwachte aantal componenten in het overlevende deelnetwerk met tenminste één server?' Van alle vragen in de lijst tonen we aan dat ze $\#P$-moeilijk te beantwoorden zijn voor algemene grafen.

In Hoofdstuk 7 beperken we ons tot grafen met begrensde boombreedte. We tonen aan dat vele van de netwerkbetrouwbaarheidsproblemen uit Hoofdstuk 6 efficiënt oplosbaar zijn. We beschrijven een methode die op dynamisch programmeren gebaseerd is en gebruiken een boomdecompositie als onderliggende structuur voor het dynamisch programmeren. Op die manier krijgen we een framework waarmee we de vragen uit Hoofdstuk 6 kunnen beantwoorden. De looptijd van ons algoritme hangt in sterke mate af van het probleem dat we willen oplossen. Hoe minder informatie we tijdens het berekeningsproces moeten bewaren om uiteindelijk het juiste resultaat te krijgen, hoe korter de looptijd van ons algoritme. De twee vragen 'Wat is de kans dat elke client verbonden is aan tenminste één server in het overlevende deelnetwerk?' en 'Wat is het verwachte aantal componenten in het overlevende deelnetwerk met tenminste één server?' kunnen bijvoorbeeld worden opgelost in een tijd die lineair in het formaat van het netwerk is, maar meestal tenminste exponentieel in de boombreedte van het netwerk. Het hele framework is praktisch voornamelijk toepasbaar op grafen met kleine boombreedte.

Curriculum Vitae

Thomas Wolle was born on 29th October 1975 in Gera, Germany. He got his degree in Computer Science from the Friedrich-Schiller-Universität Jena in 2001. In the same year, he started as a PhD student (AIO) at the Institute for Information and Computing Sciences of Utrecht University in Utrecht, The Netherlands. In 2005, he completed this thesis there.

Titles in the IPA Dissertation Series

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09