Computing Width Parameters of Graphs

Tuukka Korhonen

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Scientific environment

The research leading to the results presented in this thesis was mainly carried out at the Department of Informatics at the University of Bergen. Parts of the research were conducted when the author was performing mandatory non-military service at the Department of Computer Science at the University of Helsinki before his PhD studies. Also, some parts of the research were done during research visits at the Institute of Informatics at the University of Warsaw and at the Department of Computer Science at the University of California Santa Barbara.

The PhD studies of the author were funded by the Research Council of Norway (RCN) project "Beyond Worst-Case Analysis in Algorithms", grant number 314528. The author was also supported by the Meltzer Research Fund. At the University of Helsinki, the author was supported by the Academy of Finland, grant number 322869. The research visits to the University of Warsaw were partly supported by the European Research Council (ERC), grant number 948057.

Acknowledgements

First, I would like to thank my advisor Fedor V. Fomin, for all his guidance and wisdom, and for the enthusiastic attitude he always brings. I have had a lot of fun discussing various topics and solving algorithmic problems with Fedor. I also thank my co-advisor Petr A. Golovach for his vast knowledge he has shared with me, often being the first to provide answers to my questions, and the good times we have had working together. I'm thankful to both Fedor and Petr for providing an excellent environment for me to grow as a researcher.

I am grateful to Daniel Lokshtanov for our many fruitful collaborations and for hosting me in Santa Barbara. Similarly, I thank Michał Pilipczuk for inviting me many times to Warsaw, always leading to cool new algorithms. I also thank Parinya Chalermsook, Maria Chudnovsky, and Mikkel Thorup for having me for research visits. Furthermore, I would like to thank Saket Saurabh and Jan Arne Telle for numerous interesting discussions and the guidance they have given to me.

I thank Matti Järvisalo for introducing me to the world of research, for all his advice during our years working together, and for encouraging me to pursue my research interests. Thanks also to Jeremias Berg for his help at the start of my research journey, and to Mikko Koivisto for his generous advice. I thank Antti Laaksonen and the rest of the Finnish competitive programming community for sparking my interest in algorithms and teaching me so much about them, with special thanks to Otte Heinävaara, Olli Hirviniemi, Kalle Luopajärvi, and Topi Talvitie.

In addition to Fedor, Daniel, and Michał, I would like to thank Konrad Majewski, Wojciech Nadara, and Marek Sokołowski for collaborating on the research presented in this thesis.

I thank the evaluation committee of this thesis in advance, Hans L. Bodlaender and Archontia C. Giannopoulou for assessing the thesis, and Torstein J. F. Strømme for agreeing to be the committee leader.

I then thank all of my co-authors, Matthias Bentert, Jeremias Berg, Benjamin Bergougnoux, Édouard Bonnet, Parinya Chalermsook, Clément Dallard, Pål Grønås Drange, Fedor V. Fomin, Peter Gartland, Petr A. Golovach, Thekla Hamm, Jedrzej Hodor, Matti Järvisalo, Daniel Lokshtanov, Konrad Majewski, Tomás Masarík, Martin Milanič, Wojciech Nadara, Jesper Nederlof, Ly Orgo, Pekka Parviainen, Michał Pilipczuk, Igor Razgon, Paul Saikko, Kirill Simonov, Marek Sokołowski, and Giannos Stamoulis. I also thank everyone who I have worked with but not yet published, as well as all those with who I've had insightful discussions at conferences, workshops, during research visits, and by email.

I thank everyone at the Algorithms Group and around it at the University of Bergen, including Benjamin, Boris, both Emmanuels, Erlend, Farhad, Kenneth, Lars, Magnus, Matthias, Petra, Pål, Sayan, Svein, Tanmay, William, and Wim, for all of the lunches, Friday seminars, winter schools, and beers we've had together. Special thanks to Svein Høgemo for helping with the Norwegian abstract of this thesis.

I also wish to thank all of my friends in Finland, for the fun times when I'm around and for the encouragement during my PhD, with special thanks to Vili.

I am thankful to my parents and family for all of the encouragement throughout my life and studies.

Finally, I thank Liisa for all of her love and support. Without you I wouldn't have done this.

Abstract

The treewidth of a graph describes its tree-likeness by how well it can be decomposed by small separators. It is defined as the minimum width of a tree decomposition of the graph. When a graph is given together with a tree decomposition of small width, many problems that are NP-hard in general can be solved efficiently. This motivates the problem where we are given a graph G, and our task is to either compute a tree decomposition of G of small width or to determine that the treewidth of G is large. A similar situation holds also for other width parameters of graphs, in particular, the rankwidth.

In this thesis, we introduce a new algorithmic technique for computing width parameters of graphs and the decompositions associated with them. Our method is inspired by proofs about the existence of lean tree decompositions from the graph theory literature. The new technique allows us to address several open questions from the literature on treewidth and rankwidth. Specifically, we give two new algorithms for computing treewidth, a dynamic data structure for maintaining tree decompositions, and a new algorithm for computing rankwidth.

Our first contribution, where we introduce the technique, is a 2-approximation algorithm for treewidth running in time $2^{\mathcal{O}(k)}n$, where k is the treewidth and n the number of vertices. This improves upon a previous work about 5-approximating treewidth within the same running time.

As our second contribution, we extend our technique to compute treewidth exactly in $2^{\mathcal{O}(k^2)}n^4$ time. This answers a long-standing open question in the literature about whether there exists a $2^{o(k^3)}n^{\mathcal{O}(1)}$ time exact algorithm for treewidth.

Our third contribution is a data structure for maintaining a tree decomposition of a dynamic graph G that is updated by edge insertions and deletions. Our data structure maintains a tree decomposition of width at most $6k+5$, given a promise that the treewidth of G is at most k. The amortized update time is $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$, which is subpolynomial in n for a fixed k. This partially answers an open question from the literature.

As the fourth contribution of this thesis, we extend our technique from treewidth to rankwidth, giving a $2^{2^{\mathcal{O}(k)}}n^2$ time 2-approximation algorithm for rankwidth. The question of whether rankwidth could be approximated in subcubic time in n, for a fixed k, was another open question in the literature.

Abstract in Norwegian

Trebredden til en graf beskriver dens likhet med trær ved hvor godt den kan dekomponeres ved hjelp av små separatorer. Den er definert som minimumsbredden av en tre-dekomponering av grafen. Når en graf er gitt sammen med en tre-dekomponering av liten bredde, kan mange problemer som generelt er NP-harde løses effektivt. Dette gir grunn for å granske problemet hvor vi er gitt en graf G, og vår oppgave er å enten beregne en tredekomponering av G med liten bredde eller å avgjøre at trebredden til G er stor. En lignende situasjon gjelder også for andre breddeparametre av grafer, spesielt rangbredden.

I denne avhandlingen introduserer vi en ny algoritmisk teknikk for å beregne breddeparametre av grafer og dekomponeringene assosiert med dem. Metoden vår er inspirert av bevis fra grafteorilitteraturen om eksistensen av slanke tredekomponeringer. Den nye teknikken gjør oss i stand til å takle flere åpne spørsmål fra litteraturen om trebredde og rangbredde. Især gir vi to nye algoritmer for å beregne trebredde, en dynamisk datastruktur for å vedlikeholde tre-dekomponeringer, og en ny algoritme for å beregne rangbredde.

Vårt første bidrag, der vi introduserer teknikken, er en 2-tilnærmingsalgoritme for trebredde med kjøretid $2^{\mathcal{O}(k)}n$, der k er trebredden og n er antall noder. Dette er en forbedring over tidligere arbeid, som gir en 5-tilnærming av trebredde innen samme kjøretid.

I vårt andre bidrag utvider vi teknikken vår for å eksakt beregne trebredde i $2^{\mathcal{O}(k^2)}n^4$ tid. Dette gir svar på et gammelt åpent spørsmål i litteraturen om hvorvidt det eksisterer en $2^{o(k^3)}n^{\mathcal{O}(1)}$-tid eksakt algoritme for trebredde.

Vårt tredje bidrag er en datastruktur for å vedlikeholde en tredekomponering av en dynamisk graf G som oppdateres ved innsetting og sletting av kanter. Datastrukturen vedlikeholder en tredekomponering med bredde på maks $6k + 5$, gitt et løfte om at trebredden til G er maks k. Den amortiserte oppdateringstiden er $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$, som

er subpolynomisk i n for fastliggende k. Dette gir delvis svar på et åpent spørsmål fra litteraturen.

I det fjerde bidraget av denne avhandlingen, utvider vi teknikken vår fra trebredde til rangbredde, og gir en 2-tilnærmingsalgoritme for rangbredde med kjøretid $2^{2^{\mathcal{O}(k)}}n^2$. Spørsmålet om hvorvidt rangbredde kunne tilnærmes i subkubisk tid i n for fastliggende k, var et annet åpent spørsmål i litteraturen.

List of publications

This thesis is based on the following four published articles and contains text copied verbatim from them. In particular, Chapters 4, 5, 6, and 7 are based on Articles 1, 2, 3, and 4, respectively. The order of authors is alphabetical, as is customary in theoretical computer science.

1. Tuukka Korhonen. *A single-exponential time 2-approximation algorithm for treewidth.* In Proceedings of the 62nd Annual Symposium on Foundations of Computer Science (FOCS 2021), pages 184–192. IEEE. To appear in SIAM Journal on Computing. Full version: `https://arxiv.org/abs/2104.07463`.

2. Tuukka Korhonen and Daniel Lokshtanov. *An improved parameterized algorithm for treewidth.* In Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023), pages 528–541. ACM. Full version: `https://arxiv.org/abs/2211.07154`.

3. Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. *Dynamic treewidth.* In Proceedings of the 64th Annual Symposium on Foundations of Computer Science (FOCS 2023), pages 1734–1744. IEEE. Full version: `https://arxiv.org/abs/2304.01744`.

4. Fedor V. Fomin and Tuukka Korhonen. *Fast FPT-approximation of branchwidth.* In Proceedings of the 54th Annual ACM Symposium on Theory of Computing (STOC 2022), pages 886–899. ACM. Full version: `https://arxiv.org/abs/2111.03492`.

We remark that this is not the complete list of the author's publications. The complete list can be found online on the personal page of the author (`https://tuukkakorhonen.com/`) and in services such as dblp and Google Scholar.

Contents

Part I

Introduction and preliminaries

Chapter 1

Introduction

Treewidth and tree decompositions, defined by Robertson and Seymour [1986a] and introduced independently under various names by Bertele and Brioschi [1973], Halin [1976], and Arnborg and Proskurowski [1989], have been influential in the fields of algorithms and graph theory over the last 40 years. The *treewidth* of a graph measures how well the graph can be decomposed by small separators. It describes the tree-likeness of the graph in the sense that trees can be decomposed by separators of cardinality 1, while graphs of treewidth k can be decomposed by separators of cardinality k. More precisely, the treewidth of a graph is defined as the minimum width of a *tree decomposition* of it, which is a certain way of arranging the vertices of the graph into a tree-shaped structure. See Figure 1.1 for an illustration of a graph and a tree decomposition of it.

Formally, a tree decomposition of a graph G is a pair (T, bag), where T is a tree and bag is a function $\mathsf{bag}\colon V(T) \to 2^{V(G)}$ that maps nodes of T to sets of vertices of G called *bags* so that (1) for every edge $uv \in E(G)$, there exists a node $t \in V(T)$ with $\{u, v\} \subseteq \mathsf{bag}(t)$, and (2) for every vertex $v \in V(G)$, the set of nodes of T whose bags contain v forms a non-empty connected subtree of T. The *width* of (T, bag) is the maximum size of a bag minus one[1], and the treewidth of G is the minimum width of a tree decomposition of G.

The main significance of treewidth in algorithms is that many algorithmic problems that are hard in general can be solved efficiently if the input graph has small treewidth [Arnborg and Proskurowski, 1989; Courcelle, 1990]. Many classical NP-hard graph problems, such as 3-coloring, maximum independent set, and Hamiltonicity, admit algorithms running in time $2^{\mathcal{O}(k)}n$, where n is the number of vertices and k is the width of a given tree decomposition of the input graph [Bodlaender, 1988; Bodlaender et al., 2015; Telle and Proskurowski, 1997]. These algorithms work by bottom-up dynamic programming on

[1]The purpose of "minus one" is to make the treewidth of a tree to be at most 1.

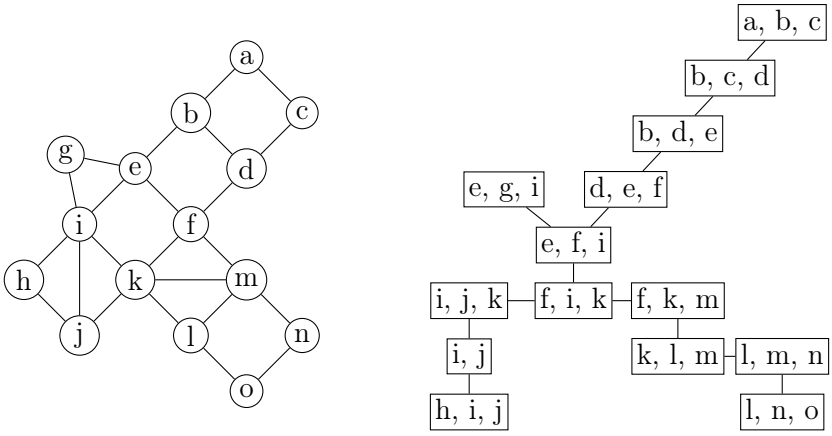

Figure 1.1: A graph G whose vertices $V(G)$ are indexed by the letters $a, \ldots, o$ (left), and a tree decomposition of G of width $3 - 1 = 2$ (right). The tree decomposition shows that the treewidth of G is at most 2. The treewidth of G is in fact exactly 2, because only forests have treewidth less than 2.

the given tree decomposition, similar to dynamic programming algorithms for solving problems on trees. The celebrated meta-theorem of Courcelle [1990] (see also [Arnborg et al., 1991; Borie et al., 1992]) states that such dynamic programming algorithms, running in time $f(k) \cdot n$ for some function f, exist for all graph problems expressible in the so-called "counting monadic second-order logic" (CMSO_2).[2]

The applications of treewidth are not limited only to graph problems, but extend to various settings by associating the input with a graph that describes its structure in an appropriate way [Chekuri and Rajaraman, 2000; Dechter and Pearl, 1989; Fomin et al., 2018; Lauritzen and Spiegelhalter, 1988; Markov and Shi, 2008; Thorup, 1998]. Furthermore, algorithms for graphs of small treewidth are not only useful when solving problems for inputs of small treewidth, but are frequently used as subroutines for solving problems on planar graphs [Baker, 1994; Bodlaender et al., 2016b], minor-free graphs [Demaine et al., 2005a; Flum and Grohe, 2001], and even on general graphs [Robertson and Seymour, 1995]. We will review different applications of treewidth in Section 3.3.

What makes treewidth fundamental is not only its applications in algorithm design, but also the fact that in many contexts treewidth describes exactly the boundary between easy and hard inputs: Inputs with small treewidth can be solved efficiently, while all inputs with large treewidth are hard [Grohe et al., 2001; Kreutzer and Tazari, 2010b; Robertson and Seymour, 1986b; Seese, 1991].[3] Even in situations where this is not the case, treewidth serves as an important base case upon which alternative structural parameters are designed.

[2] We will define CMSO_2 in Subsection 3.3.2.

[3] Of course, stating that all inputs with large treewidth are "hard" usually comes with some technical assumptions. We discuss these results in more detail in Subsection 3.3.4.

One of these alternative structural parameters is the *rankwidth* of a graph, introduced by Oum and Seymour [2006] in order to approximate a parameter called *cliquewidth*, introduced earlier by Courcelle et al. [1993] (more explicitly in [Courcelle, 1995]). Cliquewidth and rankwidth are more general parameters than treewidth in the sense that they can be upper bounded by functions of treewidth. In particular, graphs of treewidth k have cliquewidth at most $3 \cdot 2^{k-1}$ and rankwidth at most $k + 1$ [Corneil and Rotics, 2005; Oum, 2008a]. Unlike treewidth, cliquewidth and rankwidth can be small also on dense graphs, for example, their values on complete graphs and complete bipartite graphs are at most 2, while treewidth grows unboundedly. Many algorithmic results have been generalized from the setting of treewidth to that of cliquewidth and rankwidth [Courcelle et al., 2000, 2001; Kobler and Rotics, 2003], and indeed the original motivation behind cliquewidth was to capture the "tree-like" graphs from the viewpoint of a variant of CMSO_2 called CMSO_1 [Courcelle, 1995].

Most of the algorithms designed for graphs of small treewidth, or for graphs of small rankwidth or cliquewidth, need the graph to be given together with a decomposition of small width. In particular, the running times of the algorithms are not parameterized by the widths of the input graphs, but instead by the widths of the given decompositions. This makes the problem of computing small-width decompositions central in the algorithmic theory of graph width parameters.

Contributions and outline of the thesis

In this thesis, we make multiple contributions to algorithms for computing graph width parameters and associated decompositions of graphs, solving several open problems from the literature. Our contributions are based on a new technique for computing graph width parameters that we introduce in this thesis. Roughly speaking, our new technique allows to improve decompositions by successive "local improvements", which can be implemented efficiently with the use of appropriate data structures. The graph-theoretic aspects of this technique are inspired by proofs about the existence of so-called "lean tree decompositions" by Thomas [1990] and Bellenbaum and Diestel [2002], although significantly generalized in this thesis. The algorithmic aspects of the technique are completely new.

We introduce our technique in Chapter 4 by giving a new parameterized 2-approximation algorithm for treewidth. In Chapter 5 we further generalize this technique, obtaining parameterized exact and $(1 + \varepsilon)$-approximation algorithms for computing treewidth. We discuss the literature of treewidth computation and our algorithms for computing treewidth in Section 1.1 of this introduction chapter. In Chapter 6 we apply our technique to design a data structure for efficiently maintaining small-width tree decompositions

of dynamic graphs. This dynamic algorithm and the related literature are discussed in Section 1.2. Finally, in Chapter 7 we give a parameterized algorithm for approximating rankwidth and cliquewidth. This, together with the literature on rankwidth and cliquewidth is discussed in Section 1.3. Further introductory material related to the topics of this thesis will be presented in Chapters 2 and 3. In Chapter 2 we present formal definitions and preliminary results. We survey some introductory topics in detail in Chapter 3, reviewing two previous algorithms for computing treewidth, applications of graph width parameters, and the proof of [Bellenbaum and Diestel, 2002; Thomas, 1990] that inspired our new technique. This thesis is concluded in Chapter 8 with open problems and directions for future research.

1.1 Computing treewidth

Due to its importance, the problem of computing treewidth and tree decompositions has a long and rich history. We summarize it in Table 1.1 and in words below.

Before diving into the literature, let us remark that for the purpose of using small-width tree decompositions in applications, it is not necessary to compute an optimum-width tree decomposition, but an approximately optimal decomposition is also suitable. For example, if we are interested in using an algorithm that has a running time of $2^{\mathcal{O}(k)}n$ when given a tree decomposition of width k, it still attains the same running time bound even when given a tree decomposition of width $5k$. However, the algorithm becomes slower as the width grows, so minimizing the approximation ratio is a desirable goal. Furthermore, as the algorithms making use of tree decompositions often run in time at least exponential in the width k, it makes sense to allow similar running times also for algorithms computing tree decompositions.

When Robertson and Seymour [1986a] introduced treewidth, they showed that for every fixed constant k there exists a polynomial-time algorithm for testing if a given graph has treewidth at most k. Curiously, their proof yielded only the existence of an algorithm, without actually constructing it.[4] However, concurrently with them, Arnborg et al. [1987] gave an algorithm that computes a tree decomposition of width k, if one exists, in time $\mathcal{O}(n^{k+2})$, where n is the number of vertices. Arnborg et al. also showed that for unbounded k, the problem of deciding if a given graph has treewidth at most k is NP-hard.

[4] To construct the algorithm, one needs the set of minimal graphs of treewidth $k+1$ under the graph minor relation. This set was shown to be finite by Robertson and Seymour [1990], but without any upper bound on its size. Upper bounds were given later by [Lagergren, 1998; Lagergren and Arnborg, 1991].

Reference	Appx. $\alpha(k)$	Time
Robertson and Seymour [1986a]	exact	$n^{f_1(k)}$
Arnborg et al. [1987]	exact	n^{k+2}
Robertson and Seymour [1995]	$4k+3$	$3^{3k} \cdot k^2 \cdot n^2$
Robertson and Seymour [1995]	exact	$f_2(k) \cdot n^2$
Matoušek and Thomas [1991]	$6k+5$	$k^{\mathcal{O}(k)} \cdot n \log^2 n$
Lagergren [1996]	$8k+7$	$k^{\mathcal{O}(k)} \cdot n \log^2 n$
Reed [1992]	$8k+7$	$k^{\mathcal{O}(k)} \cdot n \log n$
Bodlaender [1996]	exact	$2^{\mathcal{O}(k^3)} \cdot n$
Bodlaender et al. [1995]	$\mathcal{O}(k \log n)$	$n^{\mathcal{O}(1)}$
Amir [2010]	$4.5k$	$2^{3k} \cdot k^{3/2} \cdot n^2$
Amir [2010]	$\mathcal{O}(k \log k)$	$k \log k \cdot n^4$
Feige et al. [2008]	$\mathcal{O}(k\sqrt{\log k})$	$n^{\mathcal{O}(1)}$
Fomin et al. [2015]	exact	1.7347^n
Fomin et al. [2018]	$\mathcal{O}(k^2)$	$k^7 \cdot n \log n$
Bodlaender et al. [2016a]	$3k+4$	$2^{\mathcal{O}(k)} \cdot n \log n$
Bodlaender et al. [2016a]	$5k+4$	$2^{\mathcal{O}(k)} \cdot n$
Belbasi and Fürer [2022]	$5k+4$	$2^{7.7k} \cdot n \log n$
Belbasi and Fürer [2021]	$5k+4$	$2^{6.8k} \cdot n \log n$
Chapter 4 of this thesis	$2k+1$	$2^{\mathcal{O}(k)} \cdot n$
Chapter 5 of this thesis	exact	$2^{\mathcal{O}(k^2)} \cdot n^4$
Chapter 5 of this thesis	$(1+\varepsilon)k$	$k^{\mathcal{O}(k/\varepsilon)} \cdot n^4$

Table 1.1: Overview of algorithms for computing treewidth with running time $\mathcal{O}(\text{Time})$, where n is the number of vertices and k the treewidth. All of the algorithms listed expect the exact algorithms of [Robertson and Seymour, 1986a] and [Robertson and Seymour, 1995] either output a tree decomposition of width at most $\alpha(k)$ or determine that the treewidth of the input graph is larger than k. The exact algorithms of [Robertson and Seymour, 1986a] and [Robertson and Seymour, 1995] determine whether the treewidth of the input graph is at most k. The functions $f_1(k)$ and $f_2(k)$ are fast growing functions depending on the set of minor-minimal graphs of treewidth $k+1$. Many rows of this table are based on a similar table given by Bodlaender et al. [2016a].

In the terminology of parameterized complexity, the algorithms of Robertson and Seymour [1986a] and Arnborg et al. [1987] are *slice-wise polynomial* (XP), meaning that their running times are polynomial for every fixed value of the parameter k. In these algorithms, the degree of the polynomial depends on k. Algorithms that run in polynomial time for every fixed k, without the degree of the polynomial depending on k, are called *fixed-parameter algorithms* (FPT algorithms).

The first FPT algorithms for treewidth were given by Robertson and Seymour [1995]. They gave an algorithm that in time $\mathcal{O}(3^{3k}k^2n^2)$ either outputs a tree decomposition of width at most $4k+3$, or determines that the treewidth of the input graph is more than k.[5] We will review this algorithm in Section 3.1. By making use of this 4-approximation

[5]The algorithm of Robertson and Seymour [1995] was actually given as a 3-approximation algorithm for a parameter called "branchwidth" which is closely related to treewidth and will be discussed in

algorithm, Robertson and Seymour [1995] also showed that for every fixed constant k there exists a $\mathcal{O}(n^2)$ time algorithm for testing if the treewidth of a given graph is at most k.[6]

In the start of the 1990s, Matoušek and Thomas [1991], Lagergren [1996], and Reed [1992] built upon the ideas of the 4-approximation algorithm of Robertson and Seymour [1995] to improve its running time to be near-linear as a function of n. The algorithms of Matoušek and Thomas [1991] and Lagergren [1996] run in time $k^{\mathcal{O}(k)} n \log^2 n$, and output tree decompositions of width at most $6k + 5$ and $8k + 7$, respectively. The algorithm of Reed [1992] runs in time $k^{\mathcal{O}(k)} n \log n$ and outputs a tree decomposition of width at most $8k + 7$. The algorithm of Lagergren [1996] is given as a parallel algorithm with $k^{\mathcal{O}(k)} \log^3 n$ running time on $\mathcal{O}(k^2 n)$ processors.

A direct, constructive FPT algorithm for computing treewidth exactly was given by Bodlaender and Kloks [1996] and Lagergren and Arnborg [1991] (the conference versions of both appearing concurrently at ICALP 1991[7]). They showed that when given a tree decomposition of width ℓ, one can use dynamic programming to decide if treewidth is at most k in time $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)} n$, and in the affirmative case output a tree decomposition of width at most k. At the time in 1991, this implied a $2^{\mathcal{O}(k^3)} n \log^2 n$ time exact algorithm for computing an optimum-width tree decomposition, by first running the algorithm of Lagergren [1996] (first appeared in [Lagergren, 1990]) to obtain an 8-approximate tree decomposition in time $k^{\mathcal{O}(k)} n \log^2 n$, and then running the dynamic programming algorithm on this 8-approximate tree decomposition.

Bodlaender [1996] showed that instead of using a separate approximation algorithm for obtaining the tree decomposition to run dynamic programming on, one can use a clever self-reduction scheme to assume that an approximately optimal tree decomposition is always available. This led to a linear $2^{\mathcal{O}(k^3)} n$ time algorithm for computing tree decompositions of optimum width. We will review this algorithm in Section 3.2.

As for polynomial-time approximation of treewidth, Bodlaender et al. [1995] showed that techniques introduced by Leighton and Rao [1999] can be used to $\mathcal{O}(\log n)$-approximate treewidth in polynomial-time. Amir [2010] improved the approximation ratio of this algorithm (again using the techniques of Leighton and Rao [1999]) to $\mathcal{O}(\log k)$. Later, Feige et al. [2008] gave a polynomial-time $\mathcal{O}(\sqrt{\log k})$-approximation algorithm for treewidth, which remains the best approximation ratio achieved in polynomial-time. Wu et al. [2014]

Section 1.3, but in the literature it has been interpreted as a 4-approximation algorithm for treewidth by e.g. [Reed, 1992], [Kleinberg and Tardos, 2005, Chapter 10], and [Cygan et al., 2015, Chapter 7].

[6] This algorithm had the same caveat of needing the list of minor-minimal graphs of treewidth $k+1$ to construct it as discussed above. See [Bodlaender, 1994; Fellows and Langston, 1994] for further discussion and partial lifting of the constructivity issues in this algorithm.

[7] The conference version of [Bodlaender and Kloks, 1996] is [Bodlaender and Kloks, 1991].

showed that assuming the "Small Set Expansion" conjecture, approximating treewidth is NP-hard for any constant approximation ratio. The Small Set Expansion conjecture was introduced by Raghavendra and Steurer [2010] and is a stronger version of the Unique Games conjecture of Khot [2002].

We then return to FPT algorithms for treewidth, soon getting to the contributions of this thesis. After the algorithm of Bodlaender [1996], the main question left about FPT algorithms for treewidth was about the dependence on k. In particular, most of the classical dynamic programming algorithms run in time $2^{\mathcal{O}(k)}n$ when given a tree decomposition of width k, but no matching algorithm for constant-factor approximating treewidth was known. This situation was remedied by Bodlaender et al. [2016a], who gave a $2^{\mathcal{O}(k)}n$ time 5-approximation algorithm for treewidth. They used techniques that combine ideas of Reed [1992] with additional data structures based on logarithmic-depth tree decompositions [Bodlaender and Hagerup, 1998], the dynamic programming of [Bodlaender and Kloks, 1996; Lagergren and Arnborg, 1991], and the self-reduction scheme of Bodlaender [1996]. With these techniques, they also obtained a $2^{\mathcal{O}(k)}n \log n$ time 3-approximation algorithm for treewidth.

As the first contribution of this thesis, we give in Chapter 4 an algorithm that improves upon both of the algorithms given by Bodlaender et al. [2016a].

Theorem 1.1. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k)}n$ either outputs a tree decomposition of G of width at most $2k+1$ or determines that the treewidth of G is larger than k.*

Perhaps more interesting than the theorem statement are the techniques behind the algorithm of Theorem 1.1. The previous approximation algorithms for treewidth running in time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ are based on constructing a tree decomposition in a top-down manner, as pioneered by Robertson and Seymour [1995]. The algorithm of Theorem 1.1 is instead based on improving a tree decomposition by iterative "local improvements", inspired by the proofs of [Bellenbaum and Diestel, 2002; Thomas, 1990].

Compared to the algorithm of Bodlaender et al. [2016a], the algorithm of Theorem 1.1 has also faster running time as a function of k, being roughly $2^{10.8k}$ as opposed to more than 2^{90k}. It is also arguably simpler, being significantly shorter to describe and avoiding tricks such as algorithm selection based on the relation of n and k. Most significantly, the new techniques we introduce for Theorem 1.1 lead, after further development, to the other contributions of this thesis.

It was asked by Downey and Fellows in their monograph [Downey and Fellows, 1999, Chapter 6.3] whether the dependence on k in Bodlaender's algorithm for computing

treewidth exactly could be improved from $2^{\mathcal{O}(k^3)}$ to $2^{\mathcal{O}(k)}$. Later, Telle [Bodlaender et al., 2006, Problem 2.7.1] asked the less ambitious question of whether there is any FPT algorithm for treewidth whose running time as a function of k is better than $2^{\mathcal{O}(k^3)}$. The problem of obtaining a $2^{o(k^3)}n^{\mathcal{O}(1)}$ time algorithm for treewidth was also asked by Bodlaender et al. [2016a] and called a "long-standing open problem" by Bodlaender et al. [2023]. We resolve this problem in Chapter 5 by further developing the techniques introduced for Theorem 1.1.

Theorem 1.2. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k^2)}n^4$ either outputs a tree decomposition of G of width at most k or determines that the treewidth of G is larger than k. Moreover, the algorithm runs in space $n^{\mathcal{O}(1)}$.*

An interesting feature of the algorithm of Theorem 1.2 is that it does not use dynamic programming on tree decompositions in any way, and in particular, runs in space polynomial in n. The previous parameterized algorithms for computing treewidth exactly are based on dynamic programming and use space at least exponential in k.

The proof of Theorem 1.2 has two parts. First, we introduce a problem called "Subset Treewidth", and show that by solving Subset Treewidth, one can use "local improvements", akin to those of Theorem 1.1 but significantly generalized, to improve tree decompositions until they attain the optimum width. In the second part of the proof we give an algorithm for solving the Subset Treewidth problem by branching, therefore avoiding dynamic programming.

The techniques we introduce for Theorem 1.2 allow trading off the width of the resulting tree decomposition for the running time, and result also in the following FPT approximation scheme for treewidth, presented also in Chapter 5.

Theorem 1.3. *There is an algorithm that, given an n-vertex graph G, an integer k, and a rational ε with $0 < \varepsilon < 1$, in time $k^{\mathcal{O}(k/\varepsilon)}n^4$ either outputs a tree decomposition of G of width at most $(1+\varepsilon)k$ or determines that the treewidth of G is larger than k. Moreover, the algorithm runs in space $n^{\mathcal{O}(1)}$.*

In fact, the algorithm of Theorem 1.3 is the more natural and simpler one of the two algorithms of Theorems 1.2 and 1.3, and results in a $2^{\mathcal{O}(k^2 \log k)}n^4$ time exact algorithm by setting $\varepsilon = 1/(k+1)$, which already resolves the open problem of computing treewidth in $2^{o(k^3)}n^{\mathcal{O}(1)}$ time.

We will discuss future directions and open problems related to computing treewidth in Chapter 8.

1.2 Dynamic treewidth

The field of dynamic graph algorithms studies whether solutions to graph problems can be maintained, under updates to graphs, faster than recomputing from scratch on every update. Dynamic algorithms are useful not only for processing the constantly changing real-world data, but also as subroutines for designing faster static algorithms [Sleator and Tarjan, 1981].

We consider the following dynamic treewidth problem. We have a dynamic graph G that is updated by insertions and deletions of edges, one edge at a time. We are furthermore given at the initialization a parameter k and a promise that the treewidth of G will never exceed k. The task is to maintain a tree decomposition of G of width bounded by a function of k. Ideally, we would also like to maintain arbitrary dynamic programming procedures on the tree decomposition, in order to maintain solutions to various problems about G.

A common formalization used in the literature for the feature of maintaining any dynamic programming procedure on a tree decomposition is that of maintaining whether G satisfies a fixed property φ expressible in CMSO_2. By the theorem of Courcelle [1990], such properties of G can be decided by dynamic programming on a tree decomposition of G, and in a certain sense, all properties decided by finite-state dynamic programming on tree decompositions can be expressed in CMSO_2 [Bojanczyk and Pilipczuk, 2016]. Examples of CMSO_2-expressible properties include 3-colorability and Hamiltonicity.

A trivial solution to the dynamic treewidth problem is given by recomputing the tree decomposition after every update with the linear-time algorithm of Bodlaender [1996] (or of Bodlaender et al. [2016a], or of Theorem 1.1). Then, the tree decomposition is updated in $\mathcal{O}(n)$ time for fixed k, and we can run Courcelle's dynamic programming on it in linear time after each update. In this light, the interesting challenge of dynamic treewidth is to develop algorithms that run in time sublinear in n per update, or ideally, in time $\mathcal{O}(\log n)$ per update, for fixed k, matching the running times of various data structures for dynamic trees [Alstrup et al., 2005; Frederickson, 1997; Sleator and Tarjan, 1981].

This question about dynamic algorithms for treewidth was first asked by Bodlaender [1993], and then repeated by Dvořák et al. [2014], Alman et al. [2020], Chen et al. [2021], and Majewski et al. [2023]. Before going to our contribution, let us review the existing literature about the dynamic treewidth problem.

Bodlaender [1993] showed that for dynamic graphs of treewidth at most 2, tree decompositions of width at most 11 can be maintained with worst-case update time $\mathcal{O}(\log n)$.

The data structure also supports maintaining whether the graph satisfies any fixed CMSO_2-expressible property. The approach of Bodlaender relies on a specific structural theorem for graphs of treewidth 2, which does not carry over to larger values of treewidth. Bodlaender [1993] also observed that for every fixed $k > 2$, an update time of $\mathcal{O}(\log n)$ can be achieved in the setting when no edge insertions are allowed. But this setting is significantly simpler, as no rebuilding of the tree decomposition is necessary. Independently of Bodlaender, Cohen et al. [1993][8] gave a $\mathcal{O}(\log^2 n)$ worst-case update time dynamic algorithm for maintaining tree decompositions of graphs of treewidth at most 2, and a $\mathcal{O}(\log n)$ update time dynamic algorithm for graphs of treewidth at most 3 in the setting when no edge deletions are allowed.

After that, Frederickson [1998] and Hagerup [2000] studied the dynamic treewidth problem for arbitrary fixed treewidth bound k, but in settings where either the updates to the tree decomposition are supplied in the input, or the tree decomposition is never updated. This sidesteps the main difficulty, namely, maintaining the tree decomposition itself.

With no further success in dynamic treewidth, several authors turned into more restrictive graph parameters, in particular, to the parameters "treedepth" and "feedback vertex number" that are both always at least treewidth minus one. Dvořák et al. [2014] showed that CMSO_2-expressible properties can be maintained on dynamic graphs of bounded treedepth with constant (depending on treedepth and the property) time per update. The running time of their data structure (the dependence on treedepth) was improved by Chen et al. [2021]. Building upon the work of Alman et al. [2020] on dynamic feedback vertex set, Majewski et al. [2023] showed that CMSO_2-expressible properties can be maintained in $\mathcal{O}(\log n)$ time on dynamic graphs with bounded feedback vertex number.

Let us then return to dynamic treewidth. Goranci et al. [2021] gave the first sublinear time dynamic algorithm for the general dynamic treewidth problem. They showed that tree decompositions with $n^{o(1)}$-approximately optimal width can be maintained with amortized update time $n^{o(1)}$, under the assumption that the graph has bounded maximum degree. The running time of their algorithm does not depend on the treewidth bound k. However, because the width of the maintained tree decomposition can be superlogarithmic in n even for graphs of bounded treewidth, their algorithm is not very useful in applications of treewidth for solving NP-hard problems by dynamic programming on tree decompositions. In particular, running even the simplest dynamic programming algorithms on a tree decomposition maintained by their algorithm results in a superpolynomial running time in n even when k is fixed.

[8] The author was not able to access the article [Cohen et al., 1993], and therefore our description of it is based on that of Bodlaender [1993].

In Chapter 6 of this thesis, we give the first solution for dynamic treewidth with amortized update time sublinear in n for fixed k that maintains a tree decomposition whose width is bounded by a function of k. In fact, the amortized update time of our dynamic treewidth data structure is subpolynomial in n ($n^{o(1)}$) for any fixed treewidth bound k.

Theorem 1.4. *There is a data structure that is initialized with an initially edgeless n-vertex dynamic graph G and a parameter k. The data structure supports updating G by edge insertions and deletions, and maintains a tree decomposition of G of width at most $6k+5$ whenever the treewidth of G is at most k. When the treewidth of G is more than k, the data structure contains a marker* "Treewidth too large"*. The amortized initialization time is $2^{k^{\mathcal{O}(1)}}n$ and the amortized update time is $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$.*

Moreover, the data structure can be provided a CMSO_2 *sentence φ upon initialization, in which case it maintains whether φ is true in G whenever the marker* "Treewidth too large" *is not present. In this case, the amortized initialization time is $f(k,\varphi)\cdot n$ and the amortized update time is $f(k,\varphi)\cdot 2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$, where f is a computable function.*

A few remarks are in order. First, we note that the update time function $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$ can be rewritten as $2^{2^{k^{\mathcal{O}(1)}}}\cdot 2^{\sqrt{\log n}\log\log n}$ if one wishes it to have a more traditional FPT appearance. Second, in the statement of Theorem 1.4 we formalize the feature of maintaining dynamic programming by maintaining CMSO_2-expressible properties, but in Chapter 6 we also give a framework of "prefix-rebuilding updates" that allows to plug in any reasonable dynamic programming procedure on tree decompositions to the data structure. Therefore, Theorem 1.4 directly extends to, for example, maintaining the cardinality of the maximum independent set of G, and in that case, the running time overhead over the basic maintenance of the decomposition is only $2^{\mathcal{O}(k)}$ instead of $f(k,\varphi)$. We also remark that the data structure of Theorem 1.4 persists even when the treewidth of G exceeds k, in that case displaying the "Treewidth too large" marker, although this is achieved by a standard trick of Eppstein et al. [1996].

The techniques in the proof of Theorem 1.4 build upon and extend the techniques introduced for proving Theorems 1.1 to 1.3. In particular, at the heart of the dynamic algorithm is a "refinement operation" that is used to improve and re-arrange the maintained tree decomposition efficiently. It is used for controlling both the width and the depth of the decomposition, of which the latter is ultimately is the key for achieving the subpolynomial in n running time. The refinement operation extends the general version of the "local improvement" method introduced for Theorems 1.2 and 1.3, but also uses techniques from the proof of Theorem 1.1 for its efficient implementation. In addition to these techniques, we also use several other techniques from the literature of treewidth computing

as ingredients, for example the results of [Bodlaender and Hagerup, 1998; Bodlaender and Kloks, 1996; Bojańczyk and Pilipczuk, 2022].

We will discuss (potential) applications of Theorem 1.4 and future directions related to dynamic treewidth in Chapter 8.

1.3 Rankwidth, branchwidth, and cliquewidth

We then turn to graph width parameters other than treewidth, namely, rankwidth, branchwidth, and cliquewidth. The branchwidth of graphs was introduced by Robertson and Seymour [1991], and extended to a general framework of *branchwidth of connectivity functions* by Oum and Seymour [2006][9]. As the main application of this framework, Oum and Seymour introduced also the parameter rankwidth. Let us start by presenting the general definition of branchwidth of connectivity functions, and then work down to the concrete instantiations.

Let V be a set. A function $f\colon 2^V \to \mathbb{Z}_{\geq 0}$ from the subsets of V to non-negative integers is a *connectivity function* if (1) it is *submodular*, that is, $f(A)+f(B) \geq f(A\cap B)+f(A\cup B)$ for all $A, B \subseteq V$, (2) it is *symmetric*, that is, $f(A) = f(V \setminus A)$ for all $A \subseteq V$, and (3) $f(\emptyset) = f(V) = 0$.[10]

A *branch decomposition* of a connectivity function $f\colon 2^V \to \mathbb{Z}_{\geq 0}$ is a pair (T, λ), where T is a tree whose nodes have degree either 3 or 1, and λ is a bijection from V to the leaves of T (see Figure 1.2). Every edge xy of T can be associated with a bipartition (A, B) of V by partitioning the elements $v \in V$ based on which connected component of $T \setminus \{xy\}$ the leaf $\lambda(v)$ is. Then, the *width* of the edge xy is defined as $f(A)$, which is equal to $f(B)$ because of the symmetry of f, and the width of (T, λ) is the maximum width of an edge of T. The *branchwidth* of f is the minimum width of a branch decomposition of f.

When Oum and Seymour [2006] introduced branchwidth of connectivity functions, they showed that if the function f can be evaluated in time $\gamma(n)$, where $n = |V|$, then there is an algorithm that in time $\mathcal{O}(8^k \cdot n^7 \cdot \gamma(n) \cdot \log n)$ either outputs a branch decomposition of f of width at most $3k + 1$, or concludes that the branchwidth of f is more than k. Oum and Seymour [2007] also showed that an optimum-width branch decomposition of f can be computed in time $\mathcal{O}(n^{8k+6} \cdot \gamma(n) \cdot \log n)$.

[9] Branchwidth of connectivity functions could be argued to already be implicitly considered by Robertson and Seymour [1991].

[10] We remark that the assumptions that f is non-negative and $f(\emptyset) = f(V) = 0$ are not significant restrictions. If $f\colon 2^V \to \mathbb{Z}$ satisfies (1) and (2), then $f - f(\emptyset)$ satisfies (1), (2), and (3), and is non-negative.

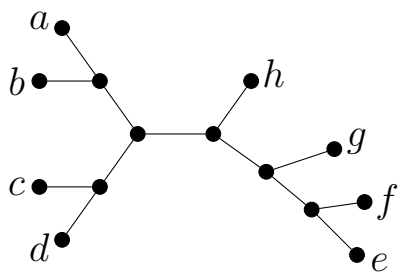

Figure 1.2: A branch decomposition of a function $f\colon 2^V \to \mathbb{Z}_{\geq 0}$, where $V = \{a, b, c, \ldots, h\}$.

In Chapter 7 we introduce a general framework for obtaining FPT 2-approximation algorithms for different instantiations of branchwidth of connectivity functions. Unlike the algorithms of Oum and Seymour, our framework does not work for arbitrary connectivity functions, but requires the branch decompositions of the connectivity function to support, informally speaking, "efficient dynamic programming" for computing certain objects. Essentially, our framework states that all of the techniques from the proof of Theorem 1.1 can be extended to the setting of branchwidth of connectivity functions, but in order to implement them efficiently, we need to be able to perform dynamic programming on the branch decomposition.

The most significant applications of both our framework and the algorithm of Oum and Seymour [2006] are about computing rankwidth, so let us next define rankwidth and review literature about it.

1.3.1 Computing rankwidth

For a graph G and a set of vertices $A \subseteq V(G)$, we define $M_G(A)$ to be the $|A| \times |V(G) \setminus A|$ matrix that describes the edges of G between A and $V(G) \setminus A$ by zeros and ones. Then, denote by $\mathsf{cutrk}_G(A)$ the rank of $M_G(A)$ over the binary field $\mathsf{GF}(2)$. Oum and Seymour [2006] showed that the function $\mathsf{cutrk}_G\colon 2^{V(G)} \to \mathbb{Z}_{\geq 0}$ is a connectivity function, and defined the *rankwidth* of G as the branchwidth of cutrk_G and a *rank decomposition* of G as a branch decomposition of cutrk_G.

Rankwidth was introduced by Oum and Seymour [2006] to approximate a width parameter called *cliquewidth*, which was introduced by Courcelle et al. [1993] in their study of logic and automata on graphs, and defined in its present form by Courcelle [1995]. We postpone the formal definition of cliquewidth to Subsection 2.3.5, but for this discussion it suffices to know that Oum and Seymour [2006] showed that rankwidth (rw) and cliquewidth

(cw) are functionally equivalent in the sense that $\mathsf{rw}(G) \leq \mathsf{cw}(G) \leq 2^{\mathsf{rw}(G)+1} - 1$ for all graphs G. A good, and correct up to a constant factor, way of thinking about cliquewidth is that it is like rankwidth, but instead of measuring the rank of $M_G(A)$ it simply counts the number of different rows and columns of $M_G(A)$.[11] A parameter closely related to cliquewidth, called *NLC-width*, was also investigated by Wanke [1994].

The motivation behind cliquewidth is that it is a generalization of treewidth that is suitable also for dense graphs. In particular, graphs of treewidth k have cliquewidth at most $3 \cdot 2^{k-1}$ [Corneil and Rotics, 2005], but there exists many classes of dense graphs that have cliquewidth bounded by a constant but unbounded treewidth, for example, complete graphs and complete bipartite graphs, and more generally cographs [Courcelle and Olariu, 2000]. Graphs of treewidth k have at most kn edges, so dense graphs can never have small treewidth.

Courcelle et al. [2000] showed that various dynamic programming algorithms for graphs of small treewidth can be generalized to graphs of small cliquewidth, if the graph is given together with a suitable decomposition witnessing that its cliquewidth is at most k, called a *k-expression*. In particular, they gave algorithms working in time $f(k) \cdot n$, when given a k-expression, for solving problems such as maximum independent set, minimum dominating set, and 3-coloring.[12]

However, at the time no algorithms for computing k-expressions were known, which was the motivation of Oum and Seymour [2006] for introducing rankwidth and giving the algorithm for approximating branchwidth of connectivity functions. They applied this to give an algorithm for approximating rankwidth, which they in turn applied for approximating cliquewidth. In particular, using their algorithm for branchwidth of connectivity functions, Oum and Seymour [2006] obtained a $\mathcal{O}(8^k n^9 \log n)$ time algorithm for computing a rank decomposition of width at most $3k+1$ or determining that $\mathsf{rw}(G) > k$. By a constructive version of the inequalities $\mathsf{rw}(G) \leq \mathsf{cw}(G) \leq 2^{\mathsf{rw}(G)+1} - 1$, this implied an algorithm that within the same running time either computes a $(2^{3k+2} - 1)$-expression, or determines that $\mathsf{cw}(G) > k$.

The result of Oum and Seymour [2006] implied $f(k) \cdot n^9 \log n$ time algorithms for solving problems on graphs of cliquewidth k even when a k-expression is not given. Not long after, Oum [2008b] improved this by giving $\mathcal{O}(8^k n^4)$ time and $f(k) \cdot n^3$ time 3-approximation algorithms for rankwidth (where $f(k)$ is a huge but computable function). The latter of these algorithms combines ideas of Oum and Seymour with ideas that Hliněný [2005] used for approximating branchwidth of matroids. Courcelle and Oum [2007] gave a

[11] A cut function defined like this is not submodular, which is the motivation for defining rankwidth.
[12] The graph can have more edges than $f(k) \cdot n$, but it is described succinctly only by the k-expression.

Reference	Appx. $\alpha(k)$	Time
Oum and Seymour [2006]	$3k+1$	$8^k \cdot n^9 \cdot \log n$
Oum and Seymour [2007]	exact	$n^{8k+12} \cdot \log n$
Oum [2008b]	$3k+1$	$8^k \cdot n^4$
Oum [2008b]	$3k-1$	$f_1(k) \cdot n^3$
Courcelle and Oum [2007]	exact	$f_2(k) \cdot n^3$
Hliněný and Oum [2008]	exact	$f_3(k) \cdot n^3$
Jeong et al. [2021]	exact	$f_4(k) \cdot n^3$
Chapter 7 of this thesis	$2k$	$2^{2^{\mathcal{O}(k)}} \cdot n^2$
Chapter 7 of this thesis	exact	$f_5(k) \cdot n^2$
Korhonen and Sokołowski [2024]	exact	$f_6(k) \cdot n \cdot 2^{\sqrt{\log n} \log\log n} + \mathcal{O}(m)$

Table 1.2: Overview of algorithms for computing rankwidth. Here n is the number of vertices, m is the number of edges, and k is the rankwidth of the input graph. All of the algorithms, except the algorithm of Courcelle and Oum [2007], either output in time $\mathcal{O}(\text{Time})$ a rank decomposition of width at most $\alpha(k)$ or determine that the rankwidth is more than k. The algorithm of Courcelle and Oum [2007] only determines if the rankwidth is at most k. All of the functions $f_i(k)$ are at least double-exponential but computable. The algorithm of [Korhonen and Sokołowski, 2024] will be discussed in Chapter 8.

$f(k) \cdot n^3$ time algorithm for computing rankwidth exactly, which however did not output a decomposition. This caveat was remedied by Hliněný and Oum [2008]. Much later, Jeong et al. [2021] gave an alternative $f(k) \cdot n^3$ time algorithm for computing optimum-width rank decompositions. Their algorithm works in fact for computing branchwidth of any connectivity function that can be described in a certain linear-algebraic way, and can be seen as a generalization of the algorithm of Bodlaender and Kloks [1996] to that setting. Algorithms for computing rankwidth are summarized in Table 1.2.

It was asked as an open problem by [Oum, 2017, Question 3] whether there exists an algorithm with functions $f(k)$, $g(k)$, and a constant $c < 3$ that finds a rank decomposition of width at most $f(k)$ or determines that the rankwidth of the input graph is more than k in time $g(k) \cdot n^c$. As the main application of our framework for FPT 2-approximation algorithms for branchwidth of connectivity functions, we solve this problem in Chapter 7.

Theorem 1.5. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{2^{\mathcal{O}(k)}} n^2$ either outputs a rank decomposition of G of width at most $2k$ or determines that the rankwidth of G is larger than k.*

Our algorithm can be combined with the algorithm of Oum and Seymour [2006] for translating rank decompositions into k-expressions, to output within the same running time, also a $(2^{2k+1}-1)$-expression for the cliquewidth of G. This improves the running times of most of the FPT algorithms parameterized by cliquewidth from $f(k) \cdot n^3$ to $f(k) \cdot n^2$.

By combining the algorithm of Theorem 1.5 with the dynamic programming algorithm for computing optimum-width rank decompositions given by Jeong et al. [2021], we obtain also the following corollary about computing rankwidth exactly.

Corollary 1.6. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $f(k) \cdot n^2$, for some computable function f, either outputs a rank decomposition of G of width at most k or determines that the rankwidth of G is larger than k.*

Graphs of rankwidth 1 are the "distance-hereditary graphs" [Oum, 2005], and for them a (non-trivial) linear-time recognition algorithm is known [Damiand et al., 2001]. However, to the best of our knowledge, already for rankwidth 2 no recognition algorithms faster than $\mathcal{O}(n^3)$ were known before Corollary 1.6.

1.3.2 Computing branchwidth of graphs

We then turn to branchwidth of graphs, which was introduced by Robertson and Seymour [1991] as an alternative for treewidth that is more suitable in some settings, in their case, that of "tangles". Let us define branchwidth of graphs as a special case of branchwidth of connectivity functions.

For a subset $A \subseteq E(G)$ of edges of a graph G, we define the *border* $\delta_G(A) \subseteq V(G)$ of A as the set of vertices that are incident both to an edge in A and to an edge in $E(G) \setminus A$. It was observed by Robertson and Seymour [1991] that the function $|\delta_G| \colon 2^{E(G)} \to \mathbb{Z}_{\geq 0}$ that maps sets of edges to cardinalities of their borders is a connectivity function. The *branchwidth* of a graph G is then defined as the branchwidth of the function $|\delta_G|$, and a *branch decomposition* of G is a branch decomposition of $|\delta_G|$. Note that here, it is the edges of G that are mapped to the leaves of the branch decomposition.

Branchwidth (bw) is closely related to treewidth (tw), as the pair of inequalities $\mathsf{bw}(G) \leq \mathsf{tw}(G) + 1 \leq \max(\frac{3}{2} \cdot \mathsf{bw}(G), 2)$ holds for all graphs G [Robertson and Seymour, 1991]. Although branchwidth and treewidth are in this sense almost equivalent, and algorithms using treewidth have certainly received more attention than algorithms using branchwidth, it has been argued that in some applications branchwidth is the more useful one of the two parameters [Cook and Seymour, 2003; Fast and Hicks, 2017; Fomin and Thilikos, 2006].

As for computing branchwidth, we note that by the aforementioned relation with treewidth (and the fact that it can be constructively implemented in $k^{\mathcal{O}(1)}n$ time), all α-approximation algorithms for treewidth can be turned into $\frac{3}{2}\alpha$-approximations for branchwidth, so all results for treewidth in Table 1.1 translate into results for branchwidth. For algorithms specifically designed for branchwidth, we mention the 3-approximation

algorithm of Robertson and Seymour [1995] with running time $2^{\mathcal{O}(k)}n^2$, and the exact algorithm of Bodlaender and Thilikos [1997] with running time $f(k) \cdot n$, for some computable function f. Also, Seymour and Thomas [1994] showed that branchwidth of planar graphs can be computed in polynomial time.

In Chapter 7, we apply our framework for branchwidth of connectivity functions to obtain the following algorithm for 2-approximating branchwidth of graphs.

Theorem 1.7. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k)}n$ either outputs a branch decomposition of G of width at most $2k$ or determines that the branchwidth of G is larger than k.*

This improves upon the 3-approximation algorithm with the same running time that follows from Theorem 1.1. Compared to results prior to this thesis, the previous best approximation ratio for branchwidth in time $2^{\mathcal{O}(k)}n$ was 7.5, which follows from the result of Bodlaender et al. [2016a].

We will discuss open problems and potential applications of our framework to further width parameters in Chapter 8.

Chapter 2

Definitions and preliminary results

In this chapter we review the formal definitions and notation used in this thesis. We also present some well-known auxiliary results about graph theory, algorithms, and width parameters, which will be used in this thesis.

2.1 Basic notation

Before going into graphs, let us fix the basic mathematical notation we will use.

We denote the set of integers by $\mathbb{Z}$, the set of non-negative integers by $\mathbb{Z}_{\geq 0}$, and the set of positive integers by $\mathbb{Z}_{\geq 1}$. For two integers a and b, $[a, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ denotes the set of integers between a and b, including a and b. The set $[a, b]$ is empty if $b < a$. For an integer n, we denote by $[n]$ the set $[1, n]$, which is empty if $n < 1$.

We fix the convention that log denotes the base-2 logarithm. We also fix that $\subset$ denotes a strict subset and $\subseteq$ a not-necessarily-strict subset. When we say that a set is maximal we mean inclusion-wise maximality, and by minimal we mean inclusion-wise minimality.

A *partition* of a set X is a set $\mathcal{C}$ of disjoint non-empty subsets of X so that $X = \bigcup_{C \in \mathcal{C}} C$. Note that in this definition the parts of a partition are not allowed to be empty, and they are not indexed. We will later introduce additional definitions to talk about partitions that allow empty sets and are indexed.

For a set X, we denote by $\binom{X}{2}$ the set of all unordered pairs of distinct elements of X. In particular, $|\binom{X}{2}| = \binom{|X|}{2} = |X|(|X| - 1)/2$, where $\binom{n}{k}$ denotes binomial coefficients.

For a function $f\colon X \to Y$ and a set $Z \subseteq X$, we denote the restriction of f to Z by $f{\restriction_Z}$. In particular, $f{\restriction_Z}$ is the function $f{\restriction_Z}\colon Z \to Y$ with $f{\restriction_Z}(x) = f(x)$ for all $x \in Z$.

2.2 Graphs

We denote the set of vertices of a graph G by $V(G)$ and the set of edges by $E(G)$. The edges of a graph are unordered pairs of distinct vertices, i.e., $E(G) \subseteq \binom{V(G)}{2}$. In particular, all graphs in this thesis are simple, i.e., are undirected, have no parallel edges, and no self-loops. We say that vertices $u, v \in V(G)$ are *adjacent* if $uv \in E(G)$ and a vertex $u \in V(G)$ is *incident* to an edge $e \in E(G)$ if $e = uv$ for some $v \in V(G)$.

Unless otherwise stated, we denote by n the number of vertices $|V(G)|$ and by m the number of edges $|E(G)|$ of a graph G. In the context of running times of algorithms, m means $|V(G)| + |E(G)|$ unless otherwise stated. This can also be interpreted as the assumption that input graphs do not contain isolated vertices, as then $|V(G)| + |E(G)| = \mathcal{O}(m)$. We assume that graphs are represented in the *adjacency list* format, in which we store for each vertex the list of all vertices adjacent to it. We will also use the *adjacency matrix* format in Chapter 7, in which an $n \times n$ matrix represents the adjacencies. We assume that vertices of graphs come from a totally ordered countable set, i.e., $V(G) \subset \mathbb{Z}_{\geq 1}$, and they can be manipulated in constant time, as is standard in the word RAM model.

For a vertex $v \in V(G)$, its *neighborhood* is $N_G(v) = \{u \mid uv \in E(G)\}$ and *closed neighborhood* $N_G[v] = N_G(v) \cup \{v\}$. For a set of vertices $X \subseteq V(G)$, its *neighborhood* is $N_G(X) = \bigcup_{v \in X} N_G(v) \setminus X$ and *closed neighborhood* $N_G[X] = N_G(X) \cup X$. We drop the subscript if the graph G is clear from the context. The *degree* of a vertex is the number of neighbors of it, and a graph is *subcubic* if its maximum degree is at most 3.

We denote the subgraph of G induced by a set of vertices $X \subseteq V(G)$ by $G[X]$. We also use the notation $G \setminus X = G[V(G) \setminus X]$. A *cut* of a graph G is a pair (A, B) of disjoint subsets of vertices $A, B \subseteq V(G)$ so that $A \cup B = V(G)$. The graph G is *bipartite* if it has a cut (A, B) so that the graphs $G[A]$ and $G[B]$ are edgeless, in which case we may say that (A, B) is a *bipartitioning cut* of G. When G is a graph and $A, B \subseteq V(G)$ are two disjoint sets of vertices, we denote by $G[A, B]$ the bipartite graph with the set of vertices $V(G[A, B]) = A \cup B$ and with $E(G[A, B])$ containing the edges of G with one endpoint in A and one in B.

An *independent set* in a graph G is a set of vertices $I \subseteq V(G)$ so that no two vertices in I are adjacent to each other. A *clique* is a set $W \subseteq V(G)$ so that $\binom{W}{2} \subseteq E(G)$. A *vertex cover* of G is a set $X \subseteq V(G)$ so that every edge of G is incident to at least one vertex in

X. A *matching* in G is a set of edges $M \subseteq E(G)$ so that every vertex of G is incident to at most one edge in M.

We use the convention that a *connected component* of a graph G is a maximal set of vertices $C \subseteq V(G)$ so that $G[C]$ is connected. We denote the set of connected components of G by $\mathsf{cc}(G)$.

A *path* in a graph G is a sequence $P = v_1, v_2, \ldots, v_\ell$ of distinct vertices of G so that v_i and v_{i+1} are adjacent in G for every $i \in [\ell - 1]$. The *endpoints* of P are v_1 and v_ℓ, and P may be called a v_1-v_ℓ-path. For two sets of vertices $A, B \subseteq V(G)$, the path P may be called an A-B-path if $v_1 \in A$ and $v_\ell \in B$. The *intermediate vertices* of P are $v_2, \ldots, v_{\ell-1}$. An edge is *on* the path P if it is the edge $v_i v_{i+1}$ for some $i \in [\ell - 1]$. The *distance* between vertices u and v is the least number of edges on an u-v-path, or infinite if no u-v-path exists. We denote the set of vertices on P by $V(P) = \{v_1, \ldots, v_\ell\}$.

Let G be a graph and $X \subseteq V(G)$. The graph $\mathsf{torso}_G(X)$ has vertices $V(\mathsf{torso}_G(X)) = X$ and has $uv \in E(\mathsf{torso}_G(X))$ if $u, v \in X$ and there is a path from u to v whose all internal vertices (if any) are in $V(G) \setminus X$. In particular, $\mathsf{torso}_G(X)$ is obtained from $G[X]$ by making $N_G(C)$ into a clique for every component $C \in \mathsf{cc}(G \setminus X)$. We may omit G from the subscript if it is clear from the context.

The *contraction* of an edge uv of a graph G means the operation that replaces the vertices u and v by a new vertex w so that w is adjacent to every vertex of G to which u or v were adjacent to (except for u and v themselves). All graphs in this thesis are simple, so contraction does not create self-loops or parallel edges. If v is a degree-2 vertex of a graph G, then *suppressing* v means the operation that adds an edge between the two neighbors of v (if it does not already exist) and then deletes v. Note that the graph resulting from suppressing v is isomorphic to the graph resulting from contracting either of the edges incident to v. *Subdividing* an edge uv means adding a vertex w and edges wu and wv, and then removing the edge uv. A graph H is a *minor* of a graph G if H is obtained from G by vertex deletions, edge deletions, and edge contractions.

We sometimes say that G *contains* H as a minor if H is a minor of G. We say that G is *H-minor-free* or *excludes* H as a minor if G does not contain H as a minor. A class of graphs is *H-minor-free* if all graphs in it are H-minor-free, and simply *minor-free* if there exists a graph H so that the class is H-minor-free. Slightly abusing the notation, we may say that a graph G is *minor-free*, meaning that the statement about G should be interpreted as concerning all graphs G in any fixed minor-free graph class.

A graph is *planar* if it can be drawn on the plane without crossings. By a theorem of Kuratowski [1930], a graph G is planar if and only if it is K_5-minor-free and $K_{3,3}$-minor-

free, where K_t denotes the complete graph on t vertices, and $K_{s,t}$ the complete bipartite graph with s vertices on one side and t on the other.

2.2.1 Separators and linkedness

Let $A, B \subseteq V(G)$ be two sets of vertices in a graph G. We say that a set $S \subseteq V(G)$ *separates* A from B if every A-B-path contains a vertex from S. In this case S is called an (A, B)-*separator*. If no A-B-path exists in G (for example, if A or B is empty), then every set $S \subseteq V(G)$, including the empty set, is an (A, B)-separator. We note that the sets A and B are allowed to intersect in this definition, but for every (A, B)-separator S it holds that $A \cap B \subseteq S$, as otherwise there would be a trivial single-vertex A-B-path avoiding S.

A *separation* of a graph G is a triple (A, S, B) of subsets of $V(G)$ so that A, S, and B are disjoint, $V(G) = A \cup S \cup B$, and there are no edges between A and B. The *order* of a separation (A, S, B) is $|S|$. Note that if $X, Y \subseteq V(G)$ and S is an (X, Y)-separator, then there exists $A, B \subseteq V(G)$ so that (A, S, B) is a separation of G, $X \subseteq A \cup S$, and $Y \subseteq S \cup B$.

For two sets of vertices $A, B \subseteq V(G)$, we denote by $\mathsf{flow}_G(A, B)$ the size of a smallest (A, B)-separator in G, dropping the subscript if it is clear form the context. Calling this flow is motivated by the following classical theorem of Menger.

Theorem 2.1 (Menger [1927])**.** *The size of a smallest (A, B)-separator is equal to the maximum size of a collection of vertex-disjoint A-B-paths.*

Here, a collection of vertex-disjoint A-B-paths means a collection $\mathcal{P} = \{P_1, \ldots, P_\ell\}$ of A-B-paths, so that every vertex of G is on at most one of the paths. In particular, also the endpoints of the paths in $\mathcal{P}$ are pairwise disjoint.

By Theorem 2.1, for any two sets $A, B \subseteq V(G)$, there exists a collection of $\mathsf{flow}(A, B)$ vertex-disjoint A-B-paths. We will use the following algorithmic version of this fact.

Theorem 2.2 (Ford and Fulkerson [1956])**.** *There is an algorithm, that given a graph G, two sets of vertices $A, B \subseteq V(G)$, and an integer k, in time $\mathcal{O}(k \cdot m)$ outputs either*

- *an (A, B)-separator of size at most k, or*
- *a collection of k vertex-disjoint A-B-paths.*

We note that even though there has been significant progress in algorithms for maximum flow since 1956 (see [Chen et al., 2022]), we only use the algorithm of Theorem 2.2 in this thesis.

We say that a set $A \subseteq V(G)$ of vertices is *linked into* a set $B \subseteq V(G)$ of vertices if $\mathsf{flow}(A, B) = |A|$. Note that this definition is asymmetric, in particular, if $|A| < |B|$, then A can be linked into B but B cannot be linked into A. We will use the following standard lemma about linkedness.

Lemma 2.3. *If S is a minimum-size (A, B)-separator, then S is linked into A and B.*

Proof. Suppose S is not linked into B, and let S' be an (S, B)-separator of size $|S'| < |S|$. Because every A-B-path contains a vertex from S, it follows that S' is also an (A, B)-separator. However, then S' contradicts the fact that S is a minimum-size (A, B)-separator. This shows that S is linked into B, and the fact that S is linked into A can be shown by interchanging the roles of A and B in the proof. □

We also say that A is *strongly linked into* B if A is linked into B, and furthermore, for every (A, B)-separator S of size $|S| = |A|$ it holds that either $S = A$ or $S = B$.

2.2.2 Trees

A *tree* is a connected acyclic graph. We usually call vertices of trees *nodes*, to underline that they are vertices of a tree. A *subtree* of a tree T is a subgraph of T that is a tree, in particular, a subgraph that is connected. The degree-1 nodes of a tree are *leaves*. A tree is *cubic* if all of its nodes except the leaves have degree 3.

Rooted trees

A *rooted tree* is a tree where one node is designated as the root. When talking about rooted trees, we use the convention that the root is not a leaf, even if it would have degree 1.

A node x of a rooted tree T is a *descendant* of a node y if the unique path from x to the root contains y. If x is a descendant of y, then y is an *ancestor* of x. Note that every node is both a descendant and an ancestor of itself. We define that x is a *strict descendant* of y if x is a descendant of y and $x \neq y$. *Strict ancestor* is defined analogously. We denote the set of descendants of a node x in a rooted tree T by $\mathsf{desc}_T(x)$. The *parent*

of a non-root node x is the unique ancestor of x that is adjacent to x, and is denoted by $\mathsf{parent}_T(x)$. We may omit the subscript if it is clear from the context. If y is the parent of x, then x is a *child* of y. A *binary tree* is a rooted tree where every node has at most two children.

When manipulating rooted trees in algorithms, we assume that in addition to the adjacency list, a pointer to the root is stored, and every node stores a pointer to its parent.

We note that although the parent of a node is defined only for rooted trees, we may call the only node adjacent to a leaf node its *parent* even in non-rooted trees.

We define a *rooted subtree* of a rooted tree T as a subtree T' of T so that $T' = T[\mathsf{desc}(x)]$ for some $x \in V(T)$. In that case, x is the root of T'. When we talk about the *subtree of T rooted at x*, we mean the rooted subtree $T[\mathsf{desc}(x)]$ of T.

We define that the *depth* of a node $x \in V(T)$ of a rooted tree T is equal to 0 if x is the root, and otherwise equal to the depth of its parent plus one, and is denoted by $\mathsf{depth}_T(x)$. The *height* of x is equal to the number of vertices on a longest path from x to a leaf, and is denoted by $\mathsf{hgt}_T(x)$. In particular, the height of each leaf is 1. Note that the function $\mathsf{depth}(x)$ takes values in $[0, |V(T)| - 1]$ and the function $\mathsf{hgt}(x)$ takes values in $[|V(T)|]$. The *height* of a rooted tree T is the height of its root, and is denoted by $\mathsf{hgt}(T)$.

A node z of a rooted tree T is the *lowest common ancestor* (LCA) of two nodes x and y of T if z is an ancestor of both x and y, and subject to that, z maximizes $\mathsf{depth}(z)$. We say that a set $X \subseteq V(T)$ of nodes is *LCA-closed* if for every two distinct nodes $x, y \in X$, the LCA of x and y is in X. The *LCA-closure* of a set X of nodes is the unique minimal superset $Y \supseteq X$ of X so that Y is LCA-closed. The following lemma shows that Y is indeed unique.

Lemma 2.4. *Let $X \subseteq V(T)$ be a set of nodes in a rooted tree T, and $L \subseteq V(T)$ the set that contains for every pair $x, y \in X$ the LCA of x and y. Then, $X \cup L$ is LCA-closed.*

Proof. Let $x, y \in X \cup L$. If x is an ancestor or a descendant of y, then their LCA is x or y, which is in $X \cup L$. Otherwise, there exists a descendant x' of x in X and a descendant y' of y in X. The LCA of x' and y' is the LCA of x and y, and by definition is in L. □

We also observe a bound on the size of the LCA-closure.

Lemma 2.5. *If $X \subseteq V(T)$ is non-empty, then its LCA-closure has size at most $2|X| - 1$.*

Proof. We prove this by induction on $|X|$. The base base of $|X| = 1$ clearly holds. Now, let $|X| \geq 2$, let Y be the LCA-closure of X, choose $x \in X$, and let Y' be the LCA-closure

of $X \setminus \{x\}$. Suppose two distinct strict ancestors a_1 and a_2 of x are in $Y \setminus Y'$, and let a_2 be a strict ancestor of a_1. Now, if a_1 is the LCA of x and $y \in X$, and a_2 is the LCA of x and $z \in X$, then a_2 is in fact the LCA of y and z, so $a_2 \in Y'$, which is a contradiction. Therefore, at most one strict ancestor of x is in $Y \setminus Y'$, so $|Y| \leq |Y'| + 2 \leq 2|X| - 1$. □

A *prefix* of a rooted tree T is a set $T_{\mathsf{pref}} \subseteq V(T)$ of nodes so that $T[T_{\mathsf{pref}}]$ is connected and contains the root of T. Note that a prefix is by definition non-empty. An *appendix* of a prefix T_{pref} is a node $a \in V(T) \setminus T_{\mathsf{pref}}$ that is not in T_{pref} but whose parent is in T_{pref}. The set of appendices of T_{pref} is denoted by $\mathsf{app}_T(T_{\mathsf{pref}})$, where the subscript may be dropped if T is clear from the context. We observe that if T is a binary tree, then $|\mathsf{app}(T_{\mathsf{pref}})| \leq |T_{\mathsf{pref}}| + 1$.

2.3 Width parameters

We review the definitions of treewidth, branchwidth of connectivity functions, branchwidth of graphs, rankwidth, and cliquewidth. We also introduce additional notation related to them and summarize well-known facts. We refer to [Cygan et al., 2015, Chapter 7] for introductory material on treewidth and to [Hliněný et al., 2008] for introductory material on the other width parameters.

2.3.1 Treewidth

Following Robertson and Seymour [1986a], we define a *tree decomposition* of a graph G to be a pair $\mathcal{T} = (T, \mathsf{bag})$, where T is a tree and $\mathsf{bag}\colon V(T) \to 2^{V(G)}$ is a function that satisfies

1. $V(G) = \bigcup_{t \in V(T)} \mathsf{bag}(t)$,
2. $E(G) \subseteq \bigcup_{t \in V(T)} \binom{\mathsf{bag}(t)}{2}$, and
3. for every $v \in V(G)$, it holds that $T[\{t \in V(T) \mid v \in \mathsf{bag}(t)\}]$ is connected.

The set $\mathsf{bag}(t)$ of a node $t \in V(T)$ is called the *bag* of t. We will call the conditions of the Items 1 to 3 the *vertex condition*, the *edge condition*, and the *connectedness condition*. The *width* of a tree decomposition is the maximum size of a bag minus one, and is denoted by $\mathsf{width}(\mathcal{T})$. The *treewidth* of a graph is the minimum width of a tree decomposition of it, and is denoted by $\mathsf{tw}(G)$.

For brevity, we will sometimes denote $|\mathcal{T}| = |V(T)|$, $V(\mathcal{T}) = V(T)$, and $E(\mathcal{T}) = E(T)$, and call nodes and edges of T nodes and edges of $\mathcal{T}$. A tree decomposition is *subcubic* if T is subcubic. For a set $X \subseteq V(T)$, we denote by $\mathsf{bags}_{\mathcal{T}}(X)$ the union $\bigcup_{t \in X} \mathsf{bag}(t)$ of bags in X, dropping the subscript if $\mathcal{T}$ is clear from the context. We may also use $\mathsf{bags}(\mathcal{T})$ to denote the union of all bags of $\mathcal{T}$ (which is equal to $V(G)$ if $\mathcal{T}$ is a tree decomposition of G). For a subset $X \subseteq V(T)$, we denote by $\mathcal{T}\restriction_X$ the pair $(T[X], \mathsf{bag}\restriction_X)$, which may or may not be a tree decomposition (recall that we use $f\restriction_X$ to denote the restriction of a function). For an edge $xy \in E(T)$ of the tree decomposition, the intersection $\mathsf{bag}(x) \cap \mathsf{bag}(y)$ is called the *adhesion* of xy.

Let us now observe some basic properties of tree decompositions that will be often used in this thesis without explicitly mentioning them. First, the connectedness condition extends to sets of vertices as follows.

Lemma 2.6. *Let (T, bag) be a tree decomposition of a graph G, and let $X \subseteq V(G)$ so that $G[X]$ is connected. Then $T[\{t \in V(T) \mid \mathsf{bag}(t) \cap X \neq \emptyset\}]$ is connected.*

Proof. Let $uv \in E(G[X])$. The set $\{t \in V(T) \mid \mathsf{bag}(t) \cap \{u, v\} \neq \emptyset\}$ is connected in T because both $\{t \in V(T) \mid u \in \mathsf{bag}(t)\}$ and $\{t \in V(T) \mid v \in \mathsf{bag}(t)\}$ are connected in T by the connectedness condition, and by the edge condition there exists a node whose bag contains both u and v. Now, the fact that $T[\{t \in V(T) \mid \mathsf{bag}(t) \cap X \neq \emptyset\}]$ is connected follows from the connectedness of $G[X]$. □

Then we prove the basic separation property of tree decompositions that any adhesion of an edge separates the vertices on different sides of it from each other.

Lemma 2.7. *Let (T, bag) be a tree decomposition of a graph G and (A, B) a cut of T so that there is exactly one edge $xy \in E(T)$ with $x \in A$ and $y \in B$. Then, the adhesion $\mathsf{bag}(x) \cap \mathsf{bag}(y)$ is a $(\mathsf{bags}(A), \mathsf{bags}(B))$-separator in G.*

Proof. Suppose not, and let $S = \mathsf{bag}(x) \cap \mathsf{bag}(y)$ and $uv \in E(G)$ be an edge of G so that $u \in \mathsf{bags}(A) \setminus S$ and $v \in \mathsf{bags}(B) \setminus S$. The subtree of bags that contain u intersects A, and therefore does not intersect y because then it would intersect also x and u would be in S. In particular, it is a subtree of $T[A]$. By a similar argument, the subtree of bags that contain v is a subtree of $T[B]$. However, this contradicts the edge condition. □

Note that Lemma 2.7 directly implies some weaker separation properties, for example, that for any node $t \in V(T)$ the set $\mathsf{bag}(t)$ is a $(\mathsf{bags}(C_1), \mathsf{bags}(C_2))$-separator for any two connected components $C_1, C_2 \in \mathsf{cc}(T \setminus \{t\})$.

We will need to bound the number of edges of graphs of small treewidth. For this, the main intermediate lemma is the following.

Lemma 2.8. *If a graph G has treewidth at most k, then G contains a vertex of degree at most k.*

Proof. Let (T, bag) be a tree decomposition of G of width at most k that minimizes the number of nodes $|V(T)|$. If T would have a leaf ℓ whose only neighbor is a node p so that $\mathsf{bag}(\ell) \subseteq \mathsf{bag}(p)$, then we could remove the node ℓ from (T, bag) and obtain a tree decomposition of G with a smaller number of nodes. Therefore, $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$ is non-empty, and therefore there exists a vertex v of G that occurs only in the bag of ℓ, implying $N(v) \subseteq \mathsf{bag}(\ell) \setminus \{v\}$, which implies that $|N(v)| \leq k$. □

The *degeneracy* of a graph G is the least integer d so that every subgraph of G has a vertex of degree at most d. Because the treewidth of any subgraph of G is at most $\mathsf{tw}(G)$, the result of Lemma 2.8 can be stated as that the degeneracy of G is at most $\mathsf{tw}(G)$.

A bound for the number of edges of G easily follows from Lemma 2.8.

Lemma 2.9. *A graph G has at most $\mathsf{tw}(G) \cdot |V(G)|$ edges.*

Proof. This follows from Lemma 2.8 and the fact that removing a vertex from G does not increase the treewidth of G. □

Another application of Lemma 2.8 is the following data structure for checking adjacencies on graphs of bounded treewidth.

Lemma 2.10. *There is a data structure, that can be initialized with a graph G in time $\mathcal{O}(\mathsf{tw}(G) \cdot n)$ and supports the query*

- $\mathsf{Adjacent}(u, v)$*: Given $u, v \in V(G)$, return whether $uv \in E(G)$ in time $\mathcal{O}(\mathsf{tw}(G))$.*

Proof. By repeatedly removing vertices with the smallest degree, let us order the vertices of G as $v_1, \ldots, v_n$ so that v_i has at most $\mathsf{tw}(G)$ neighbors v_j with $j > i$. Then, let us store for each v_i the set $N^+(v_i) \subseteq N(v_i)$ consisting of the neighbors v_j of v_i with $j > i$. This can be done in time $\mathcal{O}(\mathsf{tw}(G) \cdot n)$. Then, the $\mathsf{adjacent}(v_i, v_j)$ query for $i < j$ can be answered by checking if $v_j \in N^+(v_i)$, which takes $\mathcal{O}(\mathsf{tw}(G))$ time because $|N^+(v_i)| \leq \mathsf{tw}(G)$. □

We sometimes assume implicitly that the data structure of Lemma 2.10 is available when working with graphs of small treewidth.

We then recall the well-known property that taking minors does not increase treewidth.

Lemma 2.11. *If H is a minor of G, then $\mathsf{tw}(H) \leq \mathsf{tw}(G)$.*

Proof. For vertex and edge deletions it is obvious how to translate a tree decomposition of G into a tree decomposition of H. For edge contractions, if an edge uv is contracted into a new vertex w, then we replace each occurrence of u and v in the tree decomposition by w. □

Rooted tree decompositions

Often it will be convenient to assume that the tree T of a tree decomposition is rooted. A *rooted tree decomposition* is a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ where T is a rooted tree. Similarly, a *binary tree decomposition* is a tree decomposition where T is a binary tree.

The *adhesion* of a non-root node $t \in V(T)$ of a rooted tree decomposition is the intersection $\mathsf{bag}(t) \cap \mathsf{bag}(\mathsf{parent}(t))$ of the bags of t and its parent, and is denoted by $\mathsf{adh}_{\mathcal{T}}(t)$, where the subscript $\mathcal{T}$ may be dropped. If t is the root, then $\mathsf{adh}_{\mathcal{T}}(t) = \emptyset$. The *component* of a node t, denoted by $\mathsf{cmp}_{\mathcal{T}}(t)$, is the union of the bags of descendants of t minus the adhesion of t, i.e., $\mathsf{cmp}_{\mathcal{T}}(t) = \mathsf{bags}_{\mathcal{T}}(\mathsf{desc}_T(t)) \setminus \mathsf{adh}_{\mathcal{T}}(t)$. Note that by Lemma 2.7, if $\mathcal{T}$ is a tree decomposition G, then $(\mathsf{cmp}(t), \mathsf{adh}(t), V(G) \setminus (\mathsf{cmp}(t) \cup \mathsf{adh}(t)))$ is a separation of G.

If $\mathcal{T} = (T, \mathsf{bag})$ is a rooted tree decomposition of G and $v \in V(G)$, then the *forget-node* of v is the node with the smallest depth whose bag contains v. Equivalently, the forget-node of v is the unique node whose bag contains v but whose parent's bag does not contain v (or which does not have a parent). We denote the forget-node of v by $\mathsf{forget}_{\mathcal{T}}(v)$. We say that v is a forget-vertex of t if t is the forget-node of v. In this thesis, an important role will be played by the *vertex-depth function* $\mathsf{depth}_{\mathcal{T}}(v) = \mathsf{depth}_{\mathcal{T}}(\mathsf{forget}_{\mathcal{T}}(v))$ that maps the vertices of G to the depths of their forget-nodes.

Note that if $uv \in E(G)$, then the forget-nodes of u and v are in an ancestor-descendant relation. The *forget-node* of the edge uv is the node with the smallest depth whose bag contains both u and v, and is denoted by $\mathsf{forget}_{\mathcal{T}}(uv)$. We observe that $\mathsf{forget}_{\mathcal{T}}(uv)$ is equal to either $\mathsf{forget}_{\mathcal{T}}(u)$ or $\mathsf{forget}_{\mathcal{T}}(v)$, whichever has greater depth.

With these definitions in place, it is easy to prove the following well-known lemma about cliques and tree decompositions.

Lemma 2.12. *If $W \subseteq V(G)$ is a clique in a graph G and (T, bag) is a tree decomposition of G, then there exists a node $t \in V(T)$ so that $W \subseteq \mathsf{bag}(t)$.*

Proof. Let us root $\mathcal{T} = (T, \mathsf{bag})$ at an arbitrary selected root. Then, choose $v \in W$ that maximizes $\mathsf{depth}_{\mathcal{T}}(\mathsf{forget}_{\mathcal{T}}(v))$ among all vertices in W. Now, for all $u \in W \setminus \{v\}$, it holds that $\mathsf{forget}_{\mathcal{T}}(uv) = \mathsf{forget}_{\mathcal{T}}(v)$, implying $W \subseteq \mathsf{bag}(\mathsf{forget}_{\mathcal{T}}(v))$. □

Note that the idea of the proof of Lemma 2.12 can be used to devise a data structure that given a clique $W \subseteq V(G)$, returns a node $t \in V(T)$ with $W \subseteq \mathsf{bag}(t)$ in time $\mathcal{O}(|W|)$.

In algorithms that work on tree decompositions, it is often helpful to assume additional properties of the tree decomposition. For example, it is a convenient, and often implicitly made assumption in the literature, that the number of nodes of a tree decomposition of an n-vertex graph is $\mathcal{O}(n)$, or $k^{\mathcal{O}(1)}n$, where k is the width of the tree decomposition. We also make this assumption when talking about algorithms that take tree decompositions as input. This assumption is justified by the following lemma, that transforms any tree decomposition into a tree decomposition of such form.

Lemma 2.13. *There is an algorithm, that given a tree decomposition (T, bag) of an n-vertex graph G of width k, in time $k^{\mathcal{O}(1)}|V(T)|$ returns a tree decomposition (T', bag') of G with $|V(T')| \leq n$, and so that there exists an injective function $\phi\colon V(T') \to V(T)$ so that for all $t' \in V(T')$, $\mathsf{bag}'(t') = \mathsf{bag}(\phi(t'))$.*

Proof. Let us root (T, bag) at an arbitrary node with non-empty bag. Then, while (T, bag) contains a node t with parent p so that $\mathsf{bag}(t) \subseteq \mathsf{bag}(p)$, we delete t and attach all children of t as children of p. This can be implemented by depth-first-search in $k^{\mathcal{O}(1)}|V(T)|$ time. This results in a tree decomposition (T', bag') of G for which such function ϕ clearly exists. Moreover, every node of (T', bag') is a forget-node of some vertex, because if a non-root node would not be a forget-node of a vertex then its bag would be a subset of its parent's bag. The root is a forget-node because its bag is non-empty. Therefore, because each vertex of G has a unique forget-node, T' has at most n nodes. □

The purpose of the function ϕ in the statement of Lemma 2.13 is to formalize the idea that no matter how we measure the quality of tree decompositions, the tree decomposition (T', bag') outputted by the algorithm of the lemma is no worse than (T, bag).

Another convenient form of tree decompositions is binary tree decompositions.

Lemma 2.14. *There is an algorithm, that given a tree decomposition (T, bag) of a graph G of width k, in time $\mathcal{O}(k \cdot |V(T)|)$ outputs a binary tree decomposition (T', bag') of G of width k so that $|V(T')| \leq \mathcal{O}(|V(T)|)$.*

Proof. Let us root (T, bag) at an arbitrary node $r \in V(T)$. Then, we process (T, bag) bottom-up from the leaves to the root, and every time we encounter a node t with $c \geq 3$

children, we replace the node with a binary tree with c leaves with bags equal to $\mathsf{bag}(t)$. The sum of the sizes of constructed binary trees is linear in the sum of the degrees of T, so therefore $|V(T')| = \mathcal{O}(|V(T)|)$. □

Finally, a classical form of a tree decomposition with useful extra structure is that of *nice tree decompositions*, which to the best of our knowledge was first used by Bodlaender and Kloks [1996]. We follow the definition of [Cygan et al., 2015, Chapter 7]. A nice tree decomposition is a binary tree decomposition (T, bag), where

1. $\mathsf{bag}(r) = \emptyset$ for the root node r,
2. $\mathsf{bag}(\ell) = \emptyset$ for every leaf node ℓ, and
3. every non-leaf node t either
   (a) has exactly one child c so that $|\mathsf{bag}(t) \setminus \mathsf{bag}(c)| + |\mathsf{bag}(c) \setminus \mathsf{bag}(t)| = 1$, or
   (b) has exactly two children c_1 and c_2 so that $\mathsf{bag}(t) = \mathsf{bag}(c_1) = \mathsf{bag}(c_2)$.

We recall the well-known fact that tree decompositions can be efficiently turned into nice tree decompositions.

Lemma 2.15. *There is an algorithm that, given a tree decomposition (T, bag) of an n-vertex graph G of width k, in time $k^{\mathcal{O}(1)}|V(T)|$ outputs a nice tree decomposition (T', bag') of G of width at most k and $|V(T')| \leq \mathcal{O}(kn)$.*

Proof. First, let us use the combination of Lemmas 2.13 and 2.14 to obtain a binary tree decomposition (T_0, bag_0) of G of width at most k and $|V(T_0)| \leq \mathcal{O}(n)$.

First, we change (T_0, bag_0) by adding a new root adjacent only to the previous root with an empty bag, and for each leaf a new child with an empty bag. The resulting decomposition (T_1, bag_1) has width at most k, has $|V(T_1)| \leq \mathcal{O}(n)$, and satisfies Items 1 and 2 of the definition of nice tree decompositions.

Then, let us guarantee that for every node t of T_1 with two children, the bags of the children are equal to the bag of t by subdividing the edges between t and its children and adding new bags equal to $\mathsf{bag}_1(t)$ on the newly created nodes. The resulting decomposition (T_2, bag_2) has width at most k, $|V(T_2)| \leq \mathcal{O}(n)$, still satisfies Items 1 and 2, and its nodes with two children satisfy Item 3.

Then, for every edge ct of (T_2, bag_2), where c is a child of t, if $|\mathsf{bag}_2(t) \setminus \mathsf{bag}_2(c)| + |\mathsf{bag}_2(c) \setminus \mathsf{bag}_2(t)| > 1$, let us subdivide the edge ct into a path where first the vertices in

$\mathsf{bag}_2(c) \setminus \mathsf{bag}_2(t)$ are "forgotten" one at a time, and then the vertices in $\mathsf{bag}_2(t) \setminus \mathsf{bag}_2(c)$ are "introduced" one at a time. The resulting decomposition (T_3, bag_3) has width at most k, has $|V(T_3)| \leq \mathcal{O}(kn)$, satisfies Items 1 and 2, its nodes with two children satisfy Item 3, and its nodes t with one children c satisfy that $|\mathsf{bag}_2(t) \setminus \mathsf{bag}_2(c)| + |\mathsf{bag}_2(c) \setminus \mathsf{bag}_2(t)| \leq 1$.

Finally, to turn (T_3, bag_3) into a nice tree decomposition, it suffices to contract edges ct between a child c and a parent t, where t has only one child and $\mathsf{bag}_3(c) = \mathsf{bag}_3(t)$.

All of the transformations from (T_0, bag_0) to the final nice tree decomposition can be implemented in time $k^{\mathcal{O}(1)}|V(T_0)|$, so the overall running time is $k^{\mathcal{O}(1)}(|V(T)| + |V(T_0)|) = k^{\mathcal{O}(1)}|V(T)|$. □

2.3.2 Branchwidth of connectivity functions

Chapter 7 of this thesis is about the graph width parameters rankwidth and branchwidth, and their common generalization branchwidth of connectivity functions. We first give the general definition of branchwidth of connectivity functions, and then define the branchwidth and rankwidth of graphs in Subsections 2.3.3 and 2.3.4, respectively.

Connectivity functions

Let V be a set. A function $f\colon 2^V \to \mathbb{Z}$ from subsets of V to integers is *submodular* if

$$f(A) + f(B) \geq f(A \cap B) + f(A \cup B) \tag{2.16}$$

for all $A, B \subseteq V$. The function f is *symmetric* if $f(A) = f(V \setminus A)$ for all $A \subseteq V$. In this context, we denote $\overline{A} = V \setminus A$.

Following the definition given by Robertson and Seymour [1991], we say that a function $f\colon 2^V \to \mathbb{Z}_{\geq 0}$ is a *connectivity function* if it is both symmetric and submodular, and $f(\emptyset) = f(V) = 0$. We note that requirements that f is non-negative and $f(\emptyset) = 0$ are not real restrictions, because it is easy to show that if $f\colon 2^V \to \mathbb{Z}$ is symmetric and submodular, then the function $f - f(\emptyset)$ is also symmetric, submodular, and non-negative.[1]

[1]The proof of the non-negativity of $f - f(\emptyset)$ goes by applying (2.16) with A and $B = \overline{A}$.

Branch decompositions

Let V be a set with $|V| \geq 2$ and $f\colon 2^V \to \mathbb{Z}_{\geq 0}$ a connectivity function. A *branch decomposition* of f, as defined by [Oum and Seymour, 2006], is a pair $\mathcal{T} = (T, \lambda)$, where T is a cubic tree, and λ is a bijection from V to the leaves of T. In particular, the number of leaves of T is $|V|$, and therefore as T is cubic, the number of nodes of T is $|V(T)| = 2|V| - 2$. For brevity, we sometimes denote $V(\mathcal{T}) = V(T)$, $E(\mathcal{T}) = E(T)$, and refer to nodes and edges of T as nodes and edges of $\mathcal{T}$.

Let $uv \in E(\mathcal{T})$ be an edge of $\mathcal{T}$. We denote by $\mathcal{T}[uv] \subseteq V$ the subset of V that is mapped by λ to the leaves that are closer to u than v. In other words, $\mathcal{T}[uv]$ contains the elements $x \in V$ so that the unique path in T from $\lambda(x)$ to u does not contain v. Note that $\mathcal{T}[uv] = \overline{\mathcal{T}[vu]}$, and in particular that although uv and vu refer to the same edge of $\mathcal{T}$, in the context of this notation the order of u and v matters.

Now, we define the *width* of the edge $uv \in E(T)$ to be $f(uv) = f(\mathcal{T}[uv]) = f(\mathcal{T}[vu])$. Then, the *width* of the branch decomposition $\mathcal{T}$, denoted by $\mathsf{width}(\mathcal{T})$, is the maximum width of an edge of it. The *branchwidth* $\mathsf{bw}(f)$ of a connectivity function f is the minimum width of a branch decomposition of it, or if $|V| \leq 1$, we define the branchwidth of f to be 0.

2.3.3 Branchwidth of graphs

The branchwidth of a graph was defined by Robertson and Seymour [1991]. Here we give an equivalent definition as a special case of branchwidth of connectivity functions. Let G be a graph. We define the branchwidth of G via a connectivity function that maps subsets of $E(G)$ to non-negative integers. In particular, the leaves of a branch decomposition of G correspond to the edges of G.

Let $X \subseteq E(G)$. The *border* $\delta_G(X)$ of X is the set of vertices $\delta_G(X) \subseteq V(G)$ that are incident to both an edge in X and an edge in $E(G) \setminus X$. Now, $|\delta_G|\colon 2^{E(G)} \to \mathbb{Z}_{\geq 0}$ is the function that maps each $X \subseteq E(G)$ to the size $|\delta_G(X)|$ of its border.

Lemma 2.17. *The function $|\delta_G|$ is a connectivity function.*

Proof. It is obvious from the definition that $\delta_G(X) = \delta_G(\overline{X})$, and therefore $|\delta_G|$ is symmetric. For the submodularity, we first observe that if $v \in \delta_G(A \cap B)$, then either $v \in \delta_G(A)$ or $v \in \delta_G(B)$, and therefore $\delta_G(A \cap B) \subseteq \delta_G(A) \cup \delta_G(B)$. It also follows that

$$\delta_G(A \cup B) = \delta_G(\overline{A} \cap \overline{B}) \subseteq \delta_G(\overline{A}) \cup \delta_G(\overline{B}) = \delta_G(A) \cup \delta_G(B),$$

and therefore $\delta_G(A \cap B) \cup \delta_G(A \cup B) \subseteq \delta_G(A) \cup \delta_G(B)$. Suppose that v is in both $\delta_G(A \cap B)$ and $\delta_G(A \cup B) = \delta_G(\overline{A} \cap \overline{B})$. Now, v is incident to edges in all A, B, $\overline{A}$, and $\overline{B}$, implying that v is in both $\delta_G(A)$ and $\delta_G(B)$. Therefore, we obtain by counting that $|\delta_G(A)| + |\delta_G(B)| \geq |\delta_G(A \cup B)| + |\delta_G(A \cap B)|$. □

Now we can define a *branch decomposition* of G to be a branch decomposition of $|\delta_G|$ and the *branchwidth* of G, denoted by $\mathsf{bw}(G)$, to be the branchwidth of $|\delta_G|$. Robertson and Seymour [1991] showed the following relation of branchwidth and treewidth.

Lemma 2.18 (Robertson and Seymour [1991])**.** *For all graphs G it holds that* $\mathsf{bw}(G) \leq \mathsf{tw}(G) + 1 \leq \max(\frac{3}{2} \cdot \mathsf{bw}(G), 2)$.

The idea of the proof of the bound $\mathsf{tw}(G) + 1 \leq \max(\frac{3}{2} \cdot \mathsf{bw}(G), 2)$ is to map a branch decomposition $\mathcal{T} = (T, \lambda)$ to a tree decomposition (T, bag) so that each leaf $\ell \in V(T)$ has $\mathsf{bag}(\ell) = \{u, v\}$, where $uv \in E(G)$ and $\lambda(uv) = \ell$, and each non-leaf $t \in V(T)$ has $\mathsf{bag}(t) = \delta_G(\mathcal{T}[xt]) \cup \delta_G(\mathcal{T}[yt]) \cup \delta_G(\mathcal{T}[zt])$, where $\{x, y, z\} = N_T(t)$. The bound $\mathsf{bw}(G) \leq \mathsf{tw}(G) + 1$ is shown by first showing that any tree decomposition can be transformed to the form of the tree decomposition (T, bag) defined above without increasing its width, and then reversing the construction.

2.3.4 Rankwidth

Rankwidth was defined by Oum and Seymour [2006]. Let G be a graph, $A \subseteq V(G)$, and recall that $G[A, \overline{A}]$ denotes the bipartite graph containing the edges with one endpoint in A and other in $\overline{A} = V(G) \setminus A$. Let $M_G(A)$ be the $|A| \times |\overline{A}|$ matrix over the binary field $\mathsf{GF}(2)$, whose rows are indexed by A and columns by $\overline{A}$, and which describes the edges of the graph $G[A, \overline{A}]$ by zeros and ones. We define $\mathsf{cutrk}_G(A)$ to be the rank of $M_G(A)$.

Lemma 2.19 (Oum and Seymour [2006])**.** *The function* $\mathsf{cutrk}_G \colon 2^{V(G)} \to \mathbb{Z}_{\geq 0}$ *is a connectivity function.*

Now, a *rank decomposition* of G is a branch decomposition of cutrk_G, and the *rankwidth* of G is the branchwidth of cutrk_G, and is denoted by $\mathsf{rw}(G)$.

For branchwidth of graphs, it is clear that a partition $\{X, \overline{X}\}$ of $E(G)$ with a small value of $|\delta_G(X)|$ is "simple", because the sets of edges X and $\overline{X}$ interact only via a small number of vertices, namely $\delta_G(X)$. For rankwidth, the simplicity of cuts $(A, \overline{A})$ with a small value of $\mathsf{cutrk}_G(A)$ is a bit more subtle. One way to capture it is via representatives of sets of vertices.

Let $A \subseteq V(G)$. We say that a set $R \subseteq A$ is a *representative* of A if for every vertex $v \in A$ there exists a vertex $u \in R$ so that $N(v) \setminus A = N(u) \setminus A$. In other words, every neighborhood from A into $\overline{A}$ should be represented by some vertex in the representative. A representative R is *minimal* if for every $v \in A$ there exists exactly one such $u \in R$. The following observation is vital for dynamic programming on rank decompositions of small width.

Lemma 2.20. *If* $\mathsf{cutrk}_G(A) \leq k$ *and* R *is a minimal representative of* A*, then* $|R| \leq 2^k$.

Proof. The sets $N(v) \setminus A$ for $v \in A$ correspond to the rows of the matrix $M_G(A)$, and in particular, the sets $N(u) \setminus A$ for $u \in R$ correspond to distinct rows of $M_G(A)$. Because $M_G(A)$ has rank at most k, it can have at most 2^k distinct rows. □

2.3.5 Cliquewidth

Cliquewidth was introduced by Courcelle et al. [1993], and defined in its present form by Courcelle [1995]. We give a definition that follows that of the latter reference.

For $k \in \mathbb{Z}_{\geq 1}$, a *$k$-graph* is a pair $\mathcal{G} = (G, \mu)$, where G is a graph and $\mu\colon V(G) \to [k]$ is a function that labels the vertices of G with integers from $[k]$. A *k-expression* is an algebraic expression that constructs k-graphs by using the following four operations.

- Introduce: Construct a k-graph (G, μ), where G is the single-vertex graph and $\mu(v) = 1$ for the only vertex $v \in V(G)$.
- Union: Given two k-graphs (G_1, μ_1) and (G_2, μ_2), construct the disjoint union of them, i.e., the k-graph (G, μ) with $V(G) = V(G_1) \cup V(G_2)$, $E(G) = E(G_1) \cup E(G_2)$, $\mu(v) = \mu_1(v)$ for $v \in V(G_1)$, and $\mu(v) = \mu_2(v)$ for $v \in V(G_2)$.
- Relabel: Given a k-graph (G, μ) and two distinct integers $x, y \in [k]$, construct the k-graph (G, μ'), where $\mu'(v) = y$ for all $v \in V(G)$ with $\mu(v) \in \{x, y\}$, and $\mu'(v) = \mu(v)$ for other vertices v.
- Join: Given a k-graph (G, μ) and two distinct integers $x, y \in [k]$, construct the k-graph (G', μ), where G' is obtained from G by adding all edges uv with $\mu(u) = x$ and $\mu(v) = y$.

A *k-expression* of a graph G is a k-expression that constructs the k-graph (G, μ), where $\mu(v) = 1$ for all $v \in V(G)$. The *cliquewidth* of a graph G, denoted by $\mathsf{cw}(G)$, is the smallest k so that there exists a k-expression of G. It is not hard to observe (and indeed

was observed for example by Courcelle et al. [2000]) that any k-expression can be turned into an equivalent k-expression with at most $k^{\mathcal{O}(1)} \cdot n$ operations by removing redundant operations. Therefore, we will assume that k-expressions have size bounded by $k^{\mathcal{O}(1)} \cdot n$.

Rankwidth was introduced by Oum and Seymour [2006] to approximate cliquewidth. They showed that the two parameters are tied together in the following way.

Lemma 2.21 (Oum and Seymour [2006])**.** *For all graphs,* $\mathsf{rw}(G) \leq \mathsf{cw}(G) \leq 2^{\mathsf{rw}(G)+1} - 1$. *Moreover, there is an algorithm that, given a graph G and a rank decomposition of G of width k, in time $2^{\mathcal{O}(k)}n^2$ returns a $(2^{k+1} - 1)$-expression of G.*

The rough idea of the construction of the $(2^{k+1}-1)$-expression given a rank decomposition of width k is to follow the rank decomposition and choose the labeling μ based on minimal representatives. In particular, suppose we have rooted the tree T of a rank decomposition $\mathcal{T} = (T, \lambda)$ at some node r, and consider an edge $tp \in E(T)$, where p is the parent of t. Then, we wish to associate with tp a 2^k-graph $(G[\mathcal{T}[tp]], \mu)$, where μ is chosen based on the partition of $\mathcal{T}[tp]$ into equivalence classes of $N_G(v) \setminus \mathcal{T}[tp]$. In particular, if $R = \{u_1, \ldots, u_\ell\}$ is a minimal representative of $\mathcal{T}[tp]$, which has size at most $\ell \leq 2^k$ by Lemma 2.20, then we label $v \in \mathcal{T}[tp]$ with $i \in [2^k]$ if $N_G(v) \setminus \mathcal{T}[tp] = N_G(u_i) \setminus \mathcal{T}[tp]$. The idea of the construction is that, if x and y are the children of t, then the 2^k-graph corresponding to tp can be constructed from the 2^k-graphs corresponding to xt and yt via a $(2^{k+1} - 1)$-expression.

2.4 Computational complexity

We briefly review the main complexity-theoretic definitions relevant to this thesis. We assume that the reader is familiar with the concept of NP-hardness, and refer to [Sipser, 2012, Chapter 7]. We assume the word RAM model of computation with words of $\Theta(\log n)$ bits, where n is the input length, as is customary.

Amortized running time

When we say that some operation in a data structure has *amortized running time* of $f(\bar{x})$, where $\bar{x}$ is a tuple of parameters, we mean that the total time for the first t applications of this operation is at most $t \cdot f(\bar{x})$ for every $t \in \mathbb{Z}_{\geq 0}$. When multiple different operations in the same data structure, say $\mathsf{op}_1, \ldots, \mathsf{op}_p$, have amortized running times $f_1(\bar{x_1}), \ldots, f_p(\bar{x_p})$, we interpret this as saying that the total running time of applying the operations $t_1, \ldots, t_p$ times, respectively, is at most $\sum_{i=1}^{p} t_i \cdot f_i(\bar{x_i})$. For example, if the

initialization operation has amortized running time $\mathcal{O}(n)$ and the update operation has amortized running time $\mathcal{O}(1)$, then the first update after the initialization is allowed to run in $\Theta(n)$ time, but the total running time across the first n updates must be $\mathcal{O}(n)$.

Fixed-parameter tractability

We give a formal definition of fixed-parameter tractability. Our definition is similar to that of [Courcelle and Engelfriet, 2012, Chapter 1], and is slightly more general than the definitions of [Downey and Fellows, 2013, Chapter 2] and [Cygan et al., 2015, Chapter 1].[2] Let Σ be a fixed alphabet. A *parameterized problem* is a pair (L, κ), where $L \subseteq \Sigma^*$ is a language over Σ, and κ is a function $\kappa\colon \Sigma^* \to \mathbb{Z}_{\geq 0}$. An *instance* of (L, κ) is a word $x \in \Sigma^*$.

For example, the 3-coloring problem parameterized by treewidth can be encoded as a pair (L, κ), where $L \subseteq \{0,1\}^*$ is a language that contains a word $x \in \{0,1\}^*$ if x is an encoding of a graph that is 3-colorable, and κ is a function that maps words that encode graphs to their treewidth, and words that do not encode graphs to 0.

A parameterized problem is *fixed-parameter tractable* (FPT) if there exists an algorithm for deciding if $x \in L$ with running time $f(\kappa(x)) \cdot |x|^c$ for a computable function f and a constant c. Such an algorithm is called a *fixed-parameter algorithm*, or just an FPT algorithm. We also refer to the class of fixed-parameter tractable problems as FPT. A *slice-wise polynomial-time* (XP) algorithm is an algorithm that runs in time $|x|^{f(\kappa(x))}$ for a computable function f. Note that every FPT algorithm is also an XP algorithm, but not the other way around.

The standard argument for showing that a problem is probably not fixed-parameter tractable is to show that it is hard for a complexity class called W[1]. The class W[1] is a class of parameterized problems, which is believed to be a strict superset of the class FPT (see e.g. [Downey and Fellows, 2013, Chapter 21] and [Cygan et al., 2015, Chapter 13]). The formal definition of W[1] is complicated (we refer the reader to the above two references), but W[1]-hardness can be defined via parameterized reductions to the W[1]-complete problem of deciding if a graph contains a clique of size k, parameterized by k.

A *parameterized reduction* from a problem $P_1 = (L_1, \kappa_1)$ to a problem $P_2 = (L_2, \kappa_2)$ is an algorithm, that given an instance x_1 of P_1, runs in time $f(\kappa_1(x_1)) \cdot |x_1|^c$ for a computable function f and a constant c, and outputs an instance x_2 of P_2 so that $x_2 \in L_2$ if and only

[2] The definitions in [Downey and Fellows, 2013, Chapter 2] and [Cygan et al., 2015, Chapter 1] do not accommodate a natural way of stating that 3-coloring is fixed-parameter tractable parameterized by cliquewidth, because they require an FPT algorithm to compute the exact value of the parameter, but no exact FPT algorithms are known for cliquewidth.

if $x_1 \in L_1$, and $\kappa_2(x_2) \leq g(\kappa_1(x_1))$ for a computable function g. Then, a parameterized problem P is W[1]-hard if there exists a parameterized reduction from the problem of deciding if a given graph contains a clique of size k, parameterized by k, to the problem P.

Exponential time hypotheses

For problems known to be FPT, one can ask if the running time of a known FPT algorithm is the best possible. This can be asked both for the dependence f on the parameter and the exponent c of the polynomial in the running time. In the past 15 years, there has been a lot of success in showing lower bounds for the function f on the running times of various parameterized problems, assuming the so-called "Exponential Time Hypothesis" (ETH) or the "Strong Exponential Time Hypothesis" (SETH) [Cygan et al., 2016; Lokshtanov et al., 2018a,b]. The *ETH*, introduced by Impagliazzo and Paturi [2001] (see also Impagliazzo et al. [2001]), states that there is no $2^{o(n)}$ time algorithm for 3-SAT, where n is the number of variables. Impagliazzo and Paturi [2001] also introduced the hypothesis stating that for every $\varepsilon > 0$, there exists k so that there is no $\mathcal{O}((2-\varepsilon)^n)$ time algorithm for k-SAT, which was later dubbed the *SETH*.

So far there has been very little success in establishing tight lower bounds under the (S)ETH for algorithms computing graph width parameters. In particular, to the best of the author's knowledge, no tight lower bounds under the ETH are known for computing any of the width parameters that are the topics of this thesis. We discuss this further in Chapter 8.

Chapter 3

Survey of the literature

In this chapter we survey some previous results in more detail than we did in Chapter 1. We start by reviewing the algorithms of Robertson and Seymour [1995] and Bodlaender [1996] for computing treewidth. These have been the two most influential algorithms for computing graph width parameters, and most of the later FPT algorithms for width parameters are based on either of them. After that, we review applications of graph width parameters in Section 3.3. Finally, in Section 3.4 we review the proof of [Bellenbaum and Diestel, 2002; Thomas, 1990] on the existence of lean tree decompositions.

Several lemmas reviewed (and proved) in this chapter will be used and referred to later in this thesis, and the ideas reviewed in Section 3.4 will be the starting point for ideas introduced in the subsequent chapters. Nevertheless, the later chapters can be read mostly independently of this chapter.

3.1 Robertson-Seymour algorithm

Robertson and Seymour [1995] gave an algorithm that, given a graph G and an integer k, in time $2^{\mathcal{O}(k)}n^2$ either returns a branch decomposition of G of width at most $3k$, or concludes that the branchwidth of G is more than k. In the subsequent literature, perhaps first by Reed [1992], this algorithm has been interpreted as the following theorem about 4-approximating treewidth.

Theorem 3.1 (Robertson and Seymour [1995])**.** *There is an algorithm that, given an n-vertex graph G and an integer k, in time $\mathcal{O}(3^{3k} \cdot k^2 \cdot n^2)$ either outputs a tree decomposition of G of width at most $4k + 3$, or concludes that $\mathsf{tw}(G) > k$.*

We follow this interpretation as a treewidth approximation algorithm, which has, in addition to Reed [1992], been reviewed also by [Kleinberg and Tardos, 2005, Chapter 10], [Flum and Grohe, 2006, Chapter 11], [Cygan et al., 2015, Chapter 7], and [Pilipczuk, 2020]. We first describe and analyze the algorithm in Subsection 3.1.1, and then in Subsection 3.1.2 briefly discuss the vast body of literature inspired by this algorithm.

3.1.1 The algorithm

The Robertson-Seymour algorithm constructs a rooted tree decomposition in a greedy, top-down manner, from the root towards the leaves. The main idea is that no matter what subset of vertices we choose to have in the root bag of the decomposition, we can proceed the construction by chopping this set into smaller pieces by small balanced separators. To express this idea more formally, let us introduce some notation.

Let $W \subseteq V(G)$ be a set of vertices in a graph G. We say that a set $S \subseteq V(G)$ is a *W-balanced separator* if for every connected component $C \in \mathsf{cc}(G \setminus S)$ it holds that $|C \cap W| \leq |W|/2$. In particular, if W is large enough compared to S, then S must be a separator of G that breaks the vertices in W in a balanced manner. Our first lemma formalizes the idea that in graphs of small treewidth, there is an abundance of small separators. In particular, every set W has a W-balanced separator of size $\mathsf{tw}(G) + 1$.

Lemma 3.2. *If G is a graph of treewidth k and $W \subseteq V(G)$ is a set of vertices, then there exists a W-balanced separator of size at most $k + 1$.*

Proof. Let (T, bag) be a rooted tree decomposition of G of width k. Then, let t be a node of (T, bag) so that $|W \cap \mathsf{bags}(\mathsf{desc}(t))| > |W|/2$, but for all children c of t is holds that $|W \cap \mathsf{bags}(\mathsf{desc}(c))| \leq |W|/2$. Such a node t can be found by starting from the root and iteratively walking down while there exists a child c with $|W \cap \mathsf{bags}(\mathsf{desc}(c))| > |W|/2$.

We claim that $S = \mathsf{bag}(t)$ is a W-balanced separator. Consider a connected component $C \in \mathsf{cc}(G \setminus S)$. The set of nodes of T whose bags contain vertices from C forms a connected subtree that is disjoint from t. If the subtree is contained in a subtree of T rooted at some child c of t, then $|W \cap C| \leq |W \cap \mathsf{bags}(\mathsf{desc}(c))| \leq |W|/2$. Otherwise, if the subtree is disjoint from the subtree of T rooted at t, then C is disjoint from $\mathsf{bags}(\mathsf{desc}(t))$, implying that $|W \cap C| \leq |W| - |W \cap \mathsf{bags}(\mathsf{desc}(t))| < |W|/2$. □

Now we can describe a recursive method to construct an approximately optimal tree decomposition of a graph G. At this point, we will not worry about how to actually algorithmically implement this method, but instead only focus on the construction.

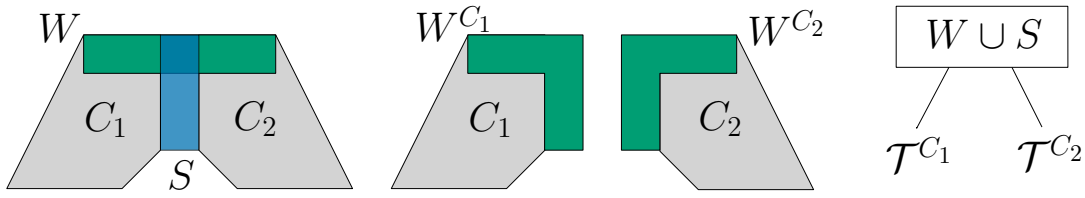

Figure 3.1: Illustration of the Robertson-Seymour algorithm. The graph G is depicted on the left, with the set W, the separator S, and the connected components C_1 and C_2 of $G \setminus S$. The graphs $G[C_1 \cup S]$ and $G[C_2 \cup S]$, together with the sets W^{C_1} and W^{C_2} are depicted in the middle. The construction of the tree decomposition $\mathcal{T}$ of G from the tree decompositions $\mathcal{T}^{C_1}$ and $\mathcal{T}^{C_2}$ of $G[C_1 \cup S]$ and $G[C_2 \cup S]$ is depicted on the right.

The method takes as input a graph G and a set of vertices $W \subseteq V(G)$. It outputs a rooted tree decomposition of G that contains W as a subset of its root bag. Let us furthermore assume that the treewidth of G is k, and that $|W| = 2k + 3$. If W is smaller than $2k + 3$, then we can add arbitrary vertices of G to it to make it larger (and if no vertices in $V(G) \setminus W$ exist, then the single bag containing $W = V(G)$ already forms a desired tree decomposition).

Let us apply Lemma 3.2 to the set W. It gives us a W-balanced separator S of size at most $|S| \leq k + 1$. We construct a tree decomposition of G as follows. First, for every component $C \in \mathsf{cc}(G \setminus S)$, we call the method recursively to construct a tree decomposition of $G[C \cup S]$ that contains the set $W^C = (W \cap C) \cup S$ in its root bag. Let us denote this tree decomposition by $\mathcal{T}^C$. Then, we construct a tree decomposition $\mathcal{T}$ of G by letting its root bag be $W \cup S$, and attaching the decompositions $\mathcal{T}^C$, for all $C \in \mathsf{cc}(G \setminus S)$, from their roots as the children of the root bag $W \cup S$. See Figure 3.1 for an illustration.

We first argue that $\mathcal{T}$ is indeed a tree decomposition of G, assuming that each $\mathcal{T}^C$ is a tree decomposition of $G[C \cup S]$ that contains $W^C = (W \cap C) \cup S$ in its root bag.

Lemma 3.3. *$\mathcal{T}$ is a tree decomposition of G.*

Proof. Let us check the three conditions of tree decompositions. First, $\mathcal{T}$ satisfies the vertex condition because all vertices of G are in at least one of the induced subgraphs $G[C \cup S]$ for $C \in \mathsf{cc}(G \setminus S)$, so all vertices of G are already contained in the tree decompositions $\mathcal{T}^C$. The same argument works for checking that $\mathcal{T}$ satisfies the edge condition, as every edge of G is an edge of at least one induced subgraph $G[C \cup S]$.

We then check the connectedness condition for vertices $v \in V(G) \setminus S$. The vertex v is in exactly one set C for $C \in \mathsf{cc}(G \setminus S)$, so v occurs in exactly one of the decompositions $\mathcal{T}^C$. Therefore, because the decompositions $\mathcal{T}^C$ satisfy the connectedness condition, the

only thing to check is that if $v \in W \cup S$, then v is in the root bag of $\mathcal{T}^C$, which is indeed true. The same argument works for $v \in S$, in particular, in that case v is in the root bag of $\mathcal{T}$ and in the root bags of every $\mathcal{T}^C$. □

Then, we should verify that the recursive calls of the method satisfy the assumption that the set W^C asked to be in the root bag has size at most $2k+3$. This follows from the fact that S is W-balanced separator, in particular

$$|W^C| = |(W \cap C) \cup S| \leq |W|/2 + k + 1 < 2k + 3.$$

The stronger property $|W^C| < 2k+3$ also implies that the recursion does not continue forever, as the fact that W^C is smaller than W implies that $G[C \cup S]$ is smaller than G.

Finally, what is the width of $\mathcal{T}$? The size of the root bag $W \cup S$ is at most $2k+3+k+1 \leq 3k+4$, so we can state an invariant that the method returns a tree decomposition of width at most $3k+3$, and the induction works out to show that the width of the resulting decomposition $\mathcal{T}$ is indeed at most $3k+3$.

Now, $3k+3$ is even better than the $4k+3$ we wanted, so what went wrong? The problem with the method we described is that we did not give an algorithm for finding the W-balanced separators of Lemma 3.2. With modern techniques ([Cygan et al., 2015, Chapter 8], [Chen et al., 2009]), one could design a $2^{\mathcal{O}(k)}m$ time algorithm for finding such separators, but there is a simpler solution by Robertson and Seymour [1995], which is to resort to a weaker balance property, with the cost of a worse approximation ratio.

Lemma 3.4. *If G is a graph of treewidth k and $W \subseteq V(G)$ is a set of vertices with $|W| \geq 2k+3$, then there exists a separation (A, S, B) of G with $|S| \leq k+1$ and $0 < |W \cap A|, |W \cap B| \leq \frac{2}{3}|W|$.*

Proof. Let S be a W-balanced separator guaranteed by Lemma 3.2, and let $\{C_1, \ldots, C_\ell\} = \mathsf{cc}(G \setminus S)$ be the connected components of $G \setminus S$. Because $|W \setminus S| \geq |W| - (k+1) > |W|/2$, at least two of the component intersect W, so let us assume that C_1 and C_ℓ intersect W. Furthermore, assume that C_1 is the component with the largest intersection $|C_i \cap W|$ with W. Now, let $t \in [\ell - 1]$ be the greatest index less than ℓ so that $\sum_{i=1}^{t} |C_i \cap W| \leq \frac{2}{3}|W|$. We claim that $(A, S, B) = (\bigcup_{i=1}^{t} C_i, S, \bigcup_{i=t+1}^{\ell} C_i)$ is the desired separation.

As $t \in [\ell - 1]$, both A and B intersect W. Also, $|A \cap W| \leq \frac{2}{3}|W|$ by definition, so it remains to prove that $|B \cap W| \leq \frac{2}{3}|W|$. First, if $|C_1 \cap W| \geq |W|/3$, we are done as $C_1 \subseteq A$ and A is disjoint from B. Otherwise, $|C_i \cap W| < |W|/3$ for all $i \in [\ell]$, and therefore if $\sum_{i=1}^{t} |C_i \cap W| \leq |W|/3$, then $\sum_{i=1}^{t+1} |C_i \cap W| \leq \frac{2}{3}|W|$. If $t+1 < \ell$ we contradict the choice of t, and if $t+1 = \ell$, then $B = C_\ell$ and thus $|B \cap W| \leq |W|/2$. □

The utility of Lemma 3.4 is that the separations guaranteed by it can be found easily, at least if we allow running time exponential in $|W|$. In particular, if we guess the intersection $(A \cap W, S \cap W, B \cap W)$ of the separation and W, then the existence of a separation as in Lemma 3.4 can be tested in time $\mathcal{O}(k \cdot m)$ by the algorithm of Ford and Fulkerson (Theorem 2.2). By trying all possible intersections, whose number is upper bounded by $3^{|W|}$, we obtain an $\mathcal{O}(3^{|W|} \cdot k \cdot m)$ time algorithm that given a graph G, an integer k, and a set $W \subseteq V(G)$ with $|W| \geq 2k + 3$, either outputs a separation as in Lemma 3.4 or concludes that the treewidth of G is more than k.

Now, let us describe the actual algorithm. The algorithm takes as an input

- a graph G,
- an integer k,
- and a set $W \subseteq V(G)$ of size at most $|W| \leq 3k + 3$,

and outputs either

- a binary tree decomposition $\mathcal{T}$ of G of width at most $4k + 3$, such that W is a subset of the root bag of $\mathcal{T}$, or
- the conclusion that $\mathsf{tw}(G) > k$.

The algorithm works as follows. First, if $|W| < 3k + 3$, we add arbitrary vertices to W to make it larger. If no such vertices exist, i.e., $|V(G)| < 3k + 3$, we can output the tree decomposition with a single bag $V(G)$. Suppose then that $|W| = 3k + 3$.

We apply the algorithmic version of Lemma 3.4, that in time $\mathcal{O}(3^{|W|} \cdot k \cdot m) = \mathcal{O}(3^{3k} \cdot k \cdot m)$ either returns a separation (A, S, B) of G with $|S| \leq k+1$ and $0 < |W \cap A|, |W \cap B| \leq \frac{2}{3}|W|$, or concludes that $\mathsf{tw}(G) > k$. In the latter case, we can return immediately. In the former case, we call the algorithm recursively with the graphs $G^A = G[A \cup S]$ and $G^B = G[B \cup S]$ and the sets $W^A = (W \cap A) \cup S$ and $W^B = (W \cap B) \cup S$. If either of the calls returns that $\mathsf{tw}(G^A) > k$ or $\mathsf{tw}(G^B) > k$ we can return that $\mathsf{tw}(G) > k$. Otherwise, let $\mathcal{T}^A$ and $\mathcal{T}^B$ be the returned binary tree decompositions. We return the binary tree decomposition $\mathcal{T}$ that is constructed by having $W \cup S$ as its root bag, and attaching the tree decompositions $\mathcal{T}^A$ and $\mathcal{T}^B$ from their roots as the children.

Let us then check the correctness. Note that the proof of Lemma 3.3 can be directly adapted to show that $\mathcal{T}$ is indeed a tree decomposition of G. Furthermore, in all cases when we return that $\mathsf{tw}(G) > k$, this fact holds, in particular because G^A and G^B are

subgraphs of G. Because $\frac{2}{3}|W| + k + 1 \leq 3k + 3$, it holds that the sets W^A and W^B have size at most $3k + 3$, as required.

To analyze the running time, let us first analyze the total number of recursive calls. We observe that the recursion tree of the algorithm is in fact isomorphic to the tree decomposition outputted by it, so it suffices to analyze the number of nodes of it.

Lemma 3.5. *The tree decomposition returned by the algorithm has $\mathcal{O}(n)$ nodes.*

Proof. Let (T, bag) be the returned binary tree decomposition. We claim that for each non-leaf node $t \in V(T)$ of (T, bag), it holds that either t is a forget-node of a vertex of G, or t has a child that is a forget-node of a vertex of G, or both of its children are leaves. This implies the conclusion because each vertex of G has exactly one forget-node and T is a binary tree.

First, observe that if a bag of (T, bag) is created as $W \cup S$, then it is a forget-node of all vertices in $S \setminus W$, because the intersection of this bag with its parent is W. It remains to consider the case when $S \subseteq W$, in which case the node is not a forget-node of any vertex, so we have to argue about its children. In this case, because $|W \cap A|, |W \cap B| > 0$, it holds that $|W^A|, |W^B| < |W|$. Therefore, as both W^A and W^B are smaller than $3k + 3$, the recursive steps start with adding vertices to them, or terminating as leaves if no vertices can be added. In the case of adding vertices, these nodes become forget-nodes of the added vertices, so it must be that at least one of the children is a forget-node of a vertex, or both of the children are leaves. □

The time taken by one recursive call is $\mathcal{O}(3^{3k} \cdot k \cdot m)$ for the algorithmic version of Lemma 3.4, plus the time taken to combine the tree decompositions $\mathcal{T}^A$ and $\mathcal{T}^B$ into $\mathcal{T}$, which can be bounded by $\mathcal{O}(k \cdot n)$ because they have $\mathcal{O}(n)$ nodes. Therefore, as the number of recursive steps is $\mathcal{O}(n)$, the total running time of the algorithm is $\mathcal{O}(3^{3k} \cdot k \cdot m \cdot n)$. By Lemma 2.9 we can assume that $m \leq kn$, so this can be upper bounded by $\mathcal{O}(3^{3k} \cdot k^2 \cdot n^2)$.

3.1.2 Related literature

The main idea of the Robertson-Seymour algorithm, that tree decompositions can be constructed in a top-down greedy fashion by finding balanced separators, has turned out to be very generally applicable in the context of approximating width parameters of graphs. All algorithms for approximating treewidth listed in Table 1.1, except the algorithms introduced in this thesis, build on this general template. Moreover, for width

parameters other than treewidth, the first algorithms for approximating them have often been adaptations of the Robertson-Seymour algorithm.

In particular, as we already mentioned, the subsequent FPT approximation algorithms for treewidth by [Belbasi and Fürer, 2021, 2022; Bodlaender et al., 2016a; Lagergren, 1996; Matoušek and Thomas, 1991; Reed, 1992] are based on the same top-down construction as the Robertson-Seymour algorithm, but with different subroutines for finding the balanced separator. The algorithms achieving subquadratic running time in n do this by occasionally using a separator that is balanced with respect to $V(G)$ instead of W, resulting in a recursion tree of depth $\mathcal{O}(\log n)$.

This construction is also the basis for polynomial-time approximation algorithms for treewidth [Amir, 2010; Bodlaender et al., 1995; Feige et al., 2008; Fomin et al., 2018]. Indeed, one can observe that the only part of the algorithm that takes superpolynomial time is that of finding a balanced separator of the set W. These approximation algorithms work by replacing this part by a polynomial-time approximation algorithm for finding balanced separators.

To turn to a slightly different context, the Robertson-Seymour algorithm was implemented in *parameterized logspace*, that is, in space $f(k)\log n$ and time $n^{f(k)}$ by Elberfeld et al. [2010]. Another adaptation was by Lokshtanov et al. [2017] to make the Robertson-Seymour algorithm to run, in some sense, in a canonical way, which served as the central ingredient of their algorithm to show that graph isomorphism is FPT parameterized by treewidth. This result was also adapted to parameterized logspace by Elberfeld and Schweitzer [2017].

In the context of other width parameters than treewidth, the algorithm of Oum and Seymour [2006] for approximating branchwidth of connectivity functions works also in a similar fashion as the Robertson-Seymour algorithm for treewidth, as well as the subsequent rankwidth approximation algorithms by Oum [2008b]. Before Oum and Seymour, similar ideas were used by Hliněný [2005] to give a 3-approximation algorithm running in time $f(k)\cdot n^3$ for branchwidth of matroids represented over finite fields. A construction similar to that of the Robertson-Seymour algorithm was used by Kim et al. [2018] to obtain a $2^{\mathcal{O}(k^2\log k)}n^2$ time 2-approximation algorithm for tree-cut width.

In the aforementioned settings of treewidth, rankwidth, branchwidth of connectivity functions, and tree-cut width, the cuts or separators of the decompositions we are seeking have submodularity properties. Perhaps surprisingly, the Robertson-Seymour algorithm extends even to settings without submodularity. In particular, it was adapted by Marx [2010a] for designing an XP approximation algorithm for a parameter called "fractional hypertreewidth". With a similar approach, XP approximation algorithms for

parameters called "α-treewidth" and "minor-matching hypertreewidth" were given by Yolov [2018], and subsequently improved by Dallard et al. [2022].

The top-down template of the Robertson-Seymour algorithm extends also to settings where we wish to compute a tree decomposition that may have large bags, but (1) the large bags are required to have a specific structure and (2) the intersections of adjacent bags, i.e., adhesions, are required to be small. In particular, the original case of a structure like this is the decomposition of minor-free graphs by Robertson and Seymour [2003], where the proof of obtaining the tree-like decomposition presented in [Robertson and Seymour, 1991] indeed follows an outline similar to the Robertson-Seymour algorithm, even though the original version was not presented as an algorithm. A subsequent algorithmic version was given by Kawarabayashi and Wollan [2011]. Similar ideas were applied for decomposing graphs excluding a topological minor by Grohe and Marx [2015], for decomposing graphs into "unbreakable" bags by Cygan et al. [2019], and for finding tree decompositions whose large bags must come from a specific graph class by Jansen et al. [2023].

3.2 Bodlaender's algorithm

Bodlaender [1996] gave a linear-time algorithm for computing optimum-width tree decompositions of bounded-treewidth graphs.

Theorem 3.6 (Bodlaender [1996])**.** *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k^3)}n$ either outputs a tree decomposition of G of width k, or determines that $\mathsf{tw}(G) > k$.*

This algorithm consists of two main ingredients. The first ingredient is an algorithm for computing optimum-width tree decompositions by dynamic programming, when given a tree decomposition of small but not optimal width. This was given by Bodlaender and Kloks [1996], and by Lagergren and Arnborg [1991].

Theorem 3.7 (Bodlaender and Kloks [1996]; Lagergren and Arnborg [1991])**.** *There is an algorithm that, given a graph G, a tree decomposition of G of width ℓ, and an integer $k < \ell$, in time $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}n$ outputs a tree decomposition of G of width k, if one exists.*[1]

At the time in 1991, Theorem 3.7 implied a $2^{\mathcal{O}(k^3)}n\log^2 n$ time exact algorithm for computing treewidth, by first running the algorithm of Lagergren [1996] (first appeared

[1]The dependence $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}$ on k and ℓ is stated by neither Bodlaender and Kloks [1996] nor Lagergren and Arnborg [1991], but is stated by Bodlaender [1996] and follows directly from the techniques presented in [Bodlaender and Kloks, 1996].

in [Lagergren, 1990]) to obtain an 8-approximate tree decomposition in $k^{\mathcal{O}(k)}n\log^2 n$ time, and then running the algorithm of Theorem 3.7 with this 8-approximate tree decomposition. A year later, this could have been replaced by the $k^{\mathcal{O}(k)}n\log n$ time 8-approximation algorithm of Reed [1992].

The second ingredient of Theorem 3.6 is a self-reduction scheme given by Bodlaender [1996], that states that any algorithm for computing treewidth that requires an approximately optimal tree decomposition as an input can be turned into an algorithm without this requirement, only with a small overhead in the running time. It was later also observed by Bodlaender et al. [2016a] that this self-reduction scheme works in the context of approximating treewidth as well. We will use this self-reduction scheme in multiple results of this thesis, so next we state it in a formal and very general form.

Theorem 3.8 (Bodlaender [1996])**.** *Let $\alpha \geq 1$ be a rational number. Suppose there is an algorithm $\mathcal{A}$, that given an n-vertex graph G, integer k, and a tree decomposition of G of width at most $2\cdot\alpha\cdot(k+1)-1$, in time $T_{\mathcal{A}}(k,n)$ either outputs a tree decomposition of G of width at most $\alpha\cdot(k+1)-1$ or determines that $\mathsf{tw}(G)>k$. Then there is an algorithm that in time $k^{\mathcal{O}(1)}\cdot(T_{\mathcal{A}}(k,n)+n)$ does the same, but without requiring a tree decomposition as an input. Moreover, if $\mathcal{A}$ runs in space $S_{\mathcal{A}}(k,n)$, then this algorithm runs in space $k^{\mathcal{O}(1)}n+S_{\mathcal{A}}(k,n)$.*

By inserting $\alpha=1$ and the algorithm of Theorem 3.7 as the algorithm $\mathcal{A}$, the combination of Theorems 3.7 and 3.8 implies Theorem 3.6. Theorem 3.8 was also used with $\alpha=5$ and $\alpha=3$ by Bodlaender et al. [2016a], and we use it with $\alpha=2$ in Chapter 4, and with $\alpha=1$ and $\alpha=1+\varepsilon$ in Chapter 5.

We will give an informal overview the proof of Theorem 3.7 in Subsection 3.2.1, and then give the full proof of Theorem 3.8 in Subsection 3.2.2. These proofs have also been reviewed, for example, by Kloks [1994], Pilipczuk [2020], and Downey and Fellows [2013]. The algorithms of Theorem 3.7 and Theorem 3.8 have been very influential in computing graph width parameters, so in Subsection 3.2.3 we briefly review the subsequent works inspired by them.

3.2.1 Bodlaender-Kloks dynamic programming

We now give an informal overview the proof of Theorem 3.7. Our presentation is based on that of Bodlaender and Kloks [1996].

Let $\mathcal{T}=(T,\mathsf{bag})$ be the given tree decomposition of width at most ℓ, and by Lemma 2.15 assume that $\mathcal{T}$ is nice. For a node $t\in V(T)$, we denote by $G_t=G[\mathsf{bags}_{\mathcal{T}}(\mathsf{desc}_T(t))]$

the graph induced by the bags of the descendants of t. The idea of the algorithm is to compute for each node $t \in V(T)$ a set of descriptions of tree decompositions of G_t of width at most k, so that the set for t can be computed given the sets for the children of t, and the set for the root node r is non-empty if and only if there exists a tree decomposition of $G_r = G$ of width at most k.

What should we remember about a tree decomposition $\mathcal{T}_0 = (T_0, \mathsf{bag}_0)$ of G_t to know whether it can be extended to a tree decomposition of the graph $G_{t'}$ for some ancestor t' of t? The first observation is, that if a vertex $v \in V(G_t)$ is not in $\mathsf{bag}(t)$, then it is not adjacent to any vertex outside of G_t, so we should have already determined whether edges incident to it satisfy the edge condition. Therefore, it is not important to remember the set $\mathsf{bag}_0(x) \setminus \mathsf{bag}(t)$ for any $x \in V(T_0)$, but in order to bound the width, it is important to remember the number $|\mathsf{bag}_0(x) \setminus \mathsf{bag}(t)|$ of such vertices in each bag of $\mathcal{T}_0$. In particular, instead of remembering (T_0, bag_0), it suffices to remember the triple $\mathcal{T}_1 = (T_1, \mathsf{bag}_1, |\mathsf{bag}_0|)$, where $T_1 = T_0$, bag_1 is the function that maps $x \in V(T_1)$ to $\mathsf{bag}_1(x) = \mathsf{bag}_0(x) \cap \mathsf{bag}(t)$, and $|\mathsf{bag}_0|$ the function mapping $x \in V(T_1)$ to $|\mathsf{bag}_0(x)|$.

Then we shall prune leaves of $\mathcal{T}_1$. Suppose that there is a leaf x of T_1 with parent p, so that $\mathsf{bag}_1(x) \subseteq \mathsf{bag}_1(p)$. Do we actually need to remember x? If we removed x, then $\mathcal{T}_1$ would still have the node p with $\mathsf{bag}_1(p) \supseteq \mathsf{bag}_1(x)$, and we know that we could take any subset $B \subseteq \mathsf{bag}_1(p)$ and create a new node adjacent to p with that B as its bag. If $|\mathsf{bag}_0(x)| > |\mathsf{bag}_1(x)|$, then this new bag with $B = \mathsf{bag}_1(x)$ would be even better than the bag of x, as it would not contain any forgotten nodes, and otherwise it would be identical to the bag of x. In either case, it is not useful to remember the node x or any information about its bag. Therefore, as long as T_1 has such a leaf x, we can remove x. Let us denote the tree resulting by pruning such leaves exhaustively by T_2, and let $\mathcal{T}_2 = (T_2, \mathsf{bag}_1 \restriction_{V(T_2)}, |\mathsf{bag}_0| \restriction_{V(T_2)})$.

Now $\mathcal{T}_2$ has at most $|\mathsf{bag}(t)|$ leaves, as each leaf bag must contain a vertex of $\mathsf{bag}(t)$ not contained in the bag of its parent, and thus by the connectedness condition, not contained anywhere else in $\mathcal{T}_2$. It follows that T_2 has at most $|\mathsf{bag}(t)| - 1$ nodes of degree more than 2. Let T_2' be the tree obtained by suppressing all degree-2 nodes of T_2. In particular, each edge of T_2' corresponds to a path of T_2 whose endpoints have degree not equal to 2 and whose internal nodes have degree equal to 2. Note that $|V(T_2')| \leq 2 \cdot |\mathsf{bag}(t)|$. We shall remember enough information about $\mathcal{T}_2$ by remembering T_2', and information about the triple $\mathcal{P} = (P, \mathsf{bag}_1 \restriction_{V(P)}, |\mathsf{bag}_0| \restriction_{V(P)})$ for every path P of T_2 corresponding to an edge of T_2'.

Let P be a path of T_2 corresponding to an edge of T_2'. First we divide P into segments based on the sets $\mathsf{bag}_1(x)$ for $x \in V(P)$. In particular, note that the connectedness

condition implies that there are at most $2 \cdot |\mathsf{bag}(t)|$ different sets $\mathsf{bag}_1(x)$ on P, as each vertex in $\mathsf{bag}(t)$ can be introduced and forgotten at most once as we travel P from one end to another. Let $P' = x_1, \ldots, x_p$ be a maximal subpath of P so that the sets $\mathsf{bag}_1(x_i)$ are equal to each other for all x_i. Now, Bodlaender and Kloks show that if there exists indices $i < j \in [p]$ so that either $|\mathsf{bag}_0(x_i)| \leq |\mathsf{bag}_0(x_h)| \leq |\mathsf{bag}_0(x_j)|$ for all $h \in [i, j]$ or $|\mathsf{bag}_0(x_i)| \geq |\mathsf{bag}_0(x_h)| \geq |\mathsf{bag}_0(x_j)|$ for all $h \in [i, j]$, then it is safe to suppress all nodes between x_i and x_j in P'. Intuitively, the reason is that if we imagine combining the decomposition with another decomposition in a join node, then, if $|\mathsf{bag}_0(x_i)| \leq |\mathsf{bag}_0(x_h)|$ for all $h \in [i, j]$, we could match all bags that are matched with bags between x_i and x_j with the bag of x_i. We also need to keep the bag of x_j to encode that in order to "slide" the other decomposition over x_j, one needs a small enough bag.

Let P'' be the path obtained from P' by exhaustively suppressing all nodes between such pairs x_i, x_j. The mapping from P' to P'' depends only on the integer sequence $|\mathsf{bag}_0(x_1)|, |\mathsf{bag}_0(x_2)|, \ldots, |\mathsf{bag}_0(x_p)|$, and indeed the corresponding integer sequence of bag sizes for P'' is called the *typical sequence* of the sequence $|\mathsf{bag}_0(x_1)|, \ldots, |\mathsf{bag}_0(x_p)|$ by Bodlaender and Kloks. They showed that the number of typical sequences of integers between 0 and k is at most $2^{\mathcal{O}(k)}$.

Now, to describe $\mathcal{P} = (P, \mathsf{bag}_1 \restriction_{V(P)}, |\mathsf{bag}_0| \restriction_{V(P)})$ it suffices to record the sequence of different sets $\mathsf{bag}_1(x_i)$ as we travel P from left to right, and for each of them the typical sequence of the numbers $|\mathsf{bag}_0(x_i)|$ on the corresponding subpath of P. As discussed, there are at most $2 \cdot |\mathsf{bag}(t)|$ different such sets $\mathsf{bag}_1(x_i)$ on $\mathcal{P}$, and by using the connectedness condition it can be shown that there are at most $|\mathsf{bag}(t)|^{\mathcal{O}(|\mathsf{bag}(t)|)}$ possible sequences of the bags $\mathsf{bag}_1(x_i)$. For each sequence we need to store up to $2 \cdot |\mathsf{bag}(t)|$ typical sequences, so in total there can be at most

$$|\mathsf{bag}(t)|^{\mathcal{O}(|\mathsf{bag}(t)|)} \cdot 2^{\mathcal{O}(k) \cdot 2 \cdot |\mathsf{bag}(t)|} \leq 2^{\mathcal{O}((k + \log \ell) \cdot \ell)}$$

different descriptions of such $\mathcal{P}$.

Now, to describe $\mathcal{T}_2 = (T_2, \mathsf{bag}_1 \restriction_{V(T_2)}, |\mathsf{bag}_0| \restriction_{V(T_2)})$, we write down the compressed tree T_2' with at most $\mathcal{O}(\ell)$ edges, and for each of its edges a description of the corresponding path in $\mathcal{T}_2$ as discussed above. The number of trees with $\mathcal{O}(\ell)$ edges is $\ell^{\mathcal{O}(\ell)}$ by Cayley's formula. Therefore, the total number of different descriptions of such $\mathcal{T}_2$ is

$$\ell^{\mathcal{O}(\ell)} \cdot 2^{\mathcal{O}((k + \log \ell) \cdot \ell) \cdot \mathcal{O}(\ell)} \leq 2^{\mathcal{O}((k + \log \ell) \cdot \ell^2)}.$$

Now, the dynamic programming records for each node t the set of all possible descriptions of tree decompositions of G_t of width at most k. We omit from this overview the

descriptions of the transitions of the dynamic programming, as well as how they are traversed backwards to actually return a tree decomposition of G of width at most k if one exists.

3.2.2 Bodlaender's self-reduction scheme

We then describe the self-reduction algorithm resulting in Theorem 3.8, given by Bodlaender [1996]. In particular, we will give a completely self-contained proof of Theorem 3.8, with slight differences compared to the original proof, but following the ideas of it.

Informally speaking, the main idea of Bodlaender's self-reduction algorithm is to compute, given a graph G, a graph G' such that,

1. the treewidth of G' is at most the treewidth of G,
2. any tree decomposition of G' of width at most k can be turned into a tree decomposition of G of width at most $2k+1$, and
3. $|V(G')| \leq |V(G)| \cdot (1 - 1/k^{\mathcal{O}(1)})$.

The algorithm proceeds by calling itself recursively on G' to obtain a tree decomposition $\mathcal{T}'$ of G', then lifts $\mathcal{T}'$ by using the property of Item 2 to a tree decomposition $\mathcal{T}$ of G, and finally runs the algorithm $\mathcal{A}$ to reduce the width of $\mathcal{T}$.

How is the graph G' obtained? The algorithm first computes a maximal matching M in G, and then analyses the following cases. First, if M is large, that is, of size roughly linear in $|V(G)|$, then G' is obtained from G by contracting all edges in M. In the other case when M is small, the vertices $V(M)$ of M form a small vertex cover of G, which can be used to either find a large number of vertices of G that can be "eliminated" without increasing the treewidth of G, i.e., removed and having their neighborhoods turned into cliques, or to conclude that G has large treewidth.

Let us then turn to a more formal description. A main concept of the algorithm is the *k-improved graph* of a graph G, denoted by $I_k(G)$. The graph $I_k(G)$ is the supergraph of G that is obtained by adding an edge between every two non-adjacent vertices u and v if there is a collection of more than k internally vertex-disjoint paths between u and v, i.e., if $\mathsf{flow}_G(N(u), N(v)) > k$. The motivation of this definition is the following lemma, stating that from the viewpoint of tree decompositions of width at most k, the graphs G and $I_k(G)$ behave the same way.

Lemma 3.9. *If (T, bag) is a tree decomposition of G of width at most k, then (T, bag) is also a tree decomposition of $I_k(G)$.*

Proof. Let (T, bag) be a tree decomposition of G of width at most k. We have to prove that if vertices u and v of G are non-adjacent but $\mathsf{flow}(N(u), N(v)) > k$, then there exists a bag of (T, bag) that contains both u and v. Let u,v be such a pair and suppose there does not exist such a bag. Let r be a node of (T, bag) with $u \in \mathsf{bag}(r)$ and let us root (T, bag) at r. Let also $t = \mathsf{forget}(v)$ be the forget-node of v. We have that $|\mathsf{adh}(t)| \leq k$ because $\mathsf{adh}(t) \subseteq \mathsf{bag}(t) \setminus \{v\}$. Moreover, $(\mathsf{cmp}(t), \mathsf{adh}(t), V(G) \setminus (\mathsf{cmp}(t) \cup \mathsf{adh}(t)))$ is a separation of G of order k, with $v \in \mathsf{cmp}(t)$ and $u \in V(G) \setminus (\mathsf{cmp}(t) \cup \mathsf{adh}(t))$ because $u \notin \mathsf{bag}(t)$ and u is in the root bag. Therefore, this separation contradicts that $\mathsf{flow}_G(N(u), N(v)) > k$. □

A vertex v of a graph G is called a *simplicial vertex* if $N(v)$ is a clique in G. We call v an *I_k-simplicial vertex* if $N(v)$ is a clique in $I_k(G)$. In the case when the maximal matching is small, the desired outcome of the algorithm will be a large collection of I_k-simplicial vertices. This will be used in the following way.

Lemma 3.10. *Let G be an n-vertex graph, k an integer, and $S \subseteq V(G)$ an independent set consisting of I_k-simplicial vertices of degree at most k in G. Let G' be the graph obtained from G by making $N(v)$ into a clique for every $v \in S$, and then removing S. It holds that*

1. *if $\mathsf{tw}(G) \leq k$, then $\mathsf{tw}(G') \leq \mathsf{tw}(G)$, and*
2. *any tree decomposition of G' of width at most k can be turned into a tree decomposition of G of width at most k in time $k^{\mathcal{O}(1)}n$.*

Proof. Let us first show Item 1. By Lemma 3.9, if $\mathcal{T}$ is a tree decomposition of G of width at most k, then $\mathcal{T}$ is also a tree decomposition of $I_k(G)$, and therefore by removing the vertices in S, we obtain a tree decomposition of $I_k(G) \setminus S$, which is also a tree decomposition of G' because $I_k(G) \setminus S$ is a supergraph of G'.

For Item 2, let (T, bag) be a tree decomposition of G'. By Lemma 2.12, and because S is an independent set, for every $v \in S$ there exists $t \in V(T)$ so that $N(v) \subseteq \mathsf{bag}(t)$. Therefore, we simply add a node t' adjacent to t with $\mathsf{bag}(t') = N[v]$. Because the degree of v is at most k, $|\mathsf{bag}(t')| \leq k + 1$, and therefore adding it does not increase the width of (T, bag) above k. Note that the proof of Lemma 2.12 gives an efficient algorithm for finding such a node t, in particular, if (T, bag) is rooted, then it can be taken as the node $\mathsf{forget}(u)$ that maximizes the depth among all $u \in N(v)$. □

Then we show that either a large matching or a large number of I_k-simplicial vertices can be found efficiently.

Lemma 3.11. *There is an algorithm that, given an n-vertex graph G and an integer $k \geq 1$, in time $k^{\mathcal{O}(1)}n$ either*

1. *determines that $\mathsf{tw}(G) > k$,*
2. *returns a matching with at least $n/\mathcal{O}(k^2)$ edges, or*
3. *returns an independent set $S \subseteq V(G)$ consisting of $\Omega(n)$ I_k-simplicial vertices of degree at most k.*

Proof. First, if the number of edges m of G is more than nk, we can by Lemma 2.9 return that $\mathsf{tw}(G) > k$. Then, we can in $\mathcal{O}(m) = \mathcal{O}(kn)$ time compute a maximal matching M in G. If $|M| \geq n/(8k^2)$, we conclude with the case 2. Otherwise, the vertices $V(M)$ of M form a vertex cover of G of size at most $n/(4k^2)$, i.e., every vertex v in the set $X = V(G) \setminus V(M)$ has $N(v) \subseteq V(M)$. Let $X' \subseteq X$ be the vertices in X with degree at most $4k$. We have that $|X| \geq n - n/(4k^2) \geq \frac{3}{4}n$, and because $m \leq nk$, we have $|X \setminus X'| \leq n/4$, implying that $|X'| \geq n/2$.

We define the graph G^M to be the graph with the vertex set $V(G^M) = V(M)$ and edge set obtained by having an edge between $u, v \in V(G^M)$ if either $uv \in E(G)$, or if there are more than k vertices $w \in X'$ with $\{u, v\} \subseteq N(w)$. Note that G^M is a subgraph of the k-improved graph $I_k(G)$. Because the vertices in X' have degree at most $4k$, the graph G^M can be be explicitly computed in time $k^{\mathcal{O}(1)}n$. Let $X'' \subseteq X'$ be the vertices in X' whose neighborhood is a clique in G^M. Because G^M is a subgraph of $I_k(G)$, the vertices in X'' are I_k-simplicial. If $|X''| \geq n/4$ and all vertices in X'' have degree at most k, we return X''.

We claim that otherwise $\mathsf{tw}(G) > k$. First, if X'' contains a vertex v of degree at least $k+1$, then $N[v]$ is a clique of size at least $k+2$ in $I_k(G)$, and therefore $\mathsf{tw}(I_k(G)) > k$, implying $\mathsf{tw}(G) > k$.

It remains to show that if $|X''| < n/4$, then $\mathsf{tw}(G) > k$. For every vertex $v \in X' \setminus X''$, there exists a pair xy of distinct vertices of G^M so that $\{x, y\} \subseteq N(v)$ but $xy \notin E(G^M)$. In particular, xy is a "reason" why the neighborhood of v is not a clique in G^M. Let us fix arbitrarily a mapping $r\colon X' \setminus X'' \to \binom{V(G^M)}{2}$ that assigns one such reason for every vertex in $X' \setminus X''$. We observe that the graph $G^M_\star$ obtained by adding the edges $r(v)$ for all $v \in X' \setminus X''$ to G^M is a minor of the graph $I_k(G)$ because the edge $r(v)$ can be created by contracting an edge incident to v. Therefore, if $|E(G^M_\star)| > k \cdot |V(G^M_\star)|$, then $\mathsf{tw}(I_k(G)) > k$ and thus $\mathsf{tw}(G) > k$.

Now assume $|E(G^M_\star)| \leq k \cdot |V(G^M_\star)| \leq n/(4k)$. Because $|X' \setminus X''| > n/4$, by the pigeonhole principle there must be $xy \in E(G^M_\star)$ so that there are more than k vertices $v \in X' \setminus X''$ so that $r(v) = xy$. However, then xy should be an edge of G^M, and not be the reason $r(v)$ for any vertex, which is a contradiction. □

We are now ready to prove Theorem 3.8, which we restate here. We remark that in the proof we make the assumption that the running time function $T_{\mathcal{A}}(k,n)$ of the given algorithm $\mathcal{A}$ is, for any fixed k, non-decreasing and convex in n. This assumption holds for any reasonable running time bound.

Theorem 3.8 (Bodlaender [1996])**.** *Let $\alpha \geq 1$ be a rational number. Suppose there is an algorithm $\mathcal{A}$, that given an n-vertex graph G, integer k, and a tree decomposition of G of width at most $2 \cdot \alpha \cdot (k+1) - 1$, in time $T_{\mathcal{A}}(k,n)$ either outputs a tree decomposition of G of width at most $\alpha \cdot (k+1) - 1$ or determines that $\mathsf{tw}(G) > k$. Then there is an algorithm that in time $k^{\mathcal{O}(1)} \cdot (T_{\mathcal{A}}(k,n) + n)$ does the same, but without requiring a tree decomposition as an input. Moreover, if $\mathcal{A}$ runs in space $S_{\mathcal{A}}(k,n)$, then this algorithm runs in space $k^{\mathcal{O}(1)}n + S_{\mathcal{A}}(k,n)$.*

Proof. Let $k_\alpha = \alpha \cdot (k+1) - 1$. We describe a recursive procedure, which is always called with a graph G_r, so that $\mathsf{tw}(G_r) \leq k$ if $\mathsf{tw}(G) \leq k$, where G is the original input graph. Each recursive call either determines that $\mathsf{tw}(G_r) > k$ (and thus also $\mathsf{tw}(G) > k$), or returns a tree decomposition of G_r of width at most k_α.

The base case of the recursion is an edgeless graph with treewidth 0, in which case we can return in $\mathcal{O}(n)$ time a trivial tree decomposition of width 0.

Otherwise, in the start of each recursive call we use the algorithm of Lemma 3.11 with the inputs G_r and k_α. If it returns that $\mathsf{tw}(G_r) > k_\alpha$, we can return $\mathsf{tw}(G) > k$ immediately. If it returns an independent set S of at least $|S| \geq \Omega(|V(G_r)|)$ I_{k_α}-simplicial vertices of degree at most k_α, we let G'_r to be the graph obtained from G_r by making the neighborhoods of vertices in S cliques and then removing S. By Lemma 3.10, if $\mathsf{tw}(G_r) \leq k$, then $\mathsf{tw}(G'_r) \leq k$. We call our procedure recursively with the graph G'_r, and if it returns a tree decomposition of G'_r of width at at most k_α, we use the algorithm of Lemma 3.10 to turn it into a tree decomposition of G_r of width at most k_α in time $k^{\mathcal{O}(1)}n$, and return it. If it returns that $\mathsf{tw}(G'_r) > k$, we can return $\mathsf{tw}(G_r) > k$.

In the case when the algorithm of Lemma 3.11 returns a matching M of size $|M| \geq |V(G)|/\mathcal{O}(k^2)$, we contract the edges in M to obtain a graph G^M_r and call the algorithm recursively on G^M_r. As contracting edges does not increase treewidth, the treewidth of G^M_r is at most the treewidth of G_r. Also, we can obtain a tree decomposition $\mathcal{T}$ of G_r

of width at most $2k_\alpha + 1$ from a tree decomposition $\mathcal{T}^M$ of G_r^M of width at most k_α by expanding the bags according to the matching, in particular, by replacing each occurrence of a vertex $w_{uv} \in V(G_r^M)$ corresponding to a contracted edge $uv \in M$ by the vertices u and v of G^M. Then we use the algorithm $\mathcal{A}$ with $\mathcal{T}$ to either get a tree decomposition of width at most k_α or to determine that the treewidth of G_r is larger than k.

When we call the algorithm recursively, it holds that $|V(G'_r)| \leq (1 - \Omega(1)) \cdot |V(G_r)|$ or $|V(G_r^M)| \leq (1 - 1/\mathcal{O}(k^2)) \cdot |V(G_r)|$. Therefore, in each recursive call the number of vertices is multiplied by a factor of at most $1 - 1/\mathcal{O}(k^2)$, and therefore the total running time can be expressed as a recurrence $T(k, n) = k^{\mathcal{O}(1)}n + T_{\mathcal{A}}(k, n) + T(k, n - n/\mathcal{O}(k^2))$, which can be bounded by $k^{\mathcal{O}(1)} \cdot (T_{\mathcal{A}}(k, n) + n)$ by using the facts that $T_{\mathcal{A}}(k, n)$ is non-decreasing and convex in n for fixed k. □

3.2.3 Related literature

The *pathwidth* of a graph is defined like treewidth, but with the restriction that the tree T in the decomposition (T, bag) must be a path, in which case it is called a *path decomposition*. Bodlaender and Kloks [1996] gave also an algorithm, that given a tree decomposition of width ℓ, in time $2^{\mathcal{O}((k+\log \ell)\cdot \ell)}n$ outputs a path decomposition of width at most k if one exists. Either combining this with $2^{\mathcal{O}(k)}n$ time constant-approximation of treewidth (Bodlaender et al. [2016a] or Theorem 1.1), or observing that the proof of Theorem 3.8 can be adapted to the setting of pathwidth, yields a $2^{\mathcal{O}(k^2)}n$ time algorithm for computing optimum-width path decompositions.

Bodlaender and Thilikos [1997] extended the ideas of Theorem 3.7 to give a $f(k) \cdot n$ time algorithm, for some computable function f, for constructing optimum-width branch decompositions of graphs, parameterized by the branchwidth k.

The parameter cutwidth (also known as "minimum cut linear arrangement"), asks for a linear ordering $v_1, \ldots, v_n$ of the vertices that minimizes the maximum number of edges $|E(G[\{v_1, \ldots, v_i\}, \{v_{i+1}, v_n\}])|$ cut by cutting the graph G along the ordering. Using techniques similar to Theorem 3.7, Thilikos et al. [2005] gave a $f(k) \cdot n$ time algorithm, for some computable function f, for constructing an ordering of width at most k, if one exists. An alternative $f(k) \cdot n$ time algorithm for cutwidth was given by Giannopoulou et al. [2019]. A linear-time FPT algorithm was also given for a parameter called "linear-width" by Bodlaender and Thilikos [2004]. Unifying frameworks for algorithms inspired by the algorithm of Bodlaender and Kloks [1996] were given by Bodlaender et al. [2009] and Soares [2013].

Later, Jeong et al. [2021] further developed the ideas of Bodlaender and Kloks [1996] to give an algorithm for constructing optimum-width branch decompositions of connectivity functions that can be expressed in a certain linear-algebraic way with subspaces of vector spaces over finite fields. This framework captures the branchwidth of linear matroids over finite fields, the rankwidth of graphs, the branchwidth of graphs and hypergraphs, and the carving width of graphs, yielding $f(k) \cdot n^3$ time algorithms for computing optimum-width decompositions for each of them. Similar result for the path-like variants of these problems were given earlier by the same authors [Jeong et al., 2017].

Bojańczyk and Pilipczuk [2022] showed, building on their earlier result [Bojanczyk and Pilipczuk, 2016], that in a certain sense, optimum-width tree decompositions of graphs can be encoded in monadic second-order logic. Their techniques give an alternative viewpoint of the dynamic programming of Bodlaender and Kloks [1996], and we make use of this in Chapter 6. Also, Bodlaender et al. [2023] gave additional structural insights on the technique of typical sequences used in the algorithm of Bodlaender and Kloks [1996], leading to a polynomial-time algorithm for cutwidth of series parallel directed graphs.

Perković and Reed [2000] gave a version of Theorem 3.6 that in the case when $\mathsf{tw}(G) > k$ returns a subgraph G' of G with $\mathsf{tw}(G') > k$ and a tree decomposition of G' of width at most $2k$. This was applied by Kawarabayashi et al. [2012] to give $f(k) \cdot n^2$ and $f(H) \cdot n^2$ time algorithms for the k-disjoint paths and H-minor-containment problems.

3.3 Applications

In this section we review some applications of graph width parameters in different contexts. We start by reviewing the role and applications of treewidth in the Graph Minors series of Robertson and Seymour in Subsection 3.3.1. In Subsection 3.3.2 we review the monadic second-order logic of graphs, and the role of treewidth and cliquewidth in it. The topic of Subsection 3.3.3 is the various algorithmic applications of treewidth. Finally, in Subsection 3.3.4 we discuss some results related to width parameters with a computational complexity flavor. The overarching theme of this section is to convince the reader that treewidth, and sometimes cliquewidth and rankwidth, is a fundamental parameter to study, and in many settings, *the right* parameter to study.

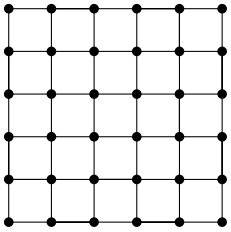

Figure 3.2: The 6×6 grid graph.

3.3.1 Graph Minors

Treewidth was introduced in the Graph Minors series of Robertson and Seymour, and many of the most significant applications of treewidth stem from there. The Graph Minors series spans 23 articles and contains several fundamental results, most of which are either directly related to treewidth, or use treewidth as an ingredient in the proof. Perhaps the most famous result of the series is the resolution of Wagner's conjecture.

Theorem 3.12 (Robertson and Seymour [2004])**.** *If $G_1, G_2, \ldots$ is an infinite sequence of graphs, then there exists $i \neq j$ so that G_i is a minor of G_j.*

The Grid Minor Theorem

While treewidth does play a role in the proof of Theorem 3.12, its significance becomes more clear in the special case of Theorem 3.12 where the graphs $G_1, G_2, \ldots$ are planar, or contain even a single planar graph. The proof of this special case has two ingredients, of which the first one is the *Grid Minor Theorem.*

Theorem 3.13 (Robertson and Seymour [1986b])**.** *There is a function $f\colon \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$, so that if a graph G does not contain a planar graph H as a minor, then $\mathsf{tw}(G) \leq f(|V(H)|)$.*

This theorem is called the "Grid Minor Theorem" because the general case presented in Theorem 3.13 is quite easily derived from the special case where the graph H is a square grid. See Figure 3.2 for an illustration of grid graphs. The significance of grids is also that the $k \times k$ grid has treewidth k, so Theorem 3.13 can be seen as saying that treewidth of a graph G is "functionally equivalent" to the largest k such that G contains a $k \times k$ grid as a minor, in the sense that $k \leq \mathsf{tw}(G) \leq f(k)$ for some function f. Note that Theorem 3.13 fails for every non-planar graph H, because planar graphs do not contain H as a minor but have unbounded treewidth, as shown by the grids.

The function f originally given by Robertson and Seymour [1986b] was very fast-growing, but after a series of improvements [Leaf and Seymour, 2015; Robertson et al., 1994],

Chekuri and Chuzhoy [2016] proved a polynomial upper bound for f. The current best upper bound for f, when H is the $k \times k$ grid, is $\mathcal{O}(k^9 \cdot \operatorname{poly}\log k)$ by Chuzhoy and Tan [2021]. It has been conjectured by Robertson et al. [1994] that the best possible bound $\Theta(k^2 \log k)$ would be the right answer, and by Demaine et al. [2009] that the correct bound would be $\Theta(k^3)$.

The second ingredient of the proof of the special case of Theorem 3.12 is to show that it holds if there is an upper bound for the treewidth of the graphs in the sequence, which was shown by Robertson and Seymour [1990]. Now, the Grid Minor Theorem implies that if $G_1, G_2, \ldots$ is an infinite sequence of graphs so that there does not exists any $i \neq j$ such that G_i is a minor of G_j, and one graph in the sequence is planar, then there is an upper bound for the treewidth of all graphs in this sequence, particularly $f(|V(G_\ell)|)$ if G_ℓ is planar. This implies Theorem 3.12 in that case.

It is not hard to think of algorithmic applications of the Grid Minor Theorem. A direct application is an FPT algorithm for deciding if a graph G contains a planar graph H as a minor, parameterized by $|V(H)|$. By Theorem 3.13, we may immediately answer yes if the treewidth of G is more than $f(|V(H)|)$. If the treewidth of G is less than $f(|V(H)|)$, we may solve the problem by first using an FPT algorithm for computing a tree decomposition, and then applying dynamic programming on the tree decomposition. If the algorithm for computing a tree decomposition runs in time linear in $n = |V(G)|$, then this gives an algorithm with running time $g(|V(H)|) \cdot n$, for some function g that depends on the function f, the treewidth algorithm, and the details of the dynamic programming.

There are several variants and enhancements of the Grid Minor Theorem. In the context when G is planar or minor-free, linear upper bounds for the function f have been given by [Demaine and Hajiaghayi, 2008; Gu and Tamaki, 2012; Kawarabayashi and Kobayashi, 2020; Robertson et al., 1994], with algorithmic applications that will be discussed in Subsection 3.3.3.

Geelen et al. [2023] gave an analogue of the Grid Minor Theorem for rankwidth, showing that graphs that exclude a "circle graph" as a "vertex-minor" have bounded rankwidth. Both vertex-minors and circle graphs have technical definitions that we omit here. Oum [2009] has conjectured a similar result about "pivot-minors" and bipartite circle graphs, which would be stronger in that it would directly imply both Theorem 3.13 and the aforementioned result about vertex-minors.

A version of the Grid Minor Theorem for directed graphs has been given by Kawarabayashi and Kreutzer [2015]. It uses analogues of treewidth, grids, and minors for directed graphs that were proposed by Johnson et al. [2001]. The author showed that "minor" in the Grid

Minor Theorem can be replaced by "induced minor", which is like minor but not allowing edge deletions, if the graph G has bounded degree [Korhonen, 2023]. The conditions on when such "Grid Induced Minor Theorem" holds have been further investigated by Alecu et al. [2023].

The k-disjoint paths and H-minor-containment problems

In the k-disjoint paths problem we are given a graph G and k pairs of terminal vertices $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$, and the problem is find a collection $P_1, \ldots, P_k$ of vertex-disjoint paths, each P_i with endpoints s_i and t_i. The H-minor-containment problem asks if G contains the graph H as a minor. The main algorithmic results of the Graph Minors series are FPT algorithms for the k-disjoint paths problem parameterized by k and for the H-minor-containment problem parameterized by $|V(H)|$.

Theorem 3.14 (Robertson and Seymour [1995])**.** *There is a computable function f, so that there is an $f(k) \cdot n^3$ time algorithm for k-disjoint paths, and an $f(|V(H)|) \cdot n^3$ time algorithm for H-minor-containment.*

We stated both of the algorithms in the same theorem because Robertson and Seymour [1995] in fact give an algorithm that solves a common generalization of the two problems, called the "folio" problem. For their algorithm, Robertson and Seymour [1995] introduced the *irrelevant vertex technique*. They showed that if a graph G has high treewidth (compared to k or $|V(H)|$), then it is possible to locate an *irrelevant vertex* in G, whose removal from G does not change the answer to the problem. Then, their algorithm works by repeatedly removing irrelevant vertices, until the treewidth of G is bounded by a function of the parameter, at which point the problem can be solved by dynamic programming using treewidth.

The irrelevant vertex argument of Robertson and Seymour [1995] is divided into two cases, depending on whether G contains a large clique as a minor. If G contains a large clique minor, identifying an irrelevant vertex based on the clique minor is relatively easy. When G does not contain a large clique minor but has large treewidth, their argument is based on the *Flat Wall Theorem*, which is a version of the Grid Minor Theorem where the grid is laid out in a planar-like piece of the graph. In this case, they use long and complicated rerouting arguments, given in [Robertson and Seymour, 2012], to show that any vertex central enough in the grid is irrelevant.

The combination of Theorems 3.12 and 3.14 yields a very general algorithmic result. Theorem 3.12 implies that if $\mathcal{C}$ is a *minor-closed* class of graphs, that is, any minor of a graph in $\mathcal{C}$ is also in $\mathcal{C}$, then there is a finite collection of *obstructions* for $\mathcal{C}$. In particular,

the set of graphs that are not in $\mathcal{C}$ but whose every (strict) minor is in $\mathcal{C}$ must be finite by Theorem 3.12. To test if $G \in \mathcal{C}$, one only needs to test for every obstruction H whether H is a minor of G, and this can be done with the algorithm of Theorem 3.14. It follows that for every minor-closed graph class $\mathcal{C}$, there exists an algorithm for testing if $G \in \mathcal{C}$ with running time $\mathcal{O}(n^3)$. The constant hidden by the $\mathcal{O}$-notation of course depends on the class $\mathcal{C}$, and the result shows only the existence of an algorithm, not how to construct it, because there is no general method for finding the obstructions.

The dependence on n in the algorithms of Robertson and Seymour [1995] has been improved to n^2 by Kawarabayashi et al. [2012]. In the case when H (or G) is planar, a $f(|V(H)|) \cdot n$ time algorithm for H-minor-containment follows from the Grid Minor Theorem as we discussed earlier. A $f(k) \cdot n$ time algorithm for k-disjoint paths on planar graphs was given by Reed et al. [1993] (see also [Reed, 1995]). Several authors have improved the function $f(k)$ on planar k-disjoint paths [Adler et al., 2017; Lokshtanov et al., 2020a], culminating in the $2^{\mathcal{O}(k^2)}n$ time algorithm by Cho et al. [2023]. Adler et al. [2012] gave a $2^{\mathcal{O}(|V(H)|)}n + \mathcal{O}(n^2 \log n)$ time algorithm for H-minor-containment when G is planar.

After Robertson and Seymour, the irrelevant vertex technique has been applied to several problems beyond Theorem 3.14. To name a few, Grohe [2004] used the irrelevant vertex technique to show that the problem of finding planar drawings with the minimum number of crossings is FPT parameterized by the number of crossings. Cygan et al. [2013] showed that the k-disjoint paths problem is FPT on directed planar graphs. Baste et al. [2023] used irrelevant vertices inside dynamic programming to show that for every minor-closed graph class $\mathcal{C}$ there exists a $2^{\mathcal{O}(\mathsf{tw}(G) \log \mathsf{tw}(G))}n$ time algorithm that, given an input graph G, finds a minimum-size set $X \subseteq V(G)$ so that $G \setminus X \in \mathcal{C}$.

3.3.2 Monadic second-order logic of graphs

Courcelle's theorem [1990] gives a semi-automatic way of obtaining dynamic programming algorithms on tree decompositions. All that one needs to do is to express the problem in monadic second-order logic (MSO). For example, the 3-coloring problem can be expressed in MSO by first defining a formula that tells if a set of vertices $X \subseteq V(G)$ is an independent set as

$$\mathrm{Ind}(X) = \forall_{u,v \in V(G)} (\neg \mathsf{adj}(u, v)) \vee (\neg u \in X) \vee (\neg v \in X)$$

and then defining a sentence φ expressing 3-colorability as

$$\varphi = \exists_{X_1,X_2,X_2 \subseteq V(G)} \operatorname{Ind}(X_1) \wedge \operatorname{Ind}(X_2) \wedge \operatorname{Ind}(X_3) \wedge (\forall_{v \in V(G)} v \in X_1 \vee v \in X_2 \vee v \in X_3).$$

Now, φ is *true in* a graph G if and only if G is 3-colorable, which is equivalent to the fact that there exists three independent sets $X_1, X_2, X_3 \subseteq V(G)$ so that every vertex of G is in at least one of them. Note that it is not necessary to define that the sets X_1, X_2, and X_3 are disjoint, but this could also be defined in the sentence φ.

In the *monadic second-order logic* of graphs we can use the usual logical connectives $\neg$, $\vee$, and $\wedge$, the equality $=$ of vertices or edges, the quantifiers $\exists$ and $\forall$ that quantify over single vertices or edges, or sets of vertices or edges, the binary predicate $\in$ that tells if a vertex belongs to a set of vertices or an edge belongs to a set of edges, the binary predicate adj for telling if two vertices are adjacent, and the binary predicate inc for telling if an edge is incident to a vertex. The *length* of an monadic second-order logic formula is the number of symbols appearing in it.

There are in fact several variants of monadic second-order logic of graphs, and the one described above is referred to as MSO_2. In a slightly more general variant, called *counting monadic second-order logic* and referred to as CMSO_2, there exists for all constants $p, q \in \mathbb{Z}_{\geq 0}$ a formula $\mathsf{card}_{p,q}(X)$ that tells if the cardinality of a set X is p modulo q. Now we can state Courcelle's theorem as follows.

Theorem 3.15 (Courcelle [1990])**.** *There is an algorithm, that given a graph G, a tree decomposition of G of width k, and a CMSO_2 sentence φ, in time $f(k,\varphi) \cdot n$ returns whether φ is true in G, where $f(k,\varphi)$ is a computable function.*

As many graph properties can be expressed in CMSO_2 sentences φ of fixed size, Courcelle's theorem, together with algorithms for computing treewidth, imply the fixed-parameter tractability of deciding these properties parameterized by treewidth. Classical examples of such properties are Hamiltonicity and c-colorability for fixed c, although for them there exists also simple hand-crafted dynamic programming algorithms that are much more efficient than what follows from Courcelle's theorem (e.g. [Bodlaender, 1988]). However, Courcelle's theorem has been used in applications where designing the dynamic programming algorithm by hand would be overwhelmingly complicated, such as the aforementioned crossing number algorithm of Grohe [2004].

Although Courcelle's theorem is formulated only for decision problems, the techniques behind it extend also to optimization and counting problems. Indeed, theorems similar to Theorem 3.15 but for optimization and counting were given by Arnborg et al. [1991] and Borie et al. [1992] (the latter independently of Courcelle and the former). Also, there

has been several approaches to general theorems for dynamic programming algorithms on tree decompositions akin to Courcelle's, but with a more reasonable running time dependence on treewidth [Pilipczuk, 2011; Telle and Proskurowski, 1997].

Cliquewidth and CMSO_1

Cliquewidth[2] was introduced by Courcelle et al. [1993] to be an analogue of treewidth for a variant of CMSO_2 called CMSO_1. A CMSO_1 formula is like a CMSO_2 formula, but without the possibility to talk about the edges of the graph. In particular, we cannot quantify over sets of edges, do not have the vertex-edge incidence predicate, and cannot use the $\mathsf{card}_{p,q}(X)$ formula on sets of edges. However, we still have the adj predicate describing adjacencies between vertices. We can still express 3-coloring in CMSO_1, in particular, our earlier example sentence φ is in fact an MSO_1 sentence, but for example Hamiltonicity cannot be expressed in CMSO_1 [Courcelle and Engelfriet, 2012, Chapter 5].

Another way of thinking about CMSO_1 and CMSO_2 is that in CMSO_2 graphs are encoded as a logical structure with an universe $V(G) \cup E(G)$, with edges described by the vertex-edge incidence predicate inc. In CMSO_1 the universe is $V(G)$, and edges are described by the vertex-vertex adjacency predicate adj. The adj predicate can also be assumed to be available in CMSO_2 because it can be written using the inc predicate as

$$\mathsf{adj}(u, v) = (u \neq v) \wedge \exists_{e \in E(G)} \mathsf{inc}(u, e) \wedge \mathsf{inc}(v, e).$$

The "Courcelle's theorem for cliquewidth" was already implicit in the early work of Courcelle [1995], but was explicitly given by Courcelle et al. [2000].

Theorem 3.16 (Courcelle et al. [2000])**.** *There is an algorithm that, given a k-expression of a graph G and a CMSO_1 sentence φ, in time $f(k, \varphi) \cdot n$ returns whether φ is true in G, where $f(k, \varphi)$ is a computable function. Furthermore, this algorithm extends to an optimization variant of CMSO_1.*

As discussed earlier, bounded cliquewidth is a significant generalization of bounded treewidth, so Theorem 3.16 is a significant generalization of Theorem 3.15 for problems expressible in CMSO_1. Note that even though a graph G with cliquewidth 2 can have $\Theta(n^2)$ edges, the algorithm of Theorem 3.16 runs in time linear in n, because it takes as an input a k-expression of G, not the graph G. Here we assume that the k-expression has length at most $f(k) \cdot n$, and this is justified by the discussion in Subsection 2.3.5. Unlike

[2]In this subsection, we will exclusively talk about cliquewidth instead of rankwidth, but all the results hold even when cliquewidth is replaced by rankwidth.

for treewidth and Theorem 3.15, it is still an open question if there exists a version of Theorem 3.16 that runs in time linear in the size of G for fixed k and φ, for graphs of cliquewidth k given without a k-expression. As mentioned earlier, we make significant progress on this question in Chapter 7, improving the dependence on n from n^3 to n^2. Theorem 3.16 was extended also to counting problems by Courcelle et al. [2001].

The more profound aspects of MSO

So far we have treated the monadic second-order logic of graphs simply as a general tool for constructing algorithms. However, the theory is deeper than that, and motivates the notions of treewidth and cliquewidth by showing that they are *the right* parameters for CMSO_2 and CMSO_1, respectively.

One consequence of the proof of Theorem 3.15 is that the CMSO_2-theory of graphs of bounded treewidth is *decidable*. This means that there exists an algorithm, that given a CMSO_2-sentence φ and an integer k, tests if φ is true in all graphs of treewidth at most k [Courcelle, 1990]. Roughly speaking, this follows from the fact that the proof of Theorem 3.15 gives a finite-state "tree automaton" processing tree decompositions of width at most k, and one can analyze all possible runs of this automaton. The converse of this was proven by Seese [1991]. He showed that if $\mathcal{C}$ is a class of graphs with unbounded treewidth, then the MSO_2-theory of $\mathcal{C}$ is undecidable. This indicates that the role of treewidth in the proof of Theorem 3.15 cannot be exchanged for any other graph parameter, unless that parameter is bounded by a function of treewidth.

Similar decidability result holds also for cliquewidth and CMSO_1 [Courcelle, 1995]. Seese [1991] conjectured that the converse also holds in that setting. Courcelle and Oum [2007] showed that if $\mathcal{C}$ is a class of graphs with unbounded cliquewidth, then the CMSO_1-theory of $\mathcal{C}$ is undecidable. This is indeed a converse of the decidability result, but the original conjecture of Seese was about MSO_1 instead of CMSO_1, which remains open.

Another piece of evidence pointing that graphs of bounded treewidth are exactly the tree-like graphs from the viewpoint of MSO_2 and graphs of bounded cliquewidth are exactly the tree-like graphs from the viewpoint of MSO_1 is given by MSO *transductions*, introduced in different forms by Arnborg et al. [1991] and Courcelle [1991] (see [Courcelle and Engelfriet, 2012] for a modern exposition). Informally speaking, we say that a graph G can be MSO_1 *interpreted* in a graph H with $V(H) = V(G)$ if we can write an MSO_1 formula $\mathsf{adj}_G(u, v)$ that based on the graph H tells if uv is an edge of G. For example, if we write $\mathsf{adj}_G(u, v) = \neg\mathsf{adj}(u, v)$, then G is the complement of H. In this way, we can

view the formula $\mathsf{adj}_G(u, v)$ as a function that maps the input graph H to the output graph G, and extend it to map graph classes to graph classes, in this example, class $\mathcal{C}$ to the class of complements of graphs in $\mathcal{C}$. MSO_1 transductions map graph classes to graph classes roughly like this, but with some additional features such as making copies of vertices and guessing a coloring that can be talked about in the formula. MSO_2 transductions are similar, but with the incidence predicate instead of adjacency. Courcelle and Engelfriet [1995] showed that a class of graphs $\mathcal{C}$ has bounded treewidth if and only if it is a subclass of an MSO_2 transduction from the class of trees, and bounded cliquewidth if and only if it is a subclass of an MSO_1 transduction from the class of trees.

Finally, let us discuss a result that instead of showing that treewidth is the right parameter for CMSO_2, shows that CMSO_2 is the right logic for dynamic programming on tree decompositions. As we mentioned earlier, the proof of Courcelle [1990] not only gives an algorithm, but shows that there exists an algorithm of a specific form. In particular, the algorithm can be formulated as dynamic programming on binary tree decompositions, that has $f(k, \varphi)$ states per node, and the state of a node depends only on the states of its children and the bag of the node. Courcelle [1990] conjectured also a converse of this, that every property of graphs that can be recognized by such finite-state dynamic programming on bounded-width tree decompositions can be defined in CMSO_2. This conjecture was proven by Bojanczyk and Pilipczuk [2016]. Bojanczyk et al. [2021] asked whether an analogous statement holds for cliquewidth and CMSO_1, and proved it in the special case of *linear cliquewidth*, which is the path-like version of cliquewidth.

We also briefly note that Courcelle's theorem has been extended to the setting of the monadic second-order logic of matroids and matroid branchwidth by Hliněný [2006]. Hliněný and Seese [2006] gave also an analogue of Seese's theorem in this setting.

3.3.3 Algorithms

In this subsection we survey some algorithmic applications of treewidth, rankwidth, and cliquewidth that we did not already mention in the previous two subsections.

Dynamic programming on tree decompositions

The fundamental application of treewidth is of course dynamic programming on tree decompositions of small width. Although this topic was already thoroughly explored in the end of the 1980s and beginning of 1990s (e.g. [Arnborg and Proskurowski, 1989; Arnborg et al., 1991; Bodlaender, 1988; Borie et al., 1992; Courcelle, 1990; Telle and

Proskurowski, 1997]), many techniques for nailing down the best dependence on the width k were discovered in the last 15 years. Let us mention a few of them.

Consider first the minimum dominating set problem, which asks for a minimum-size set of vertices $X \subseteq V(G)$ so that $N[X] = V(G)$. Telle and Proskurowski [1997] showed that minimum dominating set can be solved in time $9^k k^{\mathcal{O}(1)} n$ when given a tree decomposition of width k, and Alber and Niedermeier [2002] improved this to $4^k k^{\mathcal{O}(1)} n$. Both of these algorithms were relatively straightforward, unlike the next improvement to $3^k k^{\mathcal{O}(1)} n$ by van Rooij et al. [2009], which used a technique called "fast subset convolution", introduced by Björklund et al. [2007] (see also [Björklund et al., 2009]). With fast subset convolution, van Rooij et al. [2009] also improved the dependence on k for many other problems in the framework of "$[\rho, \sigma]$-domination problems" of Telle and Proskurowski [1997].

Lokshtanov et al. [2018b] introduced a framework for showing the optimality of the dependence on k in dynamic programming algorithms on tree decompositions, assuming the SETH. They showed that there is no $(2-\varepsilon)^k n^{\mathcal{O}(1)}$ time algorithm for maximum independent set nor a $(3-\varepsilon)^k n^{\mathcal{O}(1)}$ time algorithm for minimum dominating set when given a tree decomposition of width k, for any $\varepsilon > 0$, and several other lower bounds, assuming the SETH. In particular, the aforementioned dynamic programming algorithm for minimum dominating set using fast subset convolution is optimal. Lower bounds forbidding $2^{o(k \log k)} n^{\mathcal{O}(1)}$ time algorithms on tree decompositions of width k for problems like disjoint paths and chromatic number were given by Lokshtanov et al. [2018a].

For a long time it was not known whether the relatively straightforward $2^{\mathcal{O}(k \log k)} n$ time dynamic programming for connectivity problems such as Hamiltonian path, feedback vertex set, and connected dominating set would be optimal. This was shown to not be the case by Cygan et al. [2022][3], who gave $2^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ time algorithms for these and several other "connectivity problems". Their algorithm was rather atypical for an algorithm parameterized by treewidth, being randomized and having superlinear dependence on the number of vertices n. These defects were however fixed by Bodlaender et al. [2015], who obtained deterministic $2^{\mathcal{O}(k)} n$ time algorithms for these problems using a different linear-algebra based approach. Another alternative approach to obtain results similar to Bodlaender et al. [2015] was given by the "representative sets" technique that was introduced by Fomin et al. [2016] and further improved by Fomin et al. [2017].

[3]The conference version of [Cygan et al., 2022] appeared in 2011 [Cygan et al., 2011].

Dynamic programming on k-expressions and rank decompositions

We then turn to cliquewidth and rankwidth. In addition to the FPT results for CMSO_1 problems parameterized by cliquewidth by Courcelle et al. [2000], XP algorithms parameterized by cliquewidth for problems not expressible in CMSO_1, such as Hamiltonicity and chromatic number, were given by [Espelage et al., 2001; Gerber and Kobler, 2003; Kobler and Rotics, 2003; Suchan and Todinca, 2007; Wanke, 1994]. These two problems were shown to be W[1]-hard parameterized by cliquewidth by Fomin et al. [2010].

As for improved FPT algorithms, Kobler and Rotics [2003] designed algorithms running in time $2^{\mathcal{O}(k)}n$ for problems such as minimum dominating set and c-coloring for fixed c, when given a k-expression. Much later, Lampis [2020] nailed down the dependence on k for c-coloring, giving a $(2^c-2)^k n^{\mathcal{O}(1)}$ time algorithm and showing that no $(2^c-2-\varepsilon)^k n^{\mathcal{O}(1)}$ time algorithm, for any $\varepsilon > 0$ and $c \geq 3$, exists assuming the SETH.

Perhaps more interesting story than dynamic programming on k-expressions is dynamic programming on rank decompositions. Currently, all known FPT algorithms for computing k-expressions work via rankwidth, producing at best a k-expression with $k = 2^{\mathsf{cw}(G)+1}-1$, leading to at least double-exponential algorithms parameterized by cliquewidth even for simple vertex-partitioning problems like maximum independent set or minimum dominating set. Similar double-exponential dependence on cliquewidth results also from naive dynamic programming on rank decompositions.

However, Bui-Xuan et al. [2010] (see also [Ganian and Hliněný, 2010]) showed that by taking advantage of the "rank" in the definition of rankwidth, it is possible to design $2^{\mathcal{O}(k^2)}n^{\mathcal{O}(1)}$ time dynamic programming algorithms for rank decompositions of width k. They designed such algorithms for various vertex-partitioning problems such as maximum independent set, minimum dominating set, and c-coloring for fixed c. Ganian and Hliněný [2010] gave also such an algorithm for feedback vertex set. As $\mathsf{rw}(G) \leq \mathsf{cw}(G)$ and rankwidth can be constant-factor approximated in $2^{\mathcal{O}(\mathsf{rw}(G))}n^{\mathcal{O}(1)}$ time [Oum and Seymour, 2006], this implies $2^{\mathcal{O}(\mathsf{cw}(G)^2)}n^{\mathcal{O}(1)}$ time algorithms parameterized by cliquewidth, without needing a k-expression. Further $2^{\mathcal{O}(k^2)}n^{\mathcal{O}(1)}$ time algorithms on rank decompositions of width k were given by Bui-Xuan et al. [2011] and Bergougnoux and Kanté [2021], and the dependence $2^{\mathcal{O}(k^2)}$ was shown to be optimal for maximum independent set, assuming the ETH, by Bergougnoux et al. [2023]. Oum et al. [2014] introduced a variant of rankwidth called "$\mathbb{Q}$-rankwidth" to give $2^{\mathcal{O}(\mathsf{cw}(G)\log \mathsf{cw}(G))}n^{\mathcal{O}(1)}$ time algorithms for various vertex-partitioning problems parameterized by cliquewidth without needing a k-expression.

Beyond graph problems

So far we have focused on the applications of width parameters in graph problems, but in fact many their most significant applications come from problems that are not about graphs. Let us now discuss these applications.

How to use treewidth if we do not have a graph? The perhaps most common way of associating an instance of a problem with a graph whose tree decompositions are useful for solving the problem is the *primal graph* (also known as the *Gaifman graph*). Suppose we have a problem whose instances can be expressed as pairs (V, C), where V is a set of variables and C is a set of constraints, where each constraint $c \in C$ concerns some subset $V(c) \subseteq V$ of the variables in the sense that we can tell whether an assignment of variables satisfies the constraint based on just the assignment of $V(c)$. Now, the vertices of the primal graph of (V, C) are the variables V, and there is an edge between two vertices $u, v \in V$ if there is a constraint where u and v appear together, that is, exists $c \in C$ with $\{u, v\} \subseteq V(c)$.

The *constraint satisfaction problem* (CSP) is a problem as above, where for each variable $v \in V$ we are given a finite domain $D(v)$ from which its value can be chosen, and each constraint $c \in C$ is given by listing all of the assignments of values to the variables $V(c)$ that satisfy the constraint. The problem is to find an assignment that satisfies all of the constraints. It was shown by Freuder [1990] and Dechter and Pearl [1989] that a CSP with n variables whose each domain has size at most d can be solved in time $n \cdot d^{\mathcal{O}(k)}$ if the input is given together with a tree decomposition of the primal graph of width k, that is, for instances with *primal treewidth* k.

By observing a connection between CSPs and the "conjunctive query evaluation" problem in databases, Kolaitis and Vardi [2000] showed that the evaluation problem can be solved in polynomial time for queries with bounded primal treewidth. A related result purely in the context of the conjunctive query evaluation problem was shown a bit earlier by Chekuri and Rajaraman [2000], who introduced a parameter called "query-width", showed that the evaluation problem can be solved in polynomial time for queries with bounded query-width if a corresponding decomposition is given, and that the query-width of a query is at most its primal treewidth plus one. Other problem observed by [Chandra and Merlin, 1977; Feder and Vardi, 1998] to be closely linked to conjunctive query evaluation and CSPs is the *homomorphism* problem. Let us continue in the setting of CSPs.

The *incidence graph* of a CSP (V, C) has the union $V \cup C$ of variables and constraints as its vertices, and has an edge between a variable $v \in V$ and a constraint $c \in C$ if $v \in V(c)$. Note that the incidence graph is bipartite with a bipartitioning cut (V, C). It is not hard

to observe that the treewidth of the incidence graph, i.e., the *incidence treewidth*, of a CSP is at most its primal treewidth plus one, but the incidence treewidth can be 1 even when the primal treewidth is $|V| - 1$. Chekuri and Rajaraman [2000] showed that the query-width of a CSP is at most the treewidth of its incidence graph plus one, implying that CSPs with bounded incidence treewidth can be solved in polynomial time. Even more generally, it was shown by Gottlob and Pichler [2004] that the query-width of a CSP is at most the cliquewidth of its incidence graph. Generalizations of query-width were given by [Gottlob et al., 2002; Grohe and Marx, 2014; Marx, 2013].

The primal and incidence treewidth were also considered in the slightly different setting of the *Boolean satisfiability problem* (SAT). One can think of SAT as like CSP, but with domains of size 2 and constraints expressed by listing the assignments that do not satisfy the constraint, instead of the assignments that satisfy the constraint. Szeider [2003] observed that the Courcelle's theorem for cliquewidth [Courcelle et al., 2000], or in fact its formulation for directed graphs, implies that SAT is FPT parameterized by incidence treewidth, or more generally, by the cliquewidth of the directed incidence graph. See also the work of Ganian et al. [2013] on using rankwidth for SAT. Earlier, Alekhnovich and Razborov [2011][4] had shown that SAT can be solved in time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$, where n is the number of variables and k is the branchwidth of the hypergraph of the instance. The branchwidth of the hypergraph was shown by Szeider [2003] to be approximately equivalent to the treewidth of the primal graph.

Let us then briefly mention some other applications. In the context of probabilistic inference in Bayesian networks, the running time of the influential "junction tree algorithm" of Lauritzen and Spiegelhalter [1988] is characterized by the treewidth of the "moral graph" of the Bayesian network (see also Dechter [1999]). This setting has in fact motivated several authors to design heuristics for computing treewidth, e.g. [Gogate and Dechter, 2004; Kjærulff, 1992]. In the context of compiler optimization, Thorup [1998] showed that the register allocation problem can be efficiently approximated if the treewidth of the control-flow graph of the program is small, and showed that the control-flow graphs of "structured programs" in fact always have small treewidth. In quantum computing, Markov and Shi [2008] showed that quantum circuits can be simulated in time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$, where k is the treewidth of the underlying graph. Algorithms for solving linear equations and linear programs with running times of form $k^{\mathcal{O}(1)}n \cdot \operatorname{poly} \log n$, where k is the treewidth of a certain graph associated with the input, were given by Fomin et al. [2018] and Dong et al. [2021], respectively. Cunningham and Geelen [2007] used the branchwidth of matroids in the context of integer linear programming.

[4]The conference version of Alekhnovich and Razborov [2011] appeared in 2002 [Alekhnovich and Razborov, 2002].

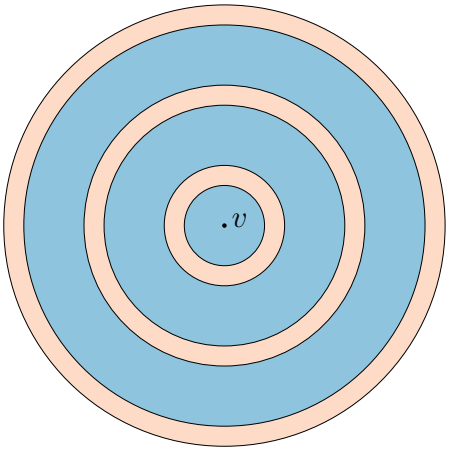

Figure 3.3: An illustration of the Baker's scheme. The area in light red depicts the vertices in a set $X_{q,k}$ for some k and q, and the area in blue the vertices of $G \setminus X_{q,k}$.

Baker's scheme and local treewidth

We then turn from applications that are useful only if the inputs have small width to applications where treewidth is used in a subroutine of an algorithm that works even on graphs of large treewidth. These applications often turn up in algorithms for planar and minor-free graphs, and sometimes can be extended all the way up to general graphs, like we saw in Subsection 3.3.1 with the k-disjoint paths problem.

The maximum independent set and minimum dominating set problems are NP-hard to constant-factor approximate on general graphs [Dinur and Steurer, 2014; Zuckerman, 2007]. However, for planar graphs, Baker [1994] gave approximation algorithms returning a solution within an ε factor of the optimum and running in time $2^{\mathcal{O}(1/\varepsilon)}n$, for any $\varepsilon > 0$. The idea of her algorithm is to show that planar graphs can be sliced into components of small treewidth without affecting the optimum solution much, and then solve each component optimally by dynamic programming on tree decompositions.[5]

In more detail, Baker's scheme is based on the following *bounded local treewidth* property of planar graphs.

Lemma 3.17 (Baker [1994]; Robertson and Seymour [1984])**.** *If G is a planar graph, $v \in V(G)$, d is an integer, and $D \subseteq V(G)$ is the set of vertices that have distance at most d to v, then $\mathsf{tw}(G[D]) \leq \mathcal{O}(d)$.*

Baker's algorithm for approximating independent set proceeds as follows (see also Figure 3.3). Let G be the input planar graph and assume that it is connected. Then, pick

[5]The conference version [Baker, 1983] of [Baker, 1994] pre-dates the wide adoption of treewidth, so it actually uses hand-crafted dynamic programming on "k-outerplanar" graphs. The algorithm has been interpreted as an application of treewidth in later literature, e.g., [Bodlaender, 1988; Eppstein, 2000].

a vertex $v \in V(G)$, let $k = \lceil 1/\varepsilon \rceil$, and for every $q \in [0, k-1]$ define that $X_{q,k} \subseteq V(G)$ is the set of vertices having distance q modulo k from v. It follows from Lemma 3.17 that for each q, the graph $G \setminus X_{q,k}$ has treewidth at most $\mathcal{O}(k)$. This can be seen by observing that each connected component C of $G \setminus X_{q,k}$ consists of vertices that are at distance $[d, d+k-1]$ from v, for some $d \in \mathbb{Z}_{\geq 0}$, and that after contracting all vertices at distance less than d from v into a single vertex, all vertices in C would have distance at most k from the contracted vertex. Now there exists some $q \in [0, k-1]$ so that $X_{q,k}$ intersects an optimum solution in at most a fraction of $1/k \leq \varepsilon$ of it, and therefore solving the problem exactly in $G \setminus X_{q,k}$ for that q results in $(1-\varepsilon)$-approximation. By using treewidth, maximum independent set can be solved in $G \setminus X_{q,k}$ in time $2^{\mathcal{O}(k)}n = 2^{\mathcal{O}(1/\varepsilon)}n$.

Eppstein [2000] (see also [Demaine and Hajiaghayi, 2004]) showed that the bounded local treewidth property of Lemma 3.17 holds in a minor-closed graph class $\mathcal{C}$ if and only if $\mathcal{C}$ excludes some *apex graph*, generalizing the algorithm to those classes. An apex graph is a graph G that contains a vertex $v \in V(G)$ so that $G \setminus \{v\}$ is planar. By showing that minor-free graphs can be decomposed into *apex-minor-free* graphs, i.e., H-minor-free graphs for an apex graph H, Grohe [2003] further generalized the scheme to minor-free graphs. Other generalizations and applications of Baker's scheme have been given by, for example, [Dawar et al., 2006; Dvorák, 2018; Fox-Epstein et al., 2019; Hunt et al., 1998; Klein et al., 1993].

Frick and Grohe [2001] applied the bounded local treewidth property of Lemma 3.17 in another context. They showed that problems expressible in *first-order logic* (FO)[6] can be solved in linear time on apex-minor-free graphs. In particular, their result is analogous to Courcelle's theorem, but with CMSO_2 replaced by FO, and treewidth replaced by the size of an apex graph excluded as a minor. The algorithm of Frick and Grohe is based on the *locality* of first-order logic shown by Gaifman [1982]. Roughly speaking, Gaifman's theorem says that deciding if an FO sentence φ is true in a graph G can be reduced to deciding a *local* FO formula $\psi(v)$ for every vertex $v \in V(G)$, and then combining the results in a certain manner. Here, the fact that $\psi(v)$ is local means that it depends only on the graph induced by vertices at distance at most $d = f(\varphi)$ from v. The central observation of Frick and Grohe is that, by the bounded local treewidth property of apex-minor-free graphs, this graph has bounded treewidth, and thus we can use Courcelle's theorem for deciding $\psi(v)$. A bit more sophistication is then needed for combining the results and making the algorithm run in linear time.

The algorithm of Frick and Grohe was subsequently generalized to larger classes of graphs [Dawar et al., 2007; Dvorák et al., 2013; Flum and Grohe, 2001], culminating in the generalization to "nowhere dense" graphs by Grohe et al. [2017], which is the highest

[6] One can think of FO as like MSO_1, but without set variables.

generality of subgraph-closed graph classes where deciding FO can be FPT. The problem of pinning down the induced subgraph-closed graph classes where FO is FPT is open, but very actively worked on recently [Bonnet et al., 2022; Dreier et al., 2023; Torunczyk, 2023].

Bidimensionality

Another class of applications of treewidth for planar and minor-free graphs stems from the Grid Minor Theorem (Theorem 3.13), or more precisely, from improved versions of it for planar and minor-free graphs given by Robertson et al. [1994] and Demaine and Hajiaghayi [2008]. In particular, Robertson et al. showed that if a planar graph G does not contain the $k \times k$ grid as a minor, then $\mathsf{tw}(G) \leq 6k - 5$. Demaine and Hajiaghayi [2008] generalized this result, showing that there is a function $f\colon \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ so that if G contains neither a graph H nor the $k \times k$ grid as a minor, then $\mathsf{tw}(G) \leq f(|V(H)|) \cdot k$.

These results can be applied to design *subexponential* $2^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$ time FPT algorithms for planar and minor-free graphs. For example, consider the minimum vertex cover problem on planar graphs, parameterized by the solution size k. We can observe that (1) if a graph H is a minor of a graph G, then the minimum vertex cover of H is not larger than that of G, and (2) the size of a minimum vertex cover of the $k \times k$ grid is $\Theta(k^2)$. By combining (1) and (2) with the Grid Minor Theorem for planar graphs, it follows that if we are seeking for a vertex cover of size at most k and the input graph has treewidth more than $\mathcal{O}(\sqrt{k})$, we can immediately return NO. Therefore, the non-trivial instances have treewidth $\mathcal{O}(\sqrt{k})$, so we obtain a $2^{\mathcal{O}(\sqrt{k})}n$ time algorithm.

For some problems, for example dominating set, the property (1) does not hold because edge deletions can increase the size of a minimum dominating set. However, Fomin et al. [2011a] showed that on planar and apex-minor-free graphs, having treewidth $\Omega(k)$ not only implies a $k \times k$ grid minor, but also a $k \times k$ "grid-like" graph as a contraction, which can be interchanged with the grid to make the same argument work also for minimum dominating set and related problems.

The first prototype of the aforementioned idea for subexponential FPT algorithms was perhaps the $2^{\mathcal{O}(\sqrt{k})}n$ time algorithm for dominating set on planar graphs by Alber et al. [2002]. Subsequently, this theory was significantly developed by [Demaine et al., 2005a,b, 2006] and dubbed "bidimensionality", referring to the fact that it is applicable if the parameter has large value on grids, or on the grid-like graphs. In addition to subexponential FPT algorithm, the ideas originating from bidimensionality were later applied also to approximation schemes [Demaine and Hajiaghayi, 2005; Fomin et al., 2011b] and kernelization [Bodlaender et al., 2016b; Fomin et al., 2020] on minor-free graphs.

3.3.4 Complexity

We have surveyed many applications of treewidth for algorithm design, and left many more unmentioned due to space constraints. After spending so much effort in designing algorithms parameterized by treewidth, it could be disappointing if there would be another graph parameter that would have all the good qualities of treewidth, but be even more general, capturing even more graphs. Of course, in some settings, like problems expressible in CMSO_1, this has indeed happened to some extent with the parameters cliquewidth and rankwidth. In this subsection we survey complexity-theoretic arguments giving evidence that treewidth is the right parameter for some problems, i.e., why one sometimes cannot "beat treewidth".

Graph classes with closure properties

The first such argument arises from the Grid Minor Theorem of Robertson and Seymour (Theorem 3.13). Suppose that $\mathcal{C}$ is a minor-closed class of graphs. Then, the Grid Minor Theorem implies that either there exists an upper bound for the treewidth of all graphs in $\mathcal{C}$, or $\mathcal{C}$ contains all planar graphs. Now, if Π is a graph problem that is NP-hard on planar graphs but linear-time solvable on graphs of bounded treewidth, for example, Hamiltonicity or 3-coloring, then we have the following dichotomy: If a minor-closed graph class $\mathcal{C}$ has bounded treewidth, then Π is linear-time solvable on $\mathcal{C}$, and otherwise Π is NP-hard on $\mathcal{C}$.

This easy argument was observed by Makowsky and Mariño [2003]. They also generalized it to the setting of graph classes closed under *topological minors* and problems that are NP-hard on planar graphs with maximum degree 3. Topological minors are like minors, but instead of contraction, only the suppression of degree-2 vertices is allowed. Makowsky and Mariño also observed that the argument does not extend to subgraph-closed graph classes because of the following example. Let $\mathcal{C}$ be a graph class that for every $n \in \mathbb{Z}_{\geq 1}$ contains the graph obtained from the complete graph on n vertices by subdividing every edge 2^{2^n} times, and all of their subgraphs. Now, $\mathcal{C}$ has unbounded treewidth and is subgraph-closed, but problems like Hamiltonicity and 3-coloring are polynomial-time solvable on $\mathcal{C}$ because n-vertex graphs from $\mathcal{C}$ have treewidth $\mathcal{O}(\log \log n)$. In fact, all CMSO_2-expressible problems are linear-time solvable on this class $\mathcal{C}$.[7]

Much more recently, Johnson et al. [2022] showed that one can get around the aforementioned problem for subgraph-closed graph classes if one considers classes that are

[7] This does not hold in general for classes with treewidth $\mathcal{O}(\log \log n)$, but in this case can be shown by a modification of the argument of [Makowsky and Mariño, 2003, Proposition 32].

defined by excluding a finite family of graphs as subgraphs. They showed that for many graph problems, for example, maximum independent set and minimum dominating set, the problem is polynomial-time solvable on a graph class $\mathcal{C}$ defined by the exclusion of a finite family $\mathcal{H}$ of subgraphs if and only if $\mathcal{C}$ has bounded treewidth, assuming $\mathsf{P} \neq \mathsf{NP}$.

Monadic second-order logic

As we recall from Subsection 3.3.2, the theorem of Seese [1991] gives evidence that Courcelle's theorem for MSO_2 cannot be extended beyond treewidth. However, the aforementioned construction of Makowsky and Mariño shows that a direct complexity-theoretic analogue of Seese's theorem is not true, even in subgraph-closed graph classes. Nevertheless, Kreutzer and Tazari [2010b] (see also [Kreutzer and Tazari, 2010a]) showed that after placing down definitions that forbid such frivolous constructions, one can prove a complexity-theoretic analogue of Seese's theorem for subgraph-closed classes.

Kreutzer and Tazari defined that the treewidth of a class $\mathcal{C}$ of graphs is *strongly unbounded* by a function $f\colon \mathbb{Z}_{\geq 1} \to \mathbb{Z}_{\geq 1}$ and *gap-degree* $\gamma \geq 1$ if there exists $\varepsilon < 1$, so that for all $n \in \mathbb{Z}_{\geq 1}$ there is a graph $G_n \in \mathcal{C}$ such that

1. the treewidth of G_n is at least $f(|V(G_n)|)$, and between n and $\mathcal{O}(n^\gamma)$, and
2. given n, G_n can be constructed in time $\mathcal{O}(2^{n^\varepsilon})$.

Then their results is as follows.

Theorem 3.18 (Kreutzer and Tazari [2010b])**.** *If $\mathcal{C}$ is a subgraph-closed graph class, and the treewidth of $\mathcal{C}$ is strongly unbounded by $f(n) = \log^{28\alpha} n$ and gap-degree γ for some $\alpha > \gamma$, then assuming the ETH, MSO_2 is not XP parameterized by the length of the sentence on the class $\mathcal{C}$.*

The idea of the proof of Theorem 3.18 is again to exploit the Grid Minor Theorem, or in fact a "grid-like-minor" theorem of Reed and Wood [2012] since no polynomial bounds for the Grid Minor Theorem were known at the time. It can indeed be seen as a complexity-theoretic analogue of Seese's theorem, since both proofs are based on encoding a Turing machine in a grid minor. Ganian et al. [2014] gave a variant of Theorem 3.18 but with MSO_1 and a different set of technical assumptions. Another variant, slightly stronger than the one by Ganian et al. was given by Amarilli et al. [2016]. They also gave similar results for "probabilistic" MSO. All of these results require the class $\mathcal{C}$ to be subgraph-closed.

Constraint satisfaction

We then turn to CSPs. Recall that an n-variable CSP with domains of size at most d can be solved in time $n \cdot d^{\mathcal{O}(k)}$ if its primal treewidth is k. In particular CSP is XP parameterized by primal treewidth. Grohe et al. [2001] showed that CSP cannot be XP parameterized by any more general parameter of primal graphs. In particular, they showed the following theorem.

Theorem 3.19 (Grohe et al. [2001])**.** *If $\mathcal{C}$ is a graph class, then CSP restricted to instances whose primal graphs are from $\mathcal{C}$ is polynomial-time if and only if $\mathcal{C}$ has bounded treewidth, assuming FPT $\neq$ W[1].*

This result does not require any technical conditions or closure properties of $\mathcal{C}$.[8] The explanation for why we avoid the technicalities of Theorem 3.18 is that here, the domain size d can be arbitrarily large. In particular, in our counterexample class $\mathcal{C}$ constructed by subdividing n-cliques 2^{2^n} times, we do not have polynomial-time algorithm for CSP, because by using domains of size $d = 2^{2^n}$ we could polynomial-time reduce the problem of finding a clique of size n in a graph with 2^{2^n} vertices to a CSP whose primal graph is the n-clique subdivided 2^{2^n} times.

Grohe [2007] generalized Theorem 3.19 to arbitrary classes of relational structures with bounded arity, showing that in this case, "bounded treewidth modulo homomorphic equivalence" characterizes the polynomial-time solvable classes. A version of this result for counting problems was given by Dalmau and Jonsson [2004]. All of these results use arguments based on the Grid Minor Theorem. By finding an argument without the Grid Minor Theorem, Marx [2010b] gave a more quantitative version of Theorem 3.19.

Theorem 3.20 (Marx [2010b])**.** *If $\mathcal{C}$ is a class of graphs with unbounded treewidth, and there exists an algorithm with running time $f(G) \cdot (nd)^{o(\mathsf{tw}(G)/\log \mathsf{tw}(G))}$ for binary CSP with n variables and domain size d on instances with primal graphs $G \in \mathcal{C}$, for some function f, then the ETH fails.*

Here, *binary CSP* means that each constraint c has $|V(c)| = 2$. By Theorem 3.20, the $n \cdot d^{\mathcal{O}(k)}$ time algorithm is (almost) optimal on every graph class $\mathcal{C}$. To obtain lower bounds such as the one in Theorem 3.20, one cannot rely on the Grid Minor Theorem. Instead, Marx developed an alternative technique for embedding problems into graphs of

[8] Although it requires some conditions on how a supposed algorithm should behave on instances whose primal graphs are not from $\mathcal{C}$, or that $\mathcal{C}$ is recursively enumerable. See the discussion in [Marx, 2010b]. Throughout this section, we ignore these issues by not formally defining what it means for an algorithm to work only on a restricted class of inputs.

large treewidth by using the (approximative) duality between balanced separators and concurrent flows introduced by Leighton and Rao [1999].

Finally, before moving from hardness on graph classes to hardness on individual graphs, we briefly mention a couple more results. In the context of Bayesian networks, Chandrasekaran et al. [2008] and Kwisthout et al. [2010] gave results showing that bounded treewidth is the only structural restriction that allows to perform inference in polynomial time, under some technical assumptions. Amarilli and Monet [2022] showed that such a result holds also for the problem of weighted counting of matchings in graphs.

Instance-specific hardness

Instead of showing that problems are hard on infinite graph classes, it would be satisfying to show that every individual instance with high treewidth is hard. However, it is not clear how such hardness would be formulated, as for every individual instance there exists a trivial algorithm that solves the instance simply by hard-coding the answer. Even if we would allow varying the solution, for example, by varying the weights of vertices, there still would exist a constant-time algorithm for solving the problem on this graph, as the input size would be constant.

Nevertheless, instance-specific hardness results can be given if we assume that the algorithm solving the problem works in a certain way, in particular, if we restrict the model of computation. This makes the most sense if there is some popular paradigm for solving problems of certain type, and we restrict our attention to a model of computation that captures this paradigm. This approach has also the additional benefit that we can obtain unconditional lower bounds without resolving major open problems in complexity theory, like P vs NP.

One such model of computation is that of *resolution proofs* for SAT (see e.g. [Buss and Nordström, 2021]). If a SAT instance is unsatisfiable, then there always exists a proof of unsatisfiability of a certain type, called a resolution proof. Many algorithms for SAT, like clause learning solvers and the $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithm parameterized by primal treewidth k, produce resolution proofs of unsatisfiability [Alekhnovich and Razborov, 2011; Beame et al., 2004]. In fact, the primal treewidth-based algorithm produces so-called *regular resolution* proofs.

A classical family of SAT instances used for showing lower bounds for regular resolution is that of *Tseitin formulas*, which associate graphs G with SAT instances $\tau(G)$ encoding certain parity conditions on G [Tseitin, 1968]. Building on the earlier work of Itsykson et al. [2021] and de Colnet and Mengel [2023], Itsykson et al. [2022] showed that every

regular resolution proof for an unsatisfiable Tseitin formula $\tau(G)$ must have length at least $2^{\Omega(\mathsf{tw}(G))}$, for every graph G. This implies that any algorithm that produces regular resolution proofs must run in time at least $2^{\Omega(\mathsf{tw}(G))}$ on Tseitin formulas $\tau(G)$, for all graphs G. As the treewidth of the primal graph of $\tau(G)$ is at most $\mathsf{tw}(G) \cdot \log|\tau(G)|$, where $|\tau(G)|$ is the length of $\tau(G)$, an almost matching upper bound of $|\tau(G)|^{\mathcal{O}(\mathsf{tw}(G))}$ follows from the algorithm of Alekhnovich and Razborov [2011].

Another related setting is that of *knowledge compilation*, which studies how certain restricted Boolean circuits can represent the sets of all solutions of SAT formulas [Darwiche and Marquis, 2002]. While these circuits have applications on their own, from the algorithms perspective it is interesting that many algorithms for the problem of counting the number of solutions of a SAT formula (#SAT) implicitly construct these circuits (see e.g. [Muise et al., 2012]), and thus their running time is lower bounded by the size of a smallest possible such circuit.

In particular, solving #SAT by dynamic programming on tree decompositions corresponds to compilation to so-called "d-DNNF" circuits, yielding a $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithm for constructing them, parameterized by the primal treewidth k. Amarilli et al. [2020] proved a host of lower bounds for compilation of monotone SAT formulas. In particular, they showed that for monotone SAT formulas with primal graphs of bounded degree, every d-DNNF circuit has size at least $2^{\Omega(k)}$, where k is the primal treewidth. This means that treewidth characterizes the sizes of d-DNNF circuits for such formulas.

The author considered in [Korhonen, 2021] a model of computation called "tropical circuits" [Jerrum and Snir, 1982; Jukna, 2023], which, as argued by Jukna, models dynamic programming algorithms. It was shown that for every graph G with treewidth k and maximum degree Δ, any tropical circuit for solving the maximum weight independent set problem has size at least $2^{\Omega(k/\Delta)}$. An upper bound of $2^{\mathcal{O}(k)}n$ is given by dynamic programming on treewidth, so on bounded-degree graphs, treewidth captures the tropical circuit complexity of maximum weight independent set.

Lastly, we mention that treewidth has been tightly related to complexity in restricted models of computation such as AC^0 [Li et al., 2017] and monotone arithmetic circuits [Komarath et al., 2023]. Branchwidth has been related to tensor network complexity by [Austrin et al., 2022].

3.4 Lean tree decompositions

In this section we first review the proof of [Bellenbaum and Diestel, 2002; Thomas, 1990] on the existence of lean tree decompositions, and then discuss how it leads us to the ideas of Chapter 4.

A tree decomposition (T, bag) of a graph G is *lean* if for every two nodes $t_1, t_2 \in V(T)$ and subsets of bags $X_1 \subseteq \mathsf{bag}(t_1)$, $X_2 \subseteq \mathsf{bag}(t_2)$ it holds that either

- $\mathsf{flow}(X_1, X_2) = \min(|X_1|, |X_2|)$, or
- there exists an edge st of T on the unique t_1-t_2-path in T such that $\mathsf{flow}(X_1, X_2) = |\mathsf{bag}(s) \cap \mathsf{bag}(t)|$.

This means that in a lean tree decomposition every non-trivial separator between two subsets of bags is explained by an adhesion of the decomposition. Note that the leannes condition is non-trivial even when $t_1 = t_2$. In particular, as a special case it states that if X_1 and X_2 are subsets of the same bag, then $\mathsf{flow}(X_1, X_2) = \min(|X_1|, |X_2|)$. This means that even the existence of any lean tree decomposition is non-trivial, as the trivial single-bag tree decomposition is not lean unless the graph is a clique.

The original motivation of lean tree decompositions arose from the theory of graph minors. They were used by Robertson and Seymour [1990] to show that Wagner's conjecture (Theorem 3.12) holds for graphs of bounded treewidth. For this result, Robertson and Seymour applied the following theorem of Thomas [1990].

Theorem 3.21 (Thomas [1990])**.** *For every graph G, there exists a lean tree decomposition of width $\mathsf{tw}(G)$.*

The preliminary version of the article [Robertson and Seymour, 1990] contained a weaker version of Theorem 3.21 proven by Robertson and Seymour, but in the final version it was replaced by the stronger and more elegant version of Thomas. Later, Bellenbaum and Diestel [2002] gave an even shorter and more elegant proof of Theorem 3.21, although using many ideas introduced by Thomas. Next we present a proof of Theorem 3.21 that mostly follows the proof of Bellenbaum and Diestel [2002] .

We remind the reader that reading this proof is not a prerequisite for reading the rest of the thesis, and warn that it can be a bit more complicated than, for example, the material in Chapter 4. We also advise the reader that the interesting case of the leannes condition from the viewpoint of this thesis is the single-bag case of $t_1 = t_2$, and the proof becomes a bit easier if one thinks only about this case.

3.4.1 The proof

This subsection is dedicated to the proof of Theorem 3.21. The high-level outline of the proof is as follows. We first take a tree decomposition (T, bag) that is "optimal" with respect to a certain optimality criterion. Then, we show that if some pair X_1, X_2 of subsets of bags would violate the leannes condition, we could use a minimum-size (X_1, X_2)-separator to re-arrange the tree decomposition into an even better tree decomposition according to the optimality criterion. This would contradict the optimality of (T, bag), so (T, bag) must be lean.

Let us then give the proof in detail. For a tree decomposition (T, bag) of an n-vertex graph G, we say that its *width-vector* is the $(n+1)$-element vector $\bar{w} = (a_0, a_1, \ldots, a_n)$, where $a_i \in \mathbb{Z}_{\geq 0}$ is the number of bags of size i. We consider the total order of width-vectors where $\bar{w}_1 = (a_0, \ldots, a_n)$ is smaller than $\bar{w}_2 = (b_0, \ldots, b_n)$ if there exists i so that $a_i < b_i$ and $a_j = b_j$ for all $j > i$, i.e., the lexicographic order starting from the end of the vector.

Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition of G with the smallest width-vector. Clearly, $\mathsf{width}(\mathcal{T}) = \mathsf{tw}(G)$, as otherwise an optimum-width tree decomposition of G would have smaller width-vector. We will prove by contradiction that $\mathcal{T}$ is lean.

Suppose not. Then there exist nodes $t_1, t_2 \in V(T)$ and sets of vertices $X_1 \subseteq \mathsf{bag}(t_1)$, $X_2 \subseteq \mathsf{bag}(t_2)$ so that $\mathsf{flow}(X_1, X_2) < \min(|X_1|, |X_2|)$ and all adhesions of edges on the t_1-t_2-path in T have size more than $\mathsf{flow}(X_1, X_2)$. Furthermore, let us select such quadruple t_1, t_2, X_1, X_2 so that t_1 and t_2 have the minimum possible distance in T.

Then, we select an (X_1, X_2)-separator S of size $|S| = \mathsf{flow}(X_1, X_2)$ as follows. We define a function $d\colon V(G) \to \mathbb{Z}_{\geq 0}$ so that $d(v)$ is the distance between the unique t_1-t_2-path in T and the closest bag containing v. Note that if v occurs on some bag on the t_1-t_2-path then $d(v)$ is 0, and otherwise a node t_v with $v \in \mathsf{bag}(t_v)$ minimizing the distance to the path is unique. Now, we let S be an (X_1, X_2)-separator of size $|S| = \mathsf{flow}(X_1, X_2)$ that minimizes $\sum_{v \in S} d(v)$.

We use S to construct an improved tree decomposition of G. Let (C_1, S, C_2) be a separation of G with $X_1 \subseteq C_1 \cup S$ and $X_2 \subseteq C_2 \cup S$. We will construct a rooted tree decomposition $\mathcal{T}^1$ of $G[C_1 \cup S]$ and a rooted tree decomposition $\mathcal{T}^2$ of $G[C_2 \cup S]$, so that both of them contain S as a subset of their root bags. Then we will put them together by adding an edge between their root bags.

Let $i \in \{1, 2\}$ and $j = 3 - i$. Informally speaking, the tree decomposition $\mathcal{T}^i = (T^i, \mathsf{bag}^i)$ of $G[C_i \cup S]$ is constructed by taking a copy of $\mathcal{T}$ rooted at the node t_j (note t_j instead of t_i), then removing all vertices in C_j, then adding S to the root bag $\mathsf{bag}^i(t_j)$, and finally

fixing the connectedness condition in a minimal way. In particular, note that adding S to the root bag can break the connectedness condition for vertices in S, and a minimal fix to that is to add, for every $v \in S$, the vertex v to every bag on the path from the root to the subtree containing the other occurrences of v.

Formally, $\mathcal{T}^i = (T^i, \mathsf{bag}^i)$ is constructed as follows. Consider $\mathcal{T} = (T, \mathsf{bag})$ to be rooted at t_j, and let T^i to be a copy of the rooted tree T. To define the bags of $\mathcal{T}^i$, we first define for all $t \in V(T)$ that

$$\mathsf{pull}^i(t) = \{v \in S \mid \mathsf{forget}_{\mathcal{T}}(v) \text{ is a strict descendant of } t \text{ in } T\}.$$

Note that when defining pull^i, we consider $\mathcal{T}$ rooted at t_j. Then,

$$\mathsf{bag}^i(t) = (\mathsf{bag}(t) \setminus C_j) \cup \mathsf{pull}^i(t)$$

for every $t \in V(T)$. Note that the purpose of the the vertices $\mathsf{pull}^i(t)$ is to add S to the root bag and fix the connectedness condition.

Let us first observe that $\mathcal{T}^i$ is indeed a tree decomposition of $G[C_i \cup S]$.

Lemma 3.22. *$\mathcal{T}^i$ is a tree decomposition of $G[C_i \cup S]$.*

Proof. The vertex condition follows from the vertex condition of $\mathcal{T}$, because no vertices in $C_i \cup S$ are deleted and all vertices in C_j are deleted. The same argument works for the edge condition. The connectedness condition holds trivially for vertices in C_i, as their occurrences are not altered compared to $\mathcal{T}$. For vertices $v \in S$, we observe that if a bag of $\mathcal{T}^i$ contains v, then the pull^i sets ensure that also the parent bag contains v, which implies the connectedness condition. □

A more surprising fact, which is at the heart of this proof and eventually the algorithms of this thesis, is that the bags of $\mathcal{T}^i$ are no larger than those of $\mathcal{T}$.

Lemma 3.23. *For every $t \in V(T)$ and $i \in \{1, 2\}$, it holds that $|\mathsf{bag}^i(t)| \leq |\mathsf{bag}(t)|$.*

Proof. For clarity, we prove this for $i = 1$. The proof is the same for $i = 2$. Because $\mathsf{bag}^1(t) = (\mathsf{bag}(t) \setminus C_2) \cup \mathsf{pull}^1(t)$, it suffices to prove that $|\mathsf{pull}^1(t)| \leq |\mathsf{bag}(t) \cap C_2|$. Because S is a minimum-size (X_1, X_2)-separator, there exists a collection $\mathcal{P} = P_1, \ldots, P_{|S|}$ of $|S|$ vertex-disjoint S-X_2-paths (Lemma 2.3). Because (C_1, S, C_2) is a separation and $X_2 \subseteq S \cup C_2$, these paths are disjoint from C_1, in fact, all of their vertices except the first ones are in C_2. As $\mathsf{pull}^1(t) \subseteq S$, there is a subcollection $\mathcal{P}' \subseteq \mathcal{P}$ of $|\mathsf{pull}^1(t)|$ vertex-disjoint $\mathsf{pull}^1(t)$-X_2-paths.

Let us again view $\mathcal{T}$ as rooted at t_2. Each path $P \in \mathcal{P}'$ intersects a bag of a strict descendant of t because the forget-nodes of vertices in $\mathsf{pull}^1(t)$ are strict descendants of t. Also, P intersects $X_2 \subseteq \mathsf{bag}(t_2)$. Therefore, as $G[P]$ is connected, P must intersect $\mathsf{bag}(t)$, and because $\mathsf{pull}^1(t)$ is disjoint with $\mathsf{bag}(t)$, P in fact intersects $\mathsf{bag}(t) \cap C_2$. The fact that these paths are vertex-disjoint implies $|\mathsf{bag}(t) \cap C_2| \geq |\mathsf{pull}^1(t)|$. □

Because $S \subseteq \mathsf{bag}^1(t_2)$ and $S \subseteq \mathsf{bag}^2(t_1)$, it is not hard to see that the tree decomposition $\mathcal{T}' = (T', \mathsf{bag}')$ constructed by connecting (T^1, bag^1) and (T^2, bag^2) by an edge between the root node t_2 of T^1 and the root node t_1 of T^2 is indeed a tree decomposition of G. By Lemma 3.23, the width of $\mathcal{T}'$ is no larger than the width of $\mathcal{T}$. However, it does not yet imply that the width-vector of $\mathcal{T}'$ is smaller than that of $\mathcal{T}$.

To prove that the width-vector of $\mathcal{T}'$ is smaller than the width-vector of $\mathcal{T}$, we will use an argument similar to that of Lemma 3.23, but in a stronger form, using the optimizations for choosing t_1, t_2, and S that we asserted in the beginning of the proof. The argument of Lemma 3.23 is in fact a special case of the argument of the following proof, but they illustrate two different viewpoints, disjoint paths and separators.

Lemma 3.24. *For every $t \in V(T)$ and $i \in \{1, 2\}$, the equality $|\mathsf{bag}^i(t)| = |\mathsf{bag}(t)|$ holds only if $\mathsf{bag}(t) \subseteq C_i \cup S$.*

Proof. For clarity, we prove this for $i = 1$. The proof is the same for $i = 2$. Observe that it suffices to prove that either $\mathsf{pull}^1(t)$ is empty or $|\mathsf{pull}^1(t)| < |\mathsf{bag}(t) \cap C_2|$. For the sake of contradiction, suppose that $\mathsf{pull}^1(t)$ is non-empty and $|\mathsf{pull}^1(t)| \geq |\mathsf{bag}(t) \cap C_2|$. Let us also again view $\mathcal{T} = (T, \mathsf{bag})$ as rooted at t_2.

Claim 3.25. *The set*

$$S' = (S \setminus \mathsf{pull}^1(t)) \cup (\mathsf{bag}(t) \cap C_2)$$

is a (X_1, X_2)-separator and a $(\mathsf{bag}(t), X_2)$-separator.

Proof of the claim. We will prove that S' is a $(C_1 \cup \mathsf{pull}^1(t), X_2)$-separator, which implies the claim because $X_1 \subseteq C_1 \cup S' \cup \mathsf{pull}^1(t)$ and $\mathsf{bag}(t) \subseteq C_1 \cup S'$.

Assume not, and let P be a shortest $(C_1 \cup \mathsf{pull}^1(t))$-$(X_2 \setminus S')$-path in $G \setminus S'$. Because $\mathsf{pull}^1(t)$ is a $(C_1, X_2 \setminus S')$-separator in $G \setminus S'$, P contains no vertices in C_1, and in fact all vertices of P except the first are in C_2. Because P is a connected set that intersects a descendant of t and the root t_2, it must intersect $\mathsf{bag}(t)$. However, $\mathsf{bag}(t)$ is disjoint with $\mathsf{pull}^1(t)$ and $\mathsf{bag}(t) \cap C_2 \subseteq S'$, so no such path P can exist in $G \setminus S'$. ◁

Now, if $|\mathsf{pull}^1(t)| > |\mathsf{bag}(t) \cap C_2|$, then $|S'| < |S|$, which contradicts the fact that S is a minimum-size (X_1, X_2)-separator. Otherwise, $|S'| = |S|$ and $\mathsf{pull}^1(t)$ is non-empty. We consider cases depending on whether t is on the t_1-t_2-path in T.

First, suppose that $t \neq t_1$ and t is on the t_1-t_2-path. Our goal is to contradict the choice of t_1, in particular, to show that t should have been chosen instead. Recall that all bags on the t_1-t_2-path, including $\mathsf{bag}(t)$, have size greater than $\mathsf{flow}(X_1, X_2)$ because the quadruple t_1, t_2, X_1, X_2 violated the leannes condition. By Claim 3.25,

$$\mathsf{flow}(\mathsf{bag}(t), X_2) \leq |S'| = \mathsf{flow}(X_1, X_2) < \min(|\mathsf{bag}(t)|, |X_2|).$$

As $\mathsf{flow}(\mathsf{bag}(t), X_2) \leq \mathsf{flow}(X_1, X_2)$, all adhesions on the t-t_2-path have size greater than $\mathsf{flow}(\mathsf{bag}(t), X_2)$, so $t, t_2, \mathsf{bag}(t), X_2$ is also a quadruple that violates the leannes condition. The distance between t and t_2 is smaller than the distance between t_1 and t_2, so this contradicts the choice of t_1 and t_2.

Then, suppose that t is not on the t_1-t_2-path or $t = t_1$. We aim to contradict the choice of S. Recall that all occurrences of vertices in $\mathsf{pull}^1(t)$ are in the bags of strict descendants of t. This implies that $d(v) < d(u)$ for all $v \in \mathsf{bag}(t)$ and $u \in \mathsf{pull}^1(t)$. As $\mathsf{pull}^1(t)$ is non-empty, it follows that $\sum_{v \in S'} d(v) < \sum_{v \in S} d(v)$, which contradicts the choice of S. □

With the help of Lemma 3.24 we can prove that the width-vector of $\mathcal{T}'$ is smaller than that of $\mathcal{T}$. Our goal will be to show that there exists $k > |S|$ so that $\mathcal{T}'$ has less bags of size k than $\mathcal{T}$, and for all $\ell > k$ at most as many bags of size ℓ as $\mathcal{T}$.

Let $i \in \{1, 2\}$ and $j = 3 - i$. If $|\mathsf{bag}^i(t)| = |\mathsf{bag}(t)|$, then by Lemma 3.24, $\mathsf{bag}(t) \subseteq C_i \cup S$, which implies that $\mathsf{bag}^j(t) \subseteq S$. In particular, either both $\mathsf{bag}^1(t)$ and $\mathsf{bag}^2(t)$ are smaller than $\mathsf{bag}(t)$, or one of them has size $|\mathsf{bag}(t)|$ and another has size at most $|S|$. Let k be the largest integer so that there is $t \in V(T)$ with $|\mathsf{bag}(t)| = k$ such that both $\mathsf{bag}^1(t)$ and $\mathsf{bag}^2(t)$ are smaller than $\mathsf{bag}(t)$. If $k > |S|$, then the above arguments imply that this is indeed the desired k to show that the width-vector of $\mathcal{T}'$ is smaller than the width-vector of $\mathcal{T}$, so it remains to prove that indeed $k > |S|$.

By Lemma 3.24 the equality $|\mathsf{bag}^i(t)| = |\mathsf{bag}(t)|$ cannot happen if $\mathsf{bag}(t)$ intersects C_j, so it suffices to show that there exists $t \in V(T)$ so that $|\mathsf{bag}(t)| > |S|$ and $\mathsf{bag}(t)$ intersects both C_1 and C_2. We show that such t exists on the t_1-t_2 path. Because each adhesion $\mathsf{bag}(a) \cap \mathsf{bag}(b)$ of an edge ab on this path has size more than $|S|$, it cannot be that $\mathsf{bag}(a) \subseteq C_i \cup S$ and $\mathsf{bag}(b) \subseteq C_j \cup S$. However, because $|X_i| > |S|$, $X_i \subseteq C_i \cup S$, and $X_i \subseteq \mathsf{bag}(t_i)$, we have that $\mathsf{bag}(t_1)$ intersects C_1 and $\mathsf{bag}(t_2)$ intersects C_2, so there must be a bag on the path that intersects both C_1 and C_2. This finishes the proof of Theorem 3.21.

3.4.2 Discussion

At first glance, the proof of Theorem 3.21 is not algorithmic. It starts with an imaginary optimal tree decomposition, and proceeds by a proof by contradiction. However, we can observe that this proof by contradiction in fact gives a procedure, that given a tree decomposition that is not lean, improves it by improving the width-vector of it. Because the width-vector cannot be improved forever, repeating this improvement must eventually make any tree decomposition lean. With this procedure we obtain a lean tree decomposition, but unlike in the statement of Theorem 3.21, it seems like we have no guarantee on the width of it. However, it turns out that leannes itself is a strong enough property to obtain a bound on the width as follows.

Lemma 3.26. *If* (T, bag) *is a lean tree decomposition of a graph* G*, then the width of* (T, bag) *is at most* $3 \cdot \mathsf{tw}(G) + 2$.

Proof. Denote $\mathsf{tw}(G) = k$, and suppose that (T, bag) has width at least $3k + 3$, meaning that it has a bag $W = \mathsf{bag}(t)$ of size $|W| \geq 3k + 4$. By Lemma 3.4, there exists a separation (A, S, B) of G with $|S| \leq k + 1$ and $0 < |W \cap A|, |W \cap B| \leq \frac{2}{3}|W|$. In particular,

$$|W \cap (A \cup S)| = |W| - |W \cap B| \geq \lceil |W|/3 \rceil \geq k + 2.$$

Similarly, we obtain that $|W \cap (B \cup S)| \geq k + 2$. Now, $W \cap (A \cup S)$ and $W \cap (B \cup S)$ are sets of size at least $k + 2$, and S is a $(W \cap (A \cup S), W \cap (B \cup S))$-separator of size $k + 1$. As both $W \cap (A \cup S)$ and $W \cap (B \cup S)$ are subsets of the same bag $W = \mathsf{bag}(t)$, this contradicts the leannes of (T, bag). □

This means that the procedure is in fact a 3-approximation algorithm for treewidth! Moreover, we only needed the single-bag case $t_1 = t_2$ of leannes to obtain the bound in Lemma 3.26. At this point, we could ask how fast the procedure for improving non-lean tree decompositions can be implemented, and whether the resulting approximation ratio of 3 can be improved. These questions lead to the proof of Theorem 1.1 in Chapter 4.

Lastly, we note that in addition to this thesis, the ideas from the proof of Theorem 3.21 have been algorithmically applied in the context of computing *unbreakable* tree decompositions by Cygan et al. [2021]. They show that for any given parameter k, there exists a tree decomposition whose adhesions have size at most k and the leannes conditions are satisfied when $t_1 = t_2$ and X_1 and X_2 have size at most $|X_1|, |X_2| \leq k + 1$. Moreover, they give an algorithm for computing such a tree decomposition in time $2^{\mathcal{O}(k \log k)} n^{\mathcal{O}(1)}$. These unbreakable tree decompositions have been applied for obtaining FPT algorithms parameterized by solution size for many different problems, for example, minimum bisection, Steiner

cut, fair bisection, and approximation of minimum k-cut [Cygan et al., 2021; Inamdar et al., 2023; Lokshtanov et al., 2020b].

In the purely graph-theoretic setting, analogues of Theorem 3.21 have been given for several other width parameters. Geelen et al. [2002] showed that connectivity functions admit optimal branch decompositions that are "linked", which is a property analogous to a weaker version of leannes that holds only for $t_1 \neq t_2$. An analogue of Theorem 3.21 but for tree-cut width was given by Giannopoulou et al. [2021]. Erde [2018] gave a general framework for results like this, and as an application showed an analogue of Theorem 3.21 for matroid treewidth.

Part II

Contributions

Chapter 4

Fast 2-approximation algorithm for treewidth

In this chapter we prove the following theorem about 2-approximating treewidth.

Theorem 1.1. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k)}n$ either outputs a tree decomposition of G of width at most $2k+1$ or determines that the treewidth of G is larger than k.*

As discussed before, the high-level idea behind the proof of Theorem 1.1 is to introduce an algorithmic version of the tree decomposition improvement procedure in the proof of [Bellenbaum and Diestel, 2002; Thomas, 1990], and then optimize its running time. In this chapter, we follow a self-contained presentation and do not assume any familiarity with the aforementioned proof.

4.1 Overview

We now give a high-level outline of the algorithm of Theorem 1.1 before formally presenting it. The algorithm is based on applying iterative improvement operations to a tree decomposition. By Theorem 3.8, suppose we have a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ of a graph G, having width between $2k+2$ and $4k+3$. Our goal is to either conclude that G has treewidth higher than k, or to improve the width of $\mathcal{T}$ to $2k+1$.

Let r be a node of $\mathcal{T}$ so that $\mathsf{width}(\mathcal{T}) = |\mathsf{bag}(r)| - 1$. We introduce an *improvement operation*, that takes $\mathsf{bag}(r)$, and either concludes that the treewidth of G is more than k, or transforms $\mathcal{T}$ into an improved tree decomposition $\mathcal{T}'$. The resulting tree decomposition

$\mathcal{T}'$ has width at most the width of $\mathcal{T}$, and it has less bags of size $|\mathsf{bag}(r)|$ than $\mathcal{T}$ has. Therefore, applying $|\mathcal{T}| = \mathcal{O}(n)$ improvement operations will be sufficient for decreasing the width of $\mathcal{T}$ by one, and $\mathcal{O}(kn)$ improvement operations for decreasing the width to $2k+1$.

A natural implementation of the improvements would give a running time of $2^{\mathcal{O}(k)}n$ per operation, resulting in a total time of $2^{\mathcal{O}(k)}n^2$. We then introduce a modified, "local", version of the improvement operation, which makes certain optimizations in the construction of $\mathcal{T}'$. With this optimized version, we then introduce a potential function $\Phi(\mathcal{T})$ that is initially bounded by $2^{\mathcal{O}(k)}n$ and for which $\Phi(\mathcal{T}') < \Phi(\mathcal{T})$ holds, and show that with the help of appropriate data structures each improvement operation can be implemented in time $2^{\mathcal{O}(k)} \cdot (\Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$. This yields the running time $2^{\mathcal{O}(k)} \cdot \Phi(\mathcal{T}) = 2^{\mathcal{O}(k)}n$ over any sequence of improvement operations, resulting in Theorem 1.1.

The rest of this chapter is organized as follows. In Section 4.2 we introduce the improvement operation and prove its main graph-theoretic properties. In Section 4.3 we introduce the optimized version of the improvement operation and the potential function Φ, and analyze them. Then, in Section 4.4 we show that the algorithm can be implemented in $2^{\mathcal{O}(k)}n$ time.

4.2 Improving a tree decomposition

In this section we describe the tree decomposition improvement operation and prove its main graph-theoretic properties. Many ideas of this section are inspired by the proofs of [Bellenbaum and Diestel, 2002; Thomas, 1990], but none of our proofs is directly from therein. In particular, our construction of an improved tree decomposition differs from their construction in that we "split" a tree decomposition into three parts, instead of their two parts. This is crucial for obtaining 2-approximation instead of 3-approximation. Our construction would also work for splitting a tree decomposition into more than three parts, but this would not anymore improve the approximation ratio.

4.2.1 Splittable sets of vertices

Let G be a graph. We say that a set of vertices $W \subseteq V(G)$ is *splittable* if $V(G)$ can be partitioned into four, possibly empty, sets (C_1, C_2, C_3, S) so that there are no edges between C_i and C_j for $i \neq j$ and $|(W \cap C_i) \cup S| < |W|$ holds for all $i \in [3]$. We call such 4-tuple a *split* of W.

The next lemma shows that if a tree decomposition of a graph G has width larger than $2k+1$, then either a largest bag of the tree decomposition is splittable or the treewidth of G is larger than k.

Lemma 4.1. *Let G be a graph of treewidth $\leq k$. Any set of vertices $W \subseteq V(G)$ of size $|W| \geq 2k+3$ is splittable.*

Proof. By Lemma 3.2, there exists a W-balanced separator S of size $|S| \leq k+1$. By the definition of W-balanced separator, for every connected component $C \in \mathsf{cc}(G \setminus S)$ it holds that $|C \cap W| \leq |W|/2$, but there may be more than three components. We claim that these components can be merged into a partition $\mathcal{C}$ of $V(G) \setminus S$ with at most three parts so that for each part $C \in \mathcal{C}$ it holds that $|C \cap W| \leq |W|/2$.

This merging can be achieved by a following process. Let initially $\mathcal{C} = \mathsf{cc}(G \setminus S)$. While $\mathcal{C}$ has at least four parts, let C_1 and C_2 be the two parts with the smallest values of $|C_i \cap W|$. We replace C_1 and C_2 by their union $C_1 \cup C_2$. In this case it must hold that $|(C_1 \cup C_2) \cap W| \leq |W|/2$, because there were at least four parts, and C_1 and C_2 were the two parts with the smallest values of $|C_i \cap W|$.

We end up with a partition $\mathcal{C}$ of $V(G) \setminus S$ with at most three parts so that $|C \cap W| \leq |W|/2$ for all $C \in \mathcal{C}$ and there are no edges between C_i and C_j for distinct $C_i, C_j \in \mathcal{C}$. This directly gives a split of W because $|(W \cap C) \cup S| \leq |W|/2 + k + 1 < |W|$. □

In the algorithm, the set W will always be the root bag of the rooted tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ we are improving, i.e., $W = \mathsf{bag}(r)$ for the root r of T. Moreover, it will always be a largest bag, i.e., $|W| = |\mathsf{bag}(r)| = \mathsf{width}(\mathcal{T}) + 1$.

We impose additional restrictions on the splits that we consider. Recall that $\mathsf{depth}_T(x)$ for a node x of T is the distance from x to the root, $\mathsf{forget}_{\mathcal{T}}(v)$ for a vertex $v \in V(G)$ is the unique node of T closest to the root whose bag contains v, and consider the vertex-depth function $\mathsf{depth}_{\mathcal{T}}(v) = \mathsf{depth}_T(\mathsf{forget}_{\mathcal{T}}(v))$. A split (C_1, C_2, C_3, S) of W is a *minimum split* of W if the split minimizes $|S|$ among all splits of W, and among splits minimizing $|S|$, it further minimizes $\mathsf{depth}_{\mathcal{T}}(S) = \sum_{v \in S} \mathsf{depth}_{\mathcal{T}}(v)$. The improvement operation will be performed using a minimum split of W.

4.2.2 The improvement operation

We describe the construction of an improved tree decomposition by using a minimum split of the root bag. This construction is illustrated with an example in Figure 4.1.

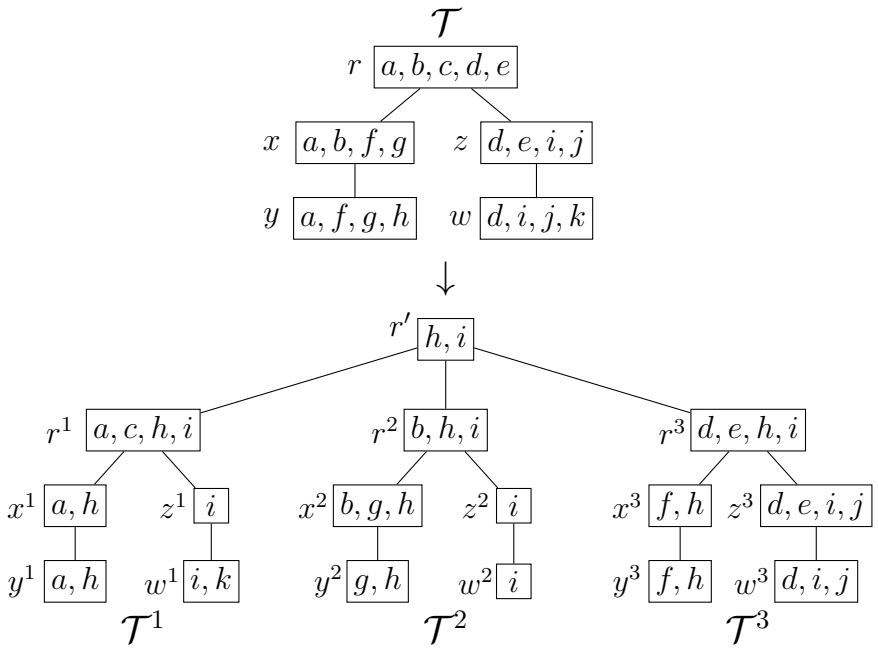

Figure 4.1: Example of the improvement operation. A tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ of a graph G with $V(T) = \{r, x, y, z, w\}$ and $V(G) = \{a, b, c, d, e, f, g, h, i, j, k\}$, with root bag $\mathsf{bag}(r) = \{a, b, c, d, e\}$ (top). For a minimum split $(C_1, C_2, C_3, S) = (\{a, c, k\}, \{b, g\}, \{d, e, f, j\}, \{h, i\})$ of $\mathsf{bag}(r)$, the constructed improved tree decomposition (bottom). It holds that $\mathsf{pull}(r) = \{h, i\}$, $\mathsf{pull}(x) = \{h\}$, and $\mathsf{pull}(t) = \emptyset$ for $t \in \{y, z, w\}$.

Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition of a graph G, rooted at a node r, and (C_1, C_2, C_3, S) a minimum split of $\mathsf{bag}(r)$. We first give a slightly informal description of the improvement operation and then a more formal description with additional notation.

For each $i \in [3]$, we construct a tree decomposition $\mathcal{T}^i = (T^i, \mathsf{bag}^i)$ of the induced subgraph $G[C_i \cup S]$ as follows. We first set T^i to be the copy of T obtained by renaming each node $x \in V(T)$ to x^i. Then, the bags of $\mathcal{T}^i$ are obtained by removing all other vertices than $C_i \cup S$ from each bag of $\mathcal{T}$, and then inserting each vertex $v \in S$ to all bags on the path from the root to the forget-node of v (excluding the forget-node, whose bag already contains v). In particular, each $\mathcal{T}^i$ will have a root bag $\mathsf{bag}^i(r^i) = (\mathsf{bag}(r) \cap C_i) \cup S$. Then, the improved tree decomposition is obtained by combining $\mathcal{T}^1$, $\mathcal{T}^2$, and $\mathcal{T}^3$ by connecting them from their roots r^1,r^2, and r^3 to a new node whose bag is equal to S.

Next we define the construction of the improved tree decomposition more formally with the help of some additional notation. First, for each node x of the tree decomposition $\mathcal{T}$ we define the subset of S inserted to $\mathsf{bag}^i(x^i)$ for all $i \in [3]$ to be

$$\mathsf{pull}(x) = \{v \in S \mid \mathsf{forget}_{\mathcal{T}}(v) \text{ is a strict descendant of } x \text{ in } T\}.$$

Note that $\mathsf{pull}(x) \subseteq S \setminus \mathsf{bag}(x)$, and if y is an ancestor of x, then $\mathsf{pull}(x) \subseteq \mathsf{pull}(y)$.

Then we can define the tree decomposition $\mathcal{T}^i = (T^i, \mathsf{bag}^i)$.

Definition 4.2 (The tree decomposition $\mathcal{T}^i$)**.** *Let (T, bag) be a tree decomposition rooted at a node r and (C_1, C_2, C_3, S) a minimum split of $\mathsf{bag}(r)$. For each $i \in [3]$, the rooted tree decomposition (T^i, bag^i) is obtained by setting T^i to be the copy of T with each node x renamed to x^i, and for all $x \in V(T)$ setting $\mathsf{bag}^i(x^i) = (\mathsf{bag}(x) \cap (C_i \cup S)) \cup \mathsf{pull}(x)$.*

In other words, for each node x of $\mathcal{T}$, each tree decomposition $\mathcal{T}^i$ contains a node x^i, so that $\mathsf{bag}^i(x^i)$ is obtained from $\mathsf{bag}(x)$ by first removing all vertices not in $(C_i \cup S)$ and then inserting the set $\mathsf{pull}(x)$. The insertions of the vertices in $\mathsf{pull}(x)$ can be seen as first adding S to the bag of the root $\mathsf{bag}^i(r^i)$, and then fixing the connectedness condition by "pulling up" vertices $v \in S$ from their forget-nodes to the root. In particular, they ensure that if p^i is the parent of a node x^i in $\mathcal{T}^i$, then $\mathsf{bag}^i(x^i) \cap S \subseteq \mathsf{bag}^i(p^i) \cap S$. This ensures that the connectedness condition holds for vertices in S, and the rest of the conditions of tree decompositions are easy to check to conclude that $\mathcal{T}^i$ is a tree decomposition of the induced subgraph $G[C_i \cup S]$.

The *improved tree decomposition* $\mathcal{T}'$ of $\mathcal{T}$ with respect to (C_1, C_2, C_3, S) is then obtained by taking the disjoint union of $\mathcal{T}^1$, $\mathcal{T}^2$, and $\mathcal{T}^3$ and connecting each of them from their roots to a new root node r' whose bag is $\mathsf{bag}'(r') = S$.

Lemma 4.3. *The improved tree decomposition is a tree decomposition of G.*

Proof. As argued above, for each $i \in [3]$, the tree decomposition $\mathcal{T}^i$ is a tree decomposition of the graph $G[C_i \cup S]$. As S pairwise separates the sets of vertices C_i from each other, it follows that each edge and vertex of G is in one of the induced subgraphs $G[C_i \cup S]$, and therefore as $\mathcal{T}^i$ is a tree decomposition of $G[C_i \cup S]$, the improved tree decomposition satisfies the vertex and edge conditions of tree decompositions. The connectedness condition for vertices not in S follows from the fact that each $\mathcal{T}^i$ satisfies the connectedness condition and that each vertex not in S appears in exactly one $\mathcal{T}^i$. For vertices in S, the connectedness condition is satisfied because it is satisfied for each $\mathcal{T}^i$, $S \subseteq \mathsf{bag}^i(r^i)$, and $S = \mathsf{bag}'(r')$. □

Next we prove the main lemma for arguing that that the improved tree decomposition is indeed improved. The structure of the proof is to assume otherwise and then construct a split (C_1', C_2', C_3', S') that would contradict the fact that (C_1, C_2, C_3, S) is a minimum split. This argument is illustrated in Figure 4.2.

Lemma 4.4. *Let $\mathcal{T} = (T, \mathsf{bag})$ be a rooted tree decomposition, $W = \mathsf{bag}(r)$ the root bag of $\mathcal{T}$, and (C_1, C_2, C_3, S) a minimum split of W. For each node x of $\mathcal{T}$ and any pair of distinct $i, j \in [3]$ it holds that either $|\mathsf{pull}(x)| < |\mathsf{bag}(x) \cap (C_i \cup C_j)|$ or $\mathsf{pull}(x) = \emptyset$.*

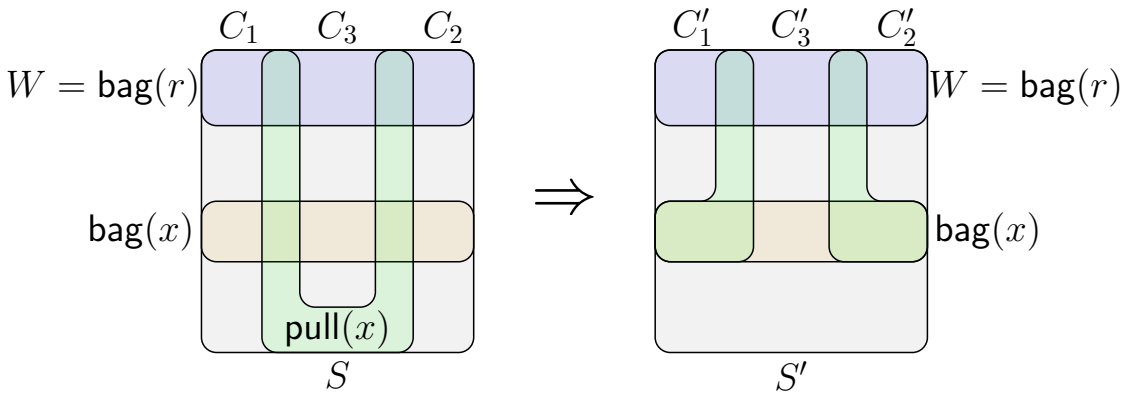

Figure 4.2: Constructing a split (C'_1, C'_2, C'_3, S') of W from a split (C_1, C_2, C_3, S) of W in the proof of Lemma 4.4. The blue illustrates the set W, the orange the set $\mathsf{bag}(x)$, and the green the set S. The set $\mathsf{pull}(x)$ is the part of S that is below $\mathsf{bag}(x)$.

Proof. By symmetry, we assume without loss of generality that $i = 1, j = 2$. Suppose that $|\mathsf{pull}(x)| \geq |\mathsf{bag}(x) \cap (C_1 \cup C_2)|$ and $\mathsf{pull}(x)$ is non-empty. We claim that there is a split (C'_1, C'_2, C'_3, S') of W with $S' = (S \setminus \mathsf{pull}(x)) \cup (\mathsf{bag}(x) \cap (C_1 \cup C_2))$. This split would contradict the minimality of the original split because $|S'| \leq |S|$ and the forget-nodes of vertices in $\mathsf{pull}(x)$ are strict descendants of x and thus strict descendants of the forget-nodes of vertices in $\mathsf{bag}(x) \cap (C_1 \cup C_2)$, implying that $\mathsf{depth}_{\mathcal{T}}(u) < \mathsf{depth}_{\mathcal{T}}(v)$ for all $u \in \mathsf{bag}(x) \cap (C_1 \cup C_2)$ and $v \in \mathsf{pull}(x)$.

To show that there is indeed such a split (C'_1, C'_2, C'_3, S'), first note that $\mathsf{pull}(x)$ does not intersect W because $\mathsf{bag}(x)$ separates $\mathsf{pull}(x)$ from W and $\mathsf{bag}(x) \cap \mathsf{pull}(x) = \emptyset$, so $W \cap S \subseteq W \cap S'$. Next we prove that the sets of vertices $(W \cap C_1) \setminus S'$, $(W \cap C_2) \setminus S'$, and $(W \cap C_3) \setminus S'$ are in different connected components of $G \setminus S'$. This implies that we can partition $V(G) \setminus S'$ to (C'_1, C'_2, C'_3) so that $W \cap C'_i = (W \cap C_i) \setminus S'$ and there are no edges between C'_i and C'_j for $i \neq j$, implying that (C'_1, C'_2, C'_3, S') is a split of W.

Suppose that there is a path from $(W \cap C_k) \setminus S'$ to $(W \cap C_\ell) \setminus S'$, with $k \neq \ell$, in $G \setminus S'$, and by symmetry assume that $k \in [2]$. The path must intersect $\mathsf{pull}(x)$ before intersecting other vertices of $V(G) \setminus C_k$ because $S' \cup \mathsf{pull}(x) \supseteq S$ separates C_k from $V(G) \setminus C_k$. Therefore we have a path from $W \cap C_k$ to $\mathsf{pull}(x)$ that is contained in $(C_k \cup \mathsf{pull}(x)) \setminus S'$. This path must have a vertex in $\mathsf{bag}(x)$ because $\mathsf{bag}(x)$ separates W from $\mathsf{pull}(x)$. However, because $k \in [2]$, $\mathsf{bag}(x) \cap (C_k \cup \mathsf{pull}(x)) = \mathsf{bag}(x) \cap C_k \subseteq S'$, so this path cannot have a vertex in $\mathsf{bag}(x)$. □

Because $\mathsf{bag}^i(x^i) = (\mathsf{bag}(x) \setminus (C_j \cup C_k)) \cup \mathsf{pull}(x)$ where i, j, k is a permutation of $1, 2, 3$, Lemma 4.4 implies that $|\mathsf{bag}^i(x^i)| \leq |\mathsf{bag}(x)|$ for all $i \in [3]$, and that $|\mathsf{bag}^i(x^i)| < |\mathsf{bag}(x)|$ if $\mathsf{pull}(x)$ is non-empty. This shows that the width of the improved tree decomposition is at

most the width of $\mathcal{T}$. Moreover, the only case when $|\mathsf{bag}^i(x^i)| = |\mathsf{bag}(x)|$ can hold is when $\mathsf{bag}(x) \subseteq C_i \cup S$, in which case it holds that $\mathsf{bag}^i(x^i) = \mathsf{bag}(x)$ and $\mathsf{bag}^j(x^j) = \mathsf{bag}(x) \cap S$ for $j \neq i$. Together with the fact that $|\mathsf{bag}^i(r^i)| < |\mathsf{bag}(r)|$ and $|S| < |\mathsf{bag}(r)|$ by the definition of a split, this implies that the number of bags of size $|\mathsf{bag}(r)|$ in the improved tree decomposition is smaller than the number of bags of size $|\mathsf{bag}(r)|$ in $\mathcal{T}$ if $\mathsf{bag}(r)$ is a largest bag of $\mathcal{T}$.

At this point, we have ingredients for a quite simple 2-approximation algorithm for treewidth running in time $2^{\mathcal{O}(k)}n^2$: By rooting the tree decomposition at a largest bag, the improvement operation decreases the number of largest bags by one. As the number of largest bags is initially at most n (recall Lemma 2.13), it suffices to perform at most n iterations of the improvement operation to improve the width by one. By making use of Theorem 3.8, we can assume to start with a 4-approximate tree decomposition, so $\mathcal{O}(nk)$ iterations of the improvement operation are sufficient. Each iteration can be implemented in $2^{\mathcal{O}(k)}n$ time by finding a minimum split by dynamic programming on the tree decomposition we have, thus resulting in a total running time of $2^{\mathcal{O}(k)}n^2$. In the following two sections we improve this to $2^{\mathcal{O}(k)}n$.

4.3 Amortized local improvement

A direct implementation of the improvement operation of the previous section would have running time $\Omega(n)$, which would result in $\Omega(n^2)$ running time over n improvements. In this section we introduce the *pruned improvement operation* that is a slightly changed version of the improvement operation of the previous section. We show that the pruned improvement operation can be implemented so that the number of nodes edited over the course of the algorithm is bounded by $2^{\mathcal{O}(k)}n$, and moreover that in each pruned improvement operation the nodes edited form a subtree containing the root, i.e., a prefix.

The main idea behind the pruned improvement operation is to exploit the fact that, as was discussed in the end of the previous section, $|\mathsf{bag}^i(x^i)| = |\mathsf{bag}(x)|$ can hold only in the case when $\mathsf{bag}(x) \subseteq C_i \cup S$, in which case $\mathsf{bag}^i(x^i) = \mathsf{bag}(x)$. In this case, the whole subtree of $\mathcal{T}$ rooted at x will be handled in constant time by directly copying it to $\mathcal{T}^i$, and not constructing a subtree corresponding to it in $\mathcal{T}^j$ for $j \neq i$. In the other case, when $|\mathsf{bag}^i(x^i)| < |\mathsf{bag}(x)|$ for all $i \in [3]$, the work will be charged from a potential function that is initially bounded by $2^{\mathcal{O}(k)}n$.

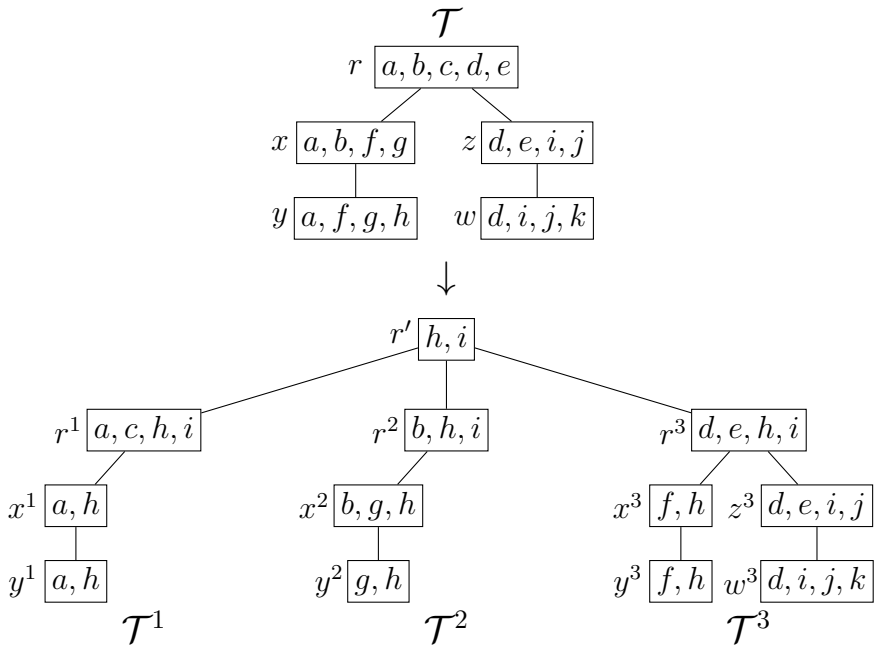

Figure 4.3: Example of the pruned improvement operation. A tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ of a graph G with $V(T) = \{r, x, y, z, w\}$ and $V(G) = \{a, b, c, d, e, f, g, h, i, j, k\}$, with root bag $\mathsf{bag}(r) = \{a, b, c, d, e\}$ (top). For a minimum split $(C_1, C_2, C_3, S) = (\{a, c, k\}, \{b, g\}, \{d, e, f, j\}, \{h, i\})$ of $\mathsf{bag}(r)$, the constructed pruned improved tree decomposition (bottom). The nodes r, x, and y are editable, and the nodes z and w are covered by C_3. Note that even though the vertex k is in C_1, it occurs in pruned $\mathcal{T}^3$ instead of pruned $\mathcal{T}^1$ because the only node whose bag contains k is covered by C_3.

4.3.1 Pruned improvement operation

We define the pruned improvement operation which will be used instead of the improvement operation of Section 4.2. The pruned improvement operation is illustrated with an example in Figure 4.3.

Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition rooted at a node r and (C_1, C_2, C_3, S) a minimum split of $\mathsf{bag}(r)$. We say that a node x of $\mathcal{T}$ is *editable* if $\mathsf{bag}(x)$ intersects at least two of the sets C_1, C_2, C_3 and every ancestor of x is editable. The root r is always editable because the definition of a split implies that $\mathsf{bag}(r)$ must intersect at least two of the sets C_1, C_2, and C_3, and therefore the set of editable nodes forms a prefix of T.

Observe that a node x that is not editable has a unique highest ancestor y (which may be x itself) for which it holds that $\mathsf{bag}(y) \subseteq C_i \cup S$ for some $i \in [3]$. In this case we say that x is *covered* by C_i (or just covered without specifying C_i). When $\mathsf{bag}(y) \subseteq S$, we define that x is covered by C_1, but not by C_2 or C_3, implying that every node that is not editable is covered by exactly one C_i. Observe that by definition, if x is covered by C_i then also all of its descendants are covered by C_i. In particular, T can be partitioned

into a prefix of editable nodes, and multiple rooted subtrees, each of which has a root x with $\mathsf{bag}(x) \subseteq C_i \cup S$ for some $i \in [3]$ and whose all nodes are covered by C_i.

We also make the following observation.

Lemma 4.5. *If a node x is covered, then $\mathsf{pull}(x) = \emptyset$.*

Proof. The node x has an ancestor y for which it holds that $\mathsf{bag}(y) \subseteq C_i \cup S$ for some $i \in [3]$. Now, as $|\mathsf{bag}(y) \cap (C_j \cup C_k)| = 0$, where i, j, k is a permutation of $1, 2, 3$, by Lemma 4.4 it holds that $\mathsf{pull}(y) = \emptyset$. By the definition of $\mathsf{pull}(x)$, we have that $\mathsf{pull}(x) \subseteq \mathsf{pull}(y)$ whenever y is an ancestor of x. □

Next we define the tree decomposition $\mathcal{T}^i$ in the pruned improvement operation.

Definition 4.6 (Pruned $\mathcal{T}^i$)**.** *Let $\mathcal{T}$ be a tree decomposition rooted at a node r and (C_1, C_2, C_3, S) a minimum split of $\mathsf{bag}(r)$. For each $i \in [3]$, the pruned $\mathcal{T}^i = (T^i, \mathsf{bag}^i)$ is obtained by replacing each node x of $\mathcal{T}$ by*

1. *a node x^i with $\mathsf{bag}^i(x^i) = (\mathsf{bag}(x) \cap (C_i \cup S)) \cup \mathsf{pull}(x)$ if x is editable,*
2. *a node x^i with $\mathsf{bag}^i(x^i) = \mathsf{bag}(x)$ if x is covered by C_i, or*
3. *nothing if x is covered by C_j for $j \neq i$.*

For editable nodes, the construction of pruned $\mathcal{T}^i$ is the same as the original construction of $\mathcal{T}^i$. For a node x that is covered by C_i, a copy x^i is created to the decomposition $\mathcal{T}^i$, but no copies x^j to $\mathcal{T}^j$ for $j \neq i$ are created. In particular, one may think of the construction of pruned $\mathcal{T}^i$ as first creating the original construction for the editable nodes, and then for each node x that is covered by C_i and whose parent p is editable, copying the subtree rooted at x from $\mathcal{T}$ to $\mathcal{T}^i$, attaching it as a child of p^i.

Next we show that pruned $\mathcal{T}^i$ can be used in the improvement operation instead of the original $\mathcal{T}^i$.

Lemma 4.7. *Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition rooted at a node r and (C_1, C_2, C_3, S) a minimum split of $\mathsf{bag}(r)$. The tree decomposition $\mathcal{T}'$ constructed by connecting pruned $\mathcal{T}^1$, $\mathcal{T}^2$, $\mathcal{T}^3$ from their roots r^1, r^2, r^3 to a new node r' with $\mathsf{bag}'(r') = S$ is a tree decomposition of G.*

Proof. First, note that for every node x of $\mathcal{T}$, either a node x^i with $\mathsf{bag}'(x^i) = \mathsf{bag}(x)$ appears in the construction, or the nodes x^i with $\mathsf{bag}'(x^i) = (\mathsf{bag}(x) \cap (C_i \cup S)) \cup \mathsf{pull}(x)$

for all $i \in [3]$ appear in the construction. Therefore, as every vertex and edge of G is in some induced subgraph $G[C_i \cup S]$ for $i \in [3]$, the constructed tree decomposition satisfies the vertex and edge conditions.

For the connectedness condition for a vertex $v \in C_i$, there are two cases. First, if v does not appear in a bag of any editable node, then v must be completely contained in the bags of a rooted subtree covered by some C_j, and therefore because this subtree is directly copied to pruned $\mathcal{T}^j$, the connectedness condition is maintained. Second, if v appears in a bag of an editable node, v will appear only in pruned $\mathcal{T}^i$. This is because now, if there is a covered node x with $v \in \mathsf{bag}(x)$, it must be covered by C_i, because otherwise the bag $\mathsf{bag}(y)$ of the highest covered ancestor y of x would not contain v, but y would separate x from the editable nodes, violating the connectedness condition for v in $\mathcal{T}$. Therefore for each node x of the subtree containing v in $\mathcal{T}$, there will be a node x^i with $v \in \mathsf{bag}^i(x^i)$ in pruned $\mathcal{T}^i$, and therefore the connectedness condition is satisfied for v.

Finally, we argue that the connectedness condition holds for each vertex $v \in S$. To this end, we first observe that because the root is editable, it holds that $S \subseteq \mathsf{bag}^i(r^i)$ for every $i \in [3]$. Second, we show that if $v \in \mathsf{bag}^i(x^i)$ for a non-root node x^i of pruned $\mathcal{T}^i$, then it also holds that $v \in \mathsf{bag}^i(p^i)$ for the parent p^i of x^i. If the parent p of x is editable, we have that if $v \in \mathsf{bag}^i(x^i)$, then either $v \in \mathsf{bag}(p)$ or $v \in \mathsf{pull}(p)$ and thus $v \in \mathsf{bag}^i(p^i)$. If both x and its parent p are covered by C_i, we have that if $v \in \mathsf{bag}(x)$, then $v \in \mathsf{bag}(p)$, because $\mathsf{pull}(p) = \emptyset$ by Lemma 4.5, implying that if $v \in \mathsf{bag}^i(x^i)$ then $v \in \mathsf{bag}^i(p^i)$. □

The pruned improvement operation will be implemented by only editing the tree decomposition for the editable nodes, and directly copying the covered rooted subtrees in constant time by just changing pointers. In Section 4.4 we will argue that with the help of appropriate data structures, the pruned improvement operation can be implemented in time $2^{\mathcal{O}(k)}t$, where t is the number of editable nodes. In order to do this, one remaining property to require in the improvement operation is to maintain that T is subcubic, i.e., has maximum degree 3. Next we give the final definition of our improvement operation that maintains this by duplicating each node r^i if necessary.

Definition 4.8 (Pruned improved tree decomposition)**.** *Let $\mathcal{T} = (T, \mathsf{bag})$ be a subcubic tree decomposition rooted at a node r and (C_1, C_2, C_3, S) a minimum split of $\mathsf{bag}(r)$. The pruned improved tree decomposition $\mathcal{T}'$ of $\mathcal{T}$ with respect to (C_1, C_2, C_3, S) is constructed by first constructing pruned $\mathcal{T}^1$, $\mathcal{T}^2$, $\mathcal{T}^3$, then for each $i \in [3]$, if r^i has three children c_1^i, c_2^i, and c_3^i, adding a new node s^i with $\mathsf{bag}^i(s^i) = \mathsf{bag}^i(r^i)$ connected to r^i, c_1^i, and c_2^i, removing the edges between r^i and c_1^i, c_2^i, and then combining $\mathcal{T}^1$, $\mathcal{T}^2$, and $\mathcal{T}^3$ by connecting each r^i to a new node r' with $\mathsf{bag}'(r') = S$.*

The construction of the pruned improved tree decomposition maintains maximum degree 3 because pruned $\mathcal{T}^i$ has the same maximum degree as $\mathcal{T}$, and splitting the node r^i into r^i and s^i ensures that the degree of r^i is at most 2 in $\mathcal{T}^i$, implying it is at most 3 in $\mathcal{T}'$.

4.3.2 Amortization

We show that the total number of editable nodes over the course of a sequence of pruned improvement operations is bounded by $2^{\mathcal{O}(k)}n$. Here we use the property that $\mathsf{bag}(r)$ is a largest bag of $\mathcal{T}$, i.e., the width of $\mathcal{T}$ is assumed to be $|\mathsf{bag}(r)| - 1$. For the amortization, we define the following potential function on a tree decomposition $\mathcal{T}$.

Definition 4.9. *Let $\mathcal{T}$ be a tree decomposition, w an integer, and x a node of $\mathcal{T}$. The w-potential of x in $\mathcal{T}$ is*

$$\Phi_{w,\mathcal{T}}(x) = \begin{cases} |\mathsf{bag}(x)| \cdot 3^{|\mathsf{bag}(x)|}, & \text{if } |\mathsf{bag}(x)| \leq w \text{ and} \\ 3|\mathsf{bag}(x)| \cdot 3^{|\mathsf{bag}(x)|}, & \text{if } |\mathsf{bag}(x)| > w. \end{cases}$$

The w-potential of $\mathcal{T}$ is $\Phi_w(\mathcal{T}) = \sum_{x \in V(T)} \Phi_{w,\mathcal{T}}(x)$.

The w-potential of a tree decomposition $\mathcal{T}$ of width k is bounded by $\mathcal{O}(3^k \cdot k \cdot |\mathcal{T}|)$. Next we show that a pruned improvement operation on a largest bag of size $w+1$ decreases the w-potential by at least the number of editable nodes.

Lemma 4.10. *Let $\mathcal{T}$ be a subcubic tree decomposition of width w rooted at a node r, and assume $|\mathsf{bag}(r)| = w + 1$. Let also (C_1, C_2, C_3, S) be a minimum split of $\mathsf{bag}(r)$. If $\mathcal{T}'$ is the pruned improved tree decomposition of $\mathcal{T}$ with respect to (C_1, C_2, C_3, S) and t is the number of editable nodes, then $\Phi_w(\mathcal{T}') \leq \Phi_w(\mathcal{T}) - t$.*

Proof. The tree decomposition $\mathcal{T}'$ will have four types of nodes: nodes x^i corresponding to covered nodes of $\mathcal{T}$, nodes x^i corresponding to editable non-root nodes of $\mathcal{T}$, nodes r^i and s^i corresponding to the root of $\mathcal{T}$, and the node r' with $\mathsf{bag}'(r') = S$.

Let $\mathcal{E}$ be the set of editable nodes of $\mathcal{T}$, excluding the root r. Let $\mathcal{E}'$ be the set of nodes of $\mathcal{T}'$ corresponding to the nodes $\mathcal{E}$, i.e., $\mathcal{E}' = \{x^i \mid x \in \mathcal{E} \text{ and } i \in [3]\}$. Define $\Phi_w(\mathcal{E}) = \sum_{x \in \mathcal{E}} \Phi_{w,\mathcal{T}}(x)$ and $\Phi_w(\mathcal{E}') = \sum_{x' \in \mathcal{E}'} \Phi_{w,\mathcal{T}'}(x')$. As the contribution of covered nodes is the same for $\Phi_w(\mathcal{T}')$ and $\Phi_w(\mathcal{T})$, we get that

$$\Phi_w(\mathcal{T}') \leq \Phi_w(\mathcal{T}) + \Phi_w(\mathcal{E}') - \Phi_w(\mathcal{E}) - \Phi_{w,\mathcal{T}}(r) + \Phi_{w,\mathcal{T}'}(r') + \sum_{i \in [3]} \left(\Phi_{w,\mathcal{T}'}(r^i) + \Phi_{w,\mathcal{T}'}(s^i)\right).$$

Let us start by bounding $\Phi_w(\mathcal{E}') - \Phi_w(\mathcal{E})$. By applying Lemma 4.4 and the fact that each editable node x intersects C_i for at least two different $i \in [3]$, we get that for every $x \in \mathcal{E}$ it holds that

$$|\mathsf{bag}'(x^i)| = |\mathsf{bag}(x)| - |\mathsf{bag}(x) \cap (C_j \cup C_k)| + |\mathsf{pull}(x)| < |\mathsf{bag}(x)|,$$

where i, j, k is a permutation of $1, 2, 3$. Therefore, by $|\mathsf{bag}(x)| \leq w + 1$ we get that

$$\sum_{i \in [3]} \Phi_{w,\mathcal{T}'}(x^i) \leq 3(|\mathsf{bag}(x)| - 1) \cdot 3^{|\mathsf{bag}(x)|-1} \leq (|\mathsf{bag}(x)| - 1) \cdot 3^{|\mathsf{bag}(x)|} \leq \Phi_{w,\mathcal{T}}(x) - 1,$$

which implies $\Phi_w(\mathcal{E}') \leq \Phi_w(\mathcal{E}) - |\mathcal{E}|$, implying that

$$\Phi_w(\mathcal{T}') \leq \Phi_w(\mathcal{T}) - |\mathcal{E}| - \Phi_{w,\mathcal{T}}(r) + \Phi_{w,\mathcal{T}'}(r') + \sum_{i \in [3]} \left(\Phi_{w,\mathcal{T}'}(r^i) + \Phi_{w,\mathcal{T}'}(s^i)\right).$$

For bounding the potential of the nodes r^i, s^i, and r', first we observe that the definition of a split implies

$$|\mathsf{bag}'(r')| \leq |\mathsf{bag}'(r^i)| = |\mathsf{bag}'(s^i)| < |\mathsf{bag}(r)| = w + 1.$$

As $|\mathsf{bag}(r)| = w + 1$, it holds that $\Phi_{w,\mathcal{T}}(r) \geq 9 \cdot \Phi_{w,\mathcal{T}'}(r^i)$, and therefore

$$\Phi_{w,\mathcal{T}}(r) \geq 1 + \Phi_{w,\mathcal{T}'}(r') + \sum_{i \in [3]} \left(\Phi_{w,\mathcal{T}'}(r^i) + \Phi_{w,\mathcal{T}'}(s^i)\right),$$

which implies

$$\Phi_w(\mathcal{T}') \leq \Phi_w(\mathcal{T}) - |\mathcal{E}| - 1,$$

which implies the conclusion, as the number of editable nodes is $|\mathcal{E}| + 1$. □

By Lemma 4.10, the total number of editable nodes over all operations when improving a tree decomposition $\mathcal{T}$ of width w using pruned improvement operations on largest bags is bounded by $\Phi_w(\mathcal{T}) = 2^{\mathcal{O}(w)} \cdot |\mathcal{T}|$, which by Lemma 2.13 can be assumed to be $2^{\mathcal{O}(w)}n$.

4.4 Implementation in linear time

In this section we show that our algorithm can be implemented in $2^{\mathcal{O}(k)}n$ time. We give a data structure that allows implementing the pruned improvement operation of Section 4.3 in $2^{\mathcal{O}(k)}t$ time, where t is the number of editable nodes, and in particular allows walking

over the tree decomposition to perform the operation to all largest bags in a total of $2^{\mathcal{O}(k)}n$ time.

4.4.1 Overview

We treat our algorithm in the form that the input consists of a graph G, an integer k, and a subcubic tree decomposition $\mathcal{T}$ of G of width w, where $2k+2 \leq w \leq 4k+3$. The algorithm either outputs a tree decomposition of width at most $w-1$, or concludes that the treewidth of G is larger than k. It is easy to see that $\mathcal{O}(k)$ applications of this algorithm gives the algorithm $\mathcal{A}$ of Theorem 3.8 and therefore also the algorithm of Theorem 1.1 up to a factor of $k^{\mathcal{O}(1)}$ in the running time.

We note that given a tree decomposition $\mathcal{T}$ of width w, by using Lemmas 2.13 and 2.14 we can obtain a subcubic tree decomposition of width w and $\mathcal{O}(n)$ nodes in $w^{\mathcal{O}(1)}|\mathcal{T}|$ time, so we will assume that the input tree decomposition $\mathcal{T}$ has this form.

During the algorithm we maintain a subcubic tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ and a *root pointer* to a node r of $\mathcal{T}$. We treat $\mathcal{T}$ as rooted at r. We implement a data structure that supports the following queries:

1. $\mathsf{Init}(\mathcal{T}, r)$: Initializes the data structure with a subcubic tree decomposition $\mathcal{T}$ of width w and a root node $r \in V(T)$ in time $2^{\mathcal{O}(w)}|\mathcal{T}|$.
2. $\mathsf{Move}(s)$: Moves the root pointer from r to an adjacent node s in time $2^{\mathcal{O}(w)}$.
3. $\mathsf{Split}()$: Returns $\bot$ if $\mathsf{bag}(r)$ is not splittable, otherwise sets the internal state of the data structure to represent a minimum split (C_1, C_2, C_3, S) of $\mathsf{bag}(r)$ and returns $\top$. Runs in time $2^{\mathcal{O}(w)}$.
4. $\mathsf{State}()$: Assuming there has been a successful Split query after the previous Init or Edit query, returns the intersection $(C_1 \cap \mathsf{bag}(r), C_2 \cap \mathsf{bag}(r), C_3 \cap \mathsf{bag}(r), S \cap \mathsf{bag}(r))$ of $\mathsf{bag}(r)$ and the minimum split (C_1, C_2, C_3, S) represented by the internal state. Runs in time $w^{\mathcal{O}(1)}$.
5. $\mathsf{Edit}(T^*, T', \mathsf{bag}', \pi, r')$: Given a subtree T^* of T with $r \in V(T^*)$, replaces T^* by a given new subtree T' and a given bag function $\mathsf{bag}'\colon V(T') \to 2^{V(G)}$. Here, π is a function from the nodes of $T \setminus T^*$ whose parents are in T^* to the nodes of T', specifying how $T \setminus T^*$ will be connected to T'. The root pointer r will be set to the given node $r' \in V(T')$. Assumes that the constructed tree decomposition is subcubic and has width at most w, and runs in time $2^{\mathcal{O}(w)}(|V(T^*)| + |V(T')|)$.

We give a detailed description of the data structure in the next subsection. Then, in Subsection 4.4.3 we give our algorithm, using the data structure. In Subsection 4.4.4 we give a more fine-grained bound for the $2^{\mathcal{O}(k)}$ factor in the running time of the algorithm.

4.4.2 The data structure

We now describe the details of the data structure. The data structure is essentially a dynamic programming table on the tree decomposition $\mathcal{T}$, directed towards the root r. The main idea of the $\mathsf{Move}(s)$ query is that moving the root r to an adjacent node s changes the dynamic programming tables of only the nodes r and s, and therefore only their tables should be recomputed. For the Split query an essential idea is that while the properties of a split depend on the intersection of $\mathsf{bag}(r)$ with the split (C_1, C_2, C_3, S), the set $\mathsf{bag}(r)$ does not need to be "globally specified" to the dynamic programming because the set $\mathsf{bag}(r)$ will also correspond to the root node of the dynamic programming. The state queries are implemented by tracing the solution backwards in the dynamic programming, and the edit query by removing the old subtree and computing the dynamic programming tables for the new subtree in a bottom-up manner.

The dynamic programming used will be a quite standard application of dynamic programming on tree decompositions for vertex partitioning problems. The perhaps most non-standard part is the secondary optimization of $\mathsf{depth}_{\mathcal{T}}(S)$ required by the definition of a minimum split. All of the $2^{\mathcal{O}(w)}$ factors in the running times of the data structure operations are of form $4^w w^{\mathcal{O}(1)}$, in particular the exponential factor 4^w arising from the number of ways a bag of size at most $w+1$ can intersect a split (C_1, C_2, C_3, S).

Stored information

Let x be a node of $\mathcal{T}$ with a bag $B = \mathsf{bag}(x)$, $\mathcal{T}_x$ the tree decomposition obtained by restricting $\mathcal{T}$ to the subtree rooted at x, and $G[\mathcal{T}_x]$ the subgraph of G induced by vertices in the bags of $\mathcal{T}_x$. For each partition $(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)$ of B and integer $0 \leq h \leq w$ we have a table entry $\mathcal{D}[x][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)][h]$. This table entry stores $\perp$ if there is no partition (C_1, C_2, C_3, S) of $V(G[\mathcal{T}_x])$ such that $|S| = h$ and there are no edges between C_1, C_2, C_3. If there is such a partition, then the minimum possible integer $\mathsf{depth}_{\mathcal{T}_x}(S)$ over all such partitions is stored, defined as $\mathsf{depth}_{\mathcal{T}_x}(S) = \sum_{v \in S} \mathsf{depth}_{T_x}(\mathsf{forget}_{\mathcal{T}_x}(v))$. In particular, if $x = r$, then $\mathsf{depth}_{\mathcal{T}_x}(S)$ is the function that should be minimized on a minimum split as a secondary measure after minimizing $|S|$.

Additionally, for each node x there may be an "internal state" stored in order to trace the dynamic programming backwards to implement the State queries after a Split query. The internal state is a pair $((C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B), h)$, specifying the table entry of this node to which the minimum split fixed by the previous Split query corresponds.

We note that if x is a leaf node then $|V(G[\mathcal{T}_x])| \leq w + 1$, and therefore all entries $\mathcal{D}[x][\ldots][\ldots]$ can be computed directly in $2^{\mathcal{O}(w)}$ time.

Transitions

Let x be a node with at most three children c_1, c_2, c_3, with x having a bag $B = \mathsf{bag}(x)$ and the children having bags $B_1 = \mathsf{bag}(c_1)$, $B_2 = \mathsf{bag}(c_2)$, and $B_3 = \mathsf{bag}(c_3)$. We next describe how to compute in $2^{\mathcal{O}(w)}$ time the table entries $\mathcal{D}[x][\ldots][\ldots]$ given the table entries $\mathcal{D}[c_1][\ldots][\ldots]$, $\mathcal{D}[c_2][\ldots][\ldots]$, and $\mathcal{D}[c_3][\ldots][\ldots]$.

First, we edit the stored depths $\mathsf{depth}_{\mathcal{T}_{c_i}}(S)$ in the entries $\mathcal{D}[\{c_1, c_2, c_3\}][\ldots][\ldots]$ to correspond to depths in $\mathcal{T}_x$ instead of $\mathcal{T}_{c_i}$. In particular, we increment the stored depth $\mathsf{depth}_{\mathcal{T}_{c_i}}(S)$ in each entry $\mathcal{D}[c_i][(C_1 \cap B_i, C_2 \cap B_i, C_3 \cap B_i, S \cap B_i)][h] \neq \bot$ by $h - |S \cap B \cap B_i|$. Then we do the transition by first decomposing it into $\mathcal{O}(w)$ "nice" transitions of types "introduce", "forget", and "join". In particular, we simulate the construction of a nice tree decomposition from Lemma 2.15 inside our tree decomposition.

In an introduce transition we have a node x with a bag B with a single child c with a bag $B' \subseteq B$ and $|B \setminus B'| = 1$. In a forget transition we have a node x with a bag B with a single child c with a bag $B' \supseteq B$ and $|B' \setminus B| = 1$. In a join transition we have a node x with a bag B with two children c_1,c_2, with bags B_1,B_2 with $B = B_1 = B_2$. The decomposition into nice transitions is done by first forgetting every vertex not in $\mathsf{bag}(x)$, then introducing every vertex in $\mathsf{bag}(x)$, and then joining, i.e., similarly to the construction of nice tree decompositions in Lemma 2.15.

The transitions follow standard ideas of dynamic programming on tree decompositions and can be performed in time $2^{\mathcal{O}(w)}$ as follows. We define $\mathcal{D}[\ldots][\ldots][h] = \bot$ for all $h < 0$ and for all $h > w$.

Introduce. Let $\{v\} = B \setminus B'$. For each partition $(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)$ of B and each integer $0 \leq h \leq w$ we set

$$\mathcal{D}[x][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)][h] =$$
$$\mathcal{D}[c][(C_1 \cap B \setminus \{v\}, C_2 \cap B \setminus \{v\}, C_3 \cap B \setminus \{v\}, S \cap B \setminus \{v\})][h - |\{v\} \cap S|],$$

if there are no edges between $C_1 \cap B$, $C_2 \cap B$, $C_3 \cap B$, and otherwise to $\perp$.

Forget. Let $\{v\} = B' \setminus B$. For each partition $(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)$ of B and each integer $0 \leq h \leq w$ we set

$$\begin{aligned} & \mathcal{D}[x][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)][h] = \\ \min\{\ & \mathcal{D}[c][(C_1 \cap B \cup \{v\}, C_2 \cap B, C_3 \cap B, S \cap B)][h], \\ & \mathcal{D}[c][(C_1 \cap B, C_2 \cap B \cup \{v\}, C_3 \cap B, S \cap B)][h], \\ & \mathcal{D}[c][(C_1 \cap B, C_2 \cap B, C_3 \cap B \cup \{v\}, S \cap B)][h], \\ & \mathcal{D}[c][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B \cup \{v\})][h] \quad \}, \end{aligned}$$

where $\min(\perp, n) = n$ for any integer n.

Join. Let c_1,c_2 be the children of x. For each integer $0 \leq h \leq w$ and each partition $(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)$ of B we set

$$\begin{aligned} & \mathcal{D}[x][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)][h] = \\ \min_{h_1+h_2=h+|S\cap B|} \Big(\ & \mathcal{D}[c_1][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)][h_1] + \\ & \mathcal{D}[c_2][(C_1 \cap B, C_2 \cap B, C_3 \cap B, S \cap B)][h_2] \quad \Big), \end{aligned}$$

where $\perp + n = \perp$ and $\min(\perp, n) = n$ for any integer n. Note that we do not double count $\mathsf{depth}_{\mathcal{T}_x}(v)$ for any $v \in S$ because if v is in both subtrees of c_1 and c_2, then it is also in B and therefore has $\mathsf{depth}_{\mathcal{T}_x}(v) = 0$.

Split query

Now the Split query amounts to iterating over all integers $0 \leq h \leq w$ and intersections $(C_1 \cap \mathsf{bag}(r), C_2 \cap \mathsf{bag}(r), C_3 \cap \mathsf{bag}(r), S \cap \mathsf{bag}(r))$ such that $|(\mathsf{bag}(r) \cap C_i)| + h < |\mathsf{bag}(r)|$ for all $i \in [3]$, and returning $\perp$ if all entries of $\mathcal{D}[r][\ldots][\ldots]$ corresponding to them contain $\perp$ and otherwise returning $\top$. In the latter case, the internal state of the root node r will be set to a pair $((C_1 \cap \mathsf{bag}(r), C_2 \cap \mathsf{bag}(r), C_3 \cap \mathsf{bag}(r), S \cap \mathsf{bag}(r)), h)$ such that $\mathcal{D}[r][(C_1 \cap \mathsf{bag}(r), C_2 \cap \mathsf{bag}(r), C_3 \cap \mathsf{bag}(r), S \cap \mathsf{bag}(r))][h]$ is not $\perp$, primarily minimizes h, and secondarily minimizes the stored integer $\mathsf{depth}_{\mathcal{T}}(S)$. In particular, so that (C_1, C_2, C_3, S) is a minimum split and $|S| = h$.

Also, the internal states of all other nodes are invalidated, for example, by incrementing a global counter.

Move query

Consider a move from a node r to an adjacent node s. First, if there has been a successful Split query after the previous Init or Edit query, but the children of r do not have valid internal states, we use the internal state of r to compute the corresponding internal states of its children by implementing the dynamic programming transitions backwards. In particular, we can in time $2^{\mathcal{O}(w)}$ find the dynamic programming states of the children of r that correspond to the split fixed by the previous successful Split query, and set the internal states of the children to correspond to these dynamic programming states. Now the node s is guaranteed to have a valid internal state before we move to it, and by induction the current node r is always guaranteed to have a valid internal state.

Then, when moving the root from the node r to the node s, the only edge whose direction towards the root changes is the edge between r and s. Therefore for all nodes x except r and s the tree decomposition $\mathcal{T}_x$ rooted at x will stay exactly the same. Thus, we first re-compute the dynamic programming table of r with a single transition and then the dynamic programming table of s with a single transition, taking in total $2^{\mathcal{O}(w)}$ time. Note that all of the re-computations of the dynamic programming tables happen after computing the internal states, so the internal state of each node corresponds to the dynamic programming table directed towards the node at which the previous successful Split query was applied.

Init query

The dynamic programming tables are initialized with the already described transitions in a bottom-up manner, starting from the leaves towards the root r. As each transition is implemented in $2^{\mathcal{O}(w)}$ time, the initialization takes $2^{\mathcal{O}(w)}|\mathcal{T}|$ time.

State query

With the move queries we have already guaranteed that the current node r has a valid internal state corresponding to a minimum split (C_1, C_2, C_3, S), if indeed there has been a successful Split query after the previous Init or Edit query. Therefore we just return the partition $(C_1 \cap \mathsf{bag}(r), C_2 \cap \mathsf{bag}(r), C_3 \cap \mathsf{bag}(r), S \cap \mathsf{bag}(r))$ of the internal state.

Edit query

Consider an edit query that replaces a subtree T^* by T', where $r \in V(T^*)$. Because $r \in V(T^*)$, all the dynamic programming tables of nodes of $T \setminus T^*$ are already oriented towards the subtree T^*, and therefore the tables for the new subtree T' can be constructed in a bottom-up manner with $|V(T')|$ transitions. Then, with at most $|V(T')|$ Move operations the root r can be moved to the specified root r'. Therefore, the total running time is $2^{\mathcal{O}(w)}|V(T')| + w^{\mathcal{O}(1)}|V(T^*)|$.

4.4.3 The algorithm

We now describe our algorithm, making use of the data structure. The goal is to traverse the given tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ of width w with the $\mathsf{Move}(s)$ operations, and every time when a bag $\mathsf{bag}(r)$ of size $|\mathsf{bag}(r)| = w+1$ is encountered, to apply the pruned improvement operation.

We start by showing that when the root pointer r of the data structure is on a splittable bag $\mathsf{bag}(r)$ of size $|\mathsf{bag}(r)| = w+1$, the pruned improvement operation can be implemented in time $2^{\mathcal{O}(w)}t$, where t is the number of editable nodes.

Lemma 4.11. *Let the state of the data structure be so that $|\mathsf{bag}(r)| = w + 1$. There is an algorithm that either in time $2^{\mathcal{O}(w)}$ reports that $\mathsf{bag}(r)$ is not splittable, or in time $2^{\mathcal{O}(w)}t$ transforms $\mathcal{T}$ into the pruned improved tree decomposition of $\mathcal{T}$ with respect to a minimum split (C_1, C_2, C_3, S) of $\mathsf{bag}(r)$, where t is the number of editable nodes. In the latter case, the root pointer r of the data structure will be placed to some new node introduced by the pruned improvement operation.*

Proof. First, we use the Split query. If it returns $\perp$ we return that $\mathsf{bag}(r)$ is not splittable. Otherwise, it returns that $\mathsf{bag}(r)$ is splittable and sets the internal state of the data structure to represent a minimum split (C_1, C_2, C_3, S) of $\mathsf{bag}(r)$.

Then, as the editable nodes form a prefix $T_E \subseteq V(T)$ of T containing r, we use the Move and State queries to find the prefix T_E, the nodes $N_T(T_E)$ neighboring T_E, and for all nodes $x \in N_T[T_E]$ the partitions $(C_1 \cap \mathsf{bag}(x), C_2 \cap \mathsf{bag}(x), C_3 \cap \mathsf{bag}(x), S \cap \mathsf{bag}(x))$. Because $\mathcal{T}$ is subcubic, this can be done with $\mathcal{O}(|T_E|)$ Move and State queries by using them to implement a depth-first search that returns from a subtree as soon as it finds a node x with $\mathsf{bag}(x) \subseteq C_i \cup S$.

Then, we construct the pruned improved tree decomposition for editable nodes. For this, we need to determine the sets $\mathsf{pull}(x)$ for all editable nodes x. We observe that by the

definition of $\mathsf{pull}(x)$, it holds that if x has children c_1, c_2, and c_3, then

$$\mathsf{pull}(x) = \mathsf{pull}(c_1) \cup \mathsf{pull}(c_2) \cup \mathsf{pull}(c_3) \cup (S \cap (\mathsf{bag}(c_1) \cup \mathsf{bag}(c_2) \cup \mathsf{bag}(c_3))) \setminus \mathsf{bag}(x).$$

Therefore, by using Lemma 4.5 that $\mathsf{pull}(x) = \emptyset$ for covered nodes x, the sets $\mathsf{pull}(x)$ can be computed in a bottom-up manner in $|T_E| \cdot w^{\mathcal{O}(1)}$ time. After computing the sets $\mathsf{pull}(x)$, computing the pruned improved tree decomposition $\mathcal{T}'_E = (T'_E, \mathsf{bag}'_E)$ for the editable nodes T_E can be done directly by definition (Definitions 4.6 and 4.8) in $|T_E| \cdot w^{\mathcal{O}(1)}$ time.

Then, we use the Edit operation to replace the prefix of editable nodes by the constructed $\mathcal{T}'_E$. Here, the mapping π from $N_T(T_E)$ to $V(T'_E)$ is determined as follows. For a node $x \in N_T(T_E)$, let $i \in [3]$ so that $\mathsf{bag}(x)$ is covered by C_i, and let p be the parent of x in T. Now, $p \in T_E$, and in $V(T'_E)$ there are three copies p^1, p^2, and p^3 corresponding to p. The mapping π is set so that $\pi(x) = p^i$. The node r' in the Edit operation is set to be an arbitrary node in T'_E. This implements the construction of the pruned improved tree decomposition.

In total, we used one Split operation, $\mathcal{O}(|T_E|)$ Move and State operations, and one Edit operation with subtrees of size $\mathcal{O}(|T_E|)$, and therefore the total running time is $2^{\mathcal{O}(w)}|T_E|$ which is $2^{\mathcal{O}(w)}t$. □

Now, by Lemma 4.10, the total time used in the improvement operations implemented as described in the proof of Lemma 4.11 is bounded by $\Phi_w(\mathcal{T}) \cdot 2^{\mathcal{O}(w)} = 2^{\mathcal{O}(w)}n$. What is left is to show is that between the improvement operations, we can use the Move operations to move the pointer r to the next node with a bag of size $w+1$ so that the total number of Move operations used is bounded by $\Phi_w(\mathcal{T})$. We do this with a depth-first-search type algorithm as we next describe.

We traverse the tree decomposition in a depth-first order with Move operations. For simplicity, we add an extra starting node z with an empty bag and degree 1 to the tree decomposition and set the root pointer r to z initially. Note that a node with an empty bag cannot be editable, so the node z will never be edited by the pruned improvement operation. For all nodes there are three states: *unseen*, *open*, and *closed*. At the start the node z is open and other nodes are unseen. Our algorithm traverses the tree decomposition according to the following cases:

1. The node r is open and has an unseen neighbor node s: Apply $\mathsf{Move}(s)$ and set the node s as open.

2. The node r is open and has no unseen neighbors:

2a. It holds that $r = z$: We are done, return $\mathcal{T}$.

2b. It holds that $|\mathsf{bag}(r)| \leq w$: Set r as closed and apply $\mathsf{Move}(s)$ where s is an open neighbor of r.

2c. It holds that $|\mathsf{bag}(r)| = w + 1$: Apply Lemma 4.11. If it returns that $\mathsf{bag}(r)$ is not splittable, then return that the treewidth of G is larger than k. Otherwise, the new nodes inserted by the pruned improvement operation are set as unseen, and then the root r is moved to a node that is open and adjacent to a newly inserted node.

Next we prove two key invariants for arguing the correctness and running time of the above described procedure, in particular that despite the improvement operations, the main properties of the procedure are similar to a standard depth-first-search. First, we show that the open nodes form a path in the tree decomposition.

Lemma 4.12. *The above described procedure maintains the invariant that the open nodes form a path $x_1, \ldots, x_t$ in T, where $x_1 = z$ and $x_t = r$.*

Proof. The case 1 maintains the invariant by appending one vertex to the end of the path and the case 2b by removing the last vertex of the path. For the case 2c, we first observe that the editable subtree contains the node r but not z because $\mathsf{bag}(z)$ is empty. Therefore, because T is a tree and $x_1, \ldots, x_t$ is a path from z to r, removing the editable subtree removes a suffix $x_i, \ldots, x_t$ of the path, where $i \geq 2$. Then, the only node on the path $x_1, \ldots, x_{i-1}$ adjacent to the newly inserted nodes is x_{i-1}, which is then chosen as the new root r, and thus the invariant is maintained. □

Then, we show that the open and unseen nodes form a subtree of T.

Lemma 4.13. *The above described procedure maintains the invariant that the open and unseen nodes form a subtree of T.*

Proof. The cases that could change the set of closed nodes are case 2b and case 2c. In the case 2b, by Lemma 4.12 the neighbor of r that is on the path between r and z is open, and the other neighbors of r are closed. Therefore, setting r as closed corresponds to removing a leaf node of the subtree of open and unseen nodes.

In the case 2c, all the neighbors of the nodes removed in the pruned improvement operation are connected to the subtree of new nodes inserted, which are all set to unseen and are connected to the path of open nodes maintained by Lemma 4.12. □

Finally, we put everything together in the following lemma.

Lemma 4.14. *There is an algorithm that, given an n-vertex graph G, integer k, and a cubic tree decomposition $\mathcal{T}$ of G of width w, where $2k+2 \leq w \leq 4k+3$, in time $2^{\mathcal{O}(w)}|\mathcal{T}|$ either outputs a tree decomposition of G of width at most $w-1$ or decides that the treewidth of G is larger than k.*

Proof. The algorithm implements the above described procedure with the data structure of Subsection 4.4.2. We first prove that the algorithm is correct if it terminates, and then show that it indeed terminates in $2^{\mathcal{O}(w)}|\mathcal{T}|$ time.

First, the algorithm is correct when it returns that the treewidth of G is larger than k because in that case we have a set $\mathsf{bag}(r)$ of size $|\mathsf{bag}(r)| = w+1 \geq 2k+3$ that is not splittable, and by Lemma 4.1 this implies that the treewidth of G is larger than k. Second, consider the case that the algorithm terminates in the case 2a. In this case, by Lemma 4.12, the only open node is the node $r = z$, and as all neighbors of r are closed, Lemma 4.13 guarantees that all nodes except r are closed. As a node can get closed only in case 2b, in which case the bag of the node is guaranteed to have size at most w, this implies that all bags in this case must have size at most w, and therefore the width of $\mathcal{T}$ must be at most $w-1$. Therefore the algorithm is correct if it terminates.

By Lemma 4.10, the total number of editable nodes over the course of the algorithm is at most $\Phi_w(\mathcal{T}) = 2^{\mathcal{O}(w)}|\mathcal{T}|$, and therefore the total number of nodes created by pruned improvement operations is also at most $2^{\mathcal{O}(w)}|\mathcal{T}|$. It also implies that the total time spent in case 2c of the procedure is bounded by $2^{\mathcal{O}(w)}\Phi_w(T) = 2^{\mathcal{O}(w)}|\mathcal{T}|$.

For bounding the Move operations applied in case 1 and case 2b of the procedure, observe that both of them advance the state of a node either from unseen to open, or from open to closed. Therefore, the number of Move operations in these cases is bounded by two times the total number of nodes over the course of the algorithm, which is bounded by $\mathcal{O}(|\mathcal{T}| + \Phi_w(\mathcal{T}))$. This gives a total running time of $2^{\mathcal{O}(w)}|\mathcal{T}|$ for the algorithm. □

By using Lemmas 2.13 and 2.14 we can assume that $|\mathcal{T}| = \mathcal{O}(n)$. Therefore, Lemma 4.14 together with Theorem 3.8 implies Theorem 1.1.

4.4.4 Analysis of the $2^{\mathcal{O}(k)}$ factor

We briefly give an upper bound for the $2^{\mathcal{O}(k)}$ factor in the running time of our algorithm, in order to support our claim that this factor in our algorithm is significantly smaller than in the algorithms of Bodlaender et al. [2016a].

First, we show that if the width w of the given tree decomposition is at least $3k+3$, then we can use splits where $C_3 = \emptyset$.

Lemma 4.15. *Let G be a graph of treewidth $\leq k$. Any set of vertices $W \subseteq V(G)$ of size $|W| \geq 3k+4$ has a split of form $(C_1, C_2, \emptyset, S)$.*

Proof. Again, as in Lemma 4.1, let S be a balanced separator of W of size $|S| \leq k+1$, and let us combine the two connected components C_i of $G \setminus S$ with the smallest sizes of $C_i \cap W$ until we obtain a partition $\{C_1, C_2\}$ of $V(G) \setminus S$. By considering the cases of whether there is a component C_i with $|W \cap C_i| \geq |W|/3$ or not, we notice that we will end up with $|W \cap C_i| \leq 2|W|/3$ for both $i \in [2]$. Therefore $(C_1, C_2, \emptyset, S)$ is a split of W because $|(W \cap C_i) \cup S| \leq 2|W|/3 + k + 1 < |W|$. □

Now, if the width w of the input tree decomposition is $w \geq 3k+3$, we apply a version of the algorithm that only considers 2-way splits, i.e., fixes $C_3 = \emptyset$. Note that this also changes the definition of a minimum split to only minimize over splits with $C_3 = \emptyset$, but the proof of Lemma 4.4 still goes through identically, in particular noting that if $C_i = \emptyset$ in the original split, then the C'_i constructed for the contradiction argument will also be empty.

Then, we note that the exponential factors $2^{\mathcal{O}(w)}$ in the running time of the data structure come from the number of ways a partition (C_1, C_2, C_3, S) of $V(G)$ can intersect a bag $\mathsf{bag}(x)$ of size at most $w+1$. This is $\mathcal{O}(4^w)$, and when $C_3 = \emptyset$ this is $\mathcal{O}(3^w)$. Therefore, when $w \geq 3k+3$, the running time of the algorithm is bounded by $(n + \Phi_w(T))3^w w^{\mathcal{O}(1)}$, and when $w < 3k+3$ the running time is bounded by $(n + \Phi_w(T))4^w w^{\mathcal{O}(1)}$.

We note that also the factors $3^{|\mathsf{bag}(x)|}$ in the definition of the potential function can be replaced by factors $2^{|\mathsf{bag}(x)|}$ in the case when $C_3 = \emptyset$. Therefore, for the case when $w \geq 3k+3$, the running time is $2^w 3^w w^{\mathcal{O}(1)} n$, which by $w \leq 4k+3$ is $\mathcal{O}(2^{10.4k} n)$. When $w < 3k+3$, the running time is $3^w 4^w w^{\mathcal{O}(1)} n$, which by $w < 3k+3$ is $\mathcal{O}(2^{10.8k} n)$. Here the factors polynomial in k are dominated by rounding up the exponential dependency on k. Therefore, the total running time of the algorithm is $\mathcal{O}(2^{10.8k} n)$.

Chapter 5

Exact and $(1+\varepsilon)$-approximation algorithms for treewidth

In this chapter we prove the following theorem about computing treewidth exactly.

Theorem 1.2. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k^2)}n^4$ either outputs a tree decomposition of G of width at most k or determines that the treewidth of G is larger than k. Moreover, the algorithm runs in space $n^{\mathcal{O}(1)}$.*

With similar techniques, we also prove the following theorem about approximating treewidth within a factor of $(1+\varepsilon)$, for any given rational ε with $0 < \varepsilon < 1$.

Theorem 1.3. *There is an algorithm that, given an n-vertex graph G, an integer k, and a rational ε with $0 < \varepsilon < 1$, in time $k^{\mathcal{O}(k/\varepsilon)}n^4$ either outputs a tree decomposition of G of width at most $(1+\varepsilon)k$ or determines that the treewidth of G is larger than k. Moreover, the algorithm runs in space $n^{\mathcal{O}(1)}$.*

The proofs of both Theorem 1.2 and Theorem 1.3 cleanly split into two parts. In the first part, we show that algorithms for a problem called *Subset Treewidth* imply algorithms for treewidth, and algorithms for a problem called *Partitioned Subset Treewidth* imply $(1+\varepsilon)$-approximation algorithms for treewidth. Both of these are new problems that we define soon. In the second part, we then give the algorithms for Subset Treewidth and Partitioned Subset Treewidth, which then imply Theorem 1.2 and Theorem 1.3. The first part can be seen as a generalization of the improvement operation introduced in the previous chapter. Related techniques are also used in the second part.

5.1 Subset Treewidth

We then introduce the Subset Treewidth problem. Let G be a graph and $X \subseteq V(G)$. Recall that $\mathsf{torso}_G(X)$ is the graph obtained from $G[X]$ by making $N(C)$ a clique for each $C \in \mathsf{cc}(G \setminus X)$. A *torso tree decomposition* in a graph G is a pair $(X, \mathcal{T})$, where $X \subseteq V(G)$ and $\mathcal{T}$ is a tree decomposition of $\mathsf{torso}_G(X)$. The width of a torso tree decomposition $(X, \mathcal{T})$ is the width of $\mathcal{T}$, and we say that $(X, \mathcal{T})$ *covers* a set $W \subseteq V(G)$ if $X \supseteq W$.

We are now ready to define the Subset Treewidth problem.

Definition 5.1 (Subset Treewidth)**.** *In the Subset Treewidth problem the input is a graph G, integer k, and a set of vertices $W \subseteq V(G)$ of size $|W| = k+2$. The problem is to either return a torso tree decomposition of width at most k that covers W or determine that the treewidth of G is at least $k+1$.*

Note that $\mathsf{torso}_G(V(G)) = G$, which implies that if the treewidth of G is at most k, then any set $W \subseteq V(G)$ can be covered by a torso tree decomposition of width at most k simply by taking $X = V(G)$. In particular, at least one of the cases of Subset Treewidth can always be returned (sometimes either one of them).

In Section 5.2 we show the following connection between treewidth and Subset Treewidth. In the statement, n denotes the number of vertices and k the treewidth.

Theorem 5.2. *Given an algorithm for Subset Treewidth with running time $T(k,n)$, where $T(k,n)$ is increasing in both k and n, an algorithm for treewidth with running time $T(2k,n) \cdot \mathcal{O}(nk) + k^{\mathcal{O}(1)}n^4$ can be constructed. Moreover, if the algorithm for Subset Treewidth runs in space $n^{\mathcal{O}(1)}$, then the algorithm for treewidth runs in space $n^{\mathcal{O}(1)}$.*

Then, in Section 5.5 we give the following algorithm for Subset Treewidth.

Theorem 5.3. *There is a $2^{\mathcal{O}(k^2)}n^2$ time $n^{\mathcal{O}(1)}$ space algorithm for Subset Treewidth.*

Combining Theorems 5.2 and 5.3 yields a $2^{\mathcal{O}(k^2)}n^3 + k^{\mathcal{O}(1)}n^4$ time algorithm for treewidth, in particular, Theorem 1.2.

For the approximation result, we use a variant of Subset Treewidth called Partitioned Subset Treewidth. This variant also turns up naturally when designing an algorithm for Subset Treewidth.

Definition 5.4 (Partitioned Subset Treewidth)**.** *In the Partitioned Subset Treewidth problem the input is a graph G, integer k, set of vertices $W \subseteq V(G)$ of size $|W| = k+2$,*

and a partition $\{W_1, \ldots, W_t\}$ *of* W *into* t *cliques* W_i *of* G*. The problem is to either return a torso tree decomposition of width at most* k *that covers* W *or determine that the treewidth of* G *is at least* $k+1$*.*

Partitioned Subset Treewidth is like Subset Treewidth, but the given set W is partitioned into t cliques, where t is an additional parameter. When t is small, we obtain a more efficient algorithm for Partitioned Subset Treewidth than for Subset Treewidth. In particular, in Section 5.4 we give the following algorithm.

Theorem 5.5. *There is a* $k^{\mathcal{O}(kt)}n^2$ *time* $n^{\mathcal{O}(1)}$ *space algorithm for Partitioned Subset Treewidth.*

This algorithm is useful for $(1+\varepsilon)$-approximation of treewidth via the following relation we prove in Section 5.2.

Theorem 5.6. *Given an algorithm for Partitioned Subset Treewidth with running time* $T(k,t,n)$*, where* $T(k,t,n)$ *is increasing on all* k, t*, and* n*, a* $(1+\varepsilon)$*-approximation algorithm for treewidth with running time* $T(\mathcal{O}(k), \mathcal{O}(1/\varepsilon), n)\cdot\mathcal{O}(kn)\cdot(1+1/\varepsilon)^{\mathcal{O}(k)}+k^{\mathcal{O}(1)}n^4$ *for* $0 < \varepsilon < 1$ *can be constructed. Moreover, if the algorithm for Partitioned Subset Treewidth runs in space* $n^{\mathcal{O}(1)}$*, then the algorithm for treewidth runs in space* $n^{\mathcal{O}(1)}$*.*

Combining Theorems 5.5 and 5.6 yields a $k^{\mathcal{O}(k/\varepsilon)}n^3 + k^{\mathcal{O}(1)}n^4$ time $(1+\varepsilon)$-approximation algorithm for treewidth, in particular, Theorem 1.3. We note that the proof of Theorem 5.5 is simpler than the proof of Theorem 5.3, in particular, the proof of Theorem 5.3 builds on the proof of Theorem 5.5. Therefore, the algorithm of Theorem 5.5 is interesting not only for the approximation result, but also for a simpler proof of Theorem 1.2 with a slightly worse running time.

The rest of this chapter is organized as follows. In Section 5.2 we prove Theorems 5.2 and 5.6. Then in Section 5.3 we presents both known and new lemmas about objects called "important separators", which will be used in the later sections. Then in Section 5.4 we prove Theorem 5.5, and building on that, in Section 5.5 we prove Theorem 5.3.

5.2 Computing treewidth via Subset Treewidth

In this section we show that algorithms for Subset Treewidth and Partitioned Subset Treewidth imply algorithms for treewidth. In particular, we prove Theorems 5.2 and 5.6.

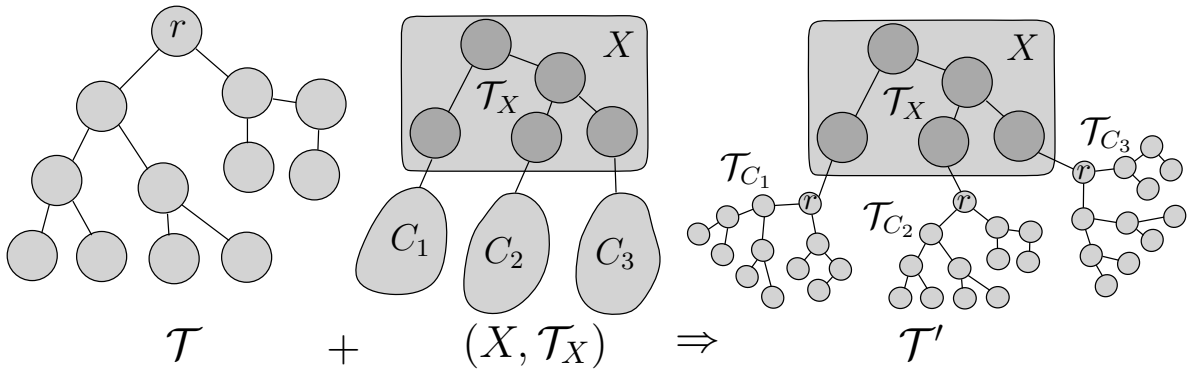

Figure 5.1: Transforming a tree decomposition $\mathcal{T}$ into an improved tree decomposition $\mathcal{T}'$ by using a torso tree decomposition $(X, \mathcal{T}_X)$. Here, $\mathsf{cc}(G \setminus X) = \{C_1, C_2, C_2\}$.

5.2.1 Overview

The main intermediate lemma that implies Theorems 5.2 and 5.6 is that given a solution $(X, \mathcal{T}_X)$ for Subset Treewidth where the set W is a largest bag of a tree decomposition $\mathcal{T}$, we show that $\mathcal{T}$ can be transformed into a tree decomposition $\mathcal{T}'$ that is improved compared to $\mathcal{T}$. In particular, the width of $\mathcal{T}'$ is at most the width of $\mathcal{T}$, and the number of bags of size $|W|$ in $\mathcal{T}'$ is strictly less than in $\mathcal{T}$. This will be more formally stated as Lemma 5.14.

We now briefly sketch the idea of the transformation of $\mathcal{T}$ into $\mathcal{T}'$ (see Figure 5.1). Assume that $W = \mathsf{bag}(r)$ is a largest bag of $\mathcal{T} = (T, \mathsf{bag})$, which is rooted at the node $r \in V(T)$, and let $(X, \mathcal{T}_X)$ be a torso tree decomposition that covers W and has width at most $|W| - 2$. The improved tree decomposition $\mathcal{T}'$ is constructed by starting with $\mathcal{T}_X$, and then for every connected component $C \in \mathsf{cc}(G \setminus X)$ constructing a tree decomposition $\mathcal{T}_C$ and attaching $\mathcal{T}_C$ to $\mathcal{T}_X$. The rooted tree decomposition $\mathcal{T}_C = (T, \mathsf{bag}_C)$ is constructed from $\mathcal{T}$ by first removing all vertices not in $N[C]$, and then forcing the neighborhood $N(C)$ to be in the root bag $\mathsf{bag}_C(r)$ of $\mathcal{T}_C$. This is similar to the construction of the tree decomposition $\mathcal{T}^i$ in Section 4.2, with C playing the role of C_i and $N(C)$ playing the role of S. Then, by the definition of $\mathsf{torso}_G(X)$, every set $N(C)$ for $C \in \mathsf{cc}(G \setminus X)$ is a clique in $\mathsf{torso}_G(X)$, so for each $\mathcal{T}_C$ there exists a bag in $\mathcal{T}_X$ to which $\mathcal{T}_C$ can be attached from its root.

To bound the width of each $\mathcal{T}_C$, we will use similar argument as in Section 4.2, except now we do not start by optimizing any criterion on X, but instead implement the exchange argument algorithmically. This results in a procedure that given $(X, \mathcal{T}_X)$, either improves $\mathcal{T}$ or improves $(X, \mathcal{T}_X)$, which we use iteratively until $\mathcal{T}$ is improved.

We remark that this is a generalization of the improvement operation of the previous chapter. In particular, a split (C_1, C_2, C_3, S) of W, as defined in Section 4.2, can always be turned into a torso tree decomposition $(X, \mathcal{T}_X)$ with $X = W \cup S$ that has width at most $|W| - 2$. The construction of $\mathcal{T}_X$ is simply a star with three leaves, where the bag of the center is S, and the bags of the leaves are $(W \cap C_i) \cup S$ for all $i \in [3]$.

While for 2-approximation such simple solution for Subset Treewidth with $\mathcal{T}_X$ of bounded size is guaranteed to exists, for approximation ratios below 2 we believe that the size of $\mathcal{T}_X$ cannot be bounded, and therefore work in the full generality of Subset Treewidth.

The rest of this section is organized as follows. In Subsection 5.2.2 we prove the "Pulling Lemma" for manipulating torso tree decompositions without increasing their width. Then in Subsection 5.2.3 we show the main intermediate lemma about improving tree decompositions. Finally, in Subsection 5.2.4 we derive Theorems 5.2 and 5.6 from this lemma.

5.2.2 Pulling Lemma

We prove a lemma that will be used throughout this section, Sections 5.4 and 5.5, and also in Chapter 6, to argue that a separator S can be incorporated as a bag of a torso tree decomposition if it satisfies certain properties. We call it the "Pulling Lemma" because the separator S will be "pulled" along disjoint paths into a bag of the tree decomposition. It is closely related to the techniques used in the previous chapter for improving tree decompositions. A similar argument has also been used by [Bodlaender and Koster, 2006].

Before proving the Pulling Lemma, we start with a simple auxiliary lemma.

Lemma 5.7. *Let $(X, (T, \mathsf{bag}))$ be a torso tree decomposition in a graph G, and let $Y \subseteq V(G)$ so that $G[Y]$ is connected. The nodes $\{t \in V(T) \mid \mathsf{bag}(t) \cap Y \neq \emptyset\}$ induce a (possibly empty) connected subtree of T.*

Proof. By the definition of $\mathsf{torso}_G(X)$, any u-v-path in $G[Y]$ with $u, v \in X$ can be mapped into an u-v-path in $\mathsf{torso}_G(X)[Y \cap X]$, and therefore $\mathsf{torso}_G(X)[Y \cap X]$ is connected. Then, the conclusion follows from the corresponding property of tree decompositions (Lemma 2.6). □

Then we prove the Pulling Lemma.

Lemma 5.8 (Pulling Lemma)**.** *Let G be a graph and $(X, (T, \mathsf{bag}))$ a torso tree decomposition in G. Let (A, S, B) be a separation of G so that there exists a node $r \in V(T)$*

so that S is linked into $\mathsf{bag}(r) \cap (S \cup B)$. There exists a torso tree decomposition $((X \cap A) \cup S, (T', \mathsf{bag}'))$ so that

1. *$T' = T$*
2. *for all $t \in V(T)$, $|\mathsf{bag}'(t)| \leq |\mathsf{bag}(t)|$, and*
3. *$S \subseteq \mathsf{bag}'(r)$.*

Moreover, when G, $(X, (T, \mathsf{bag}))$, (A, S, B), and r are given as inputs, the torso tree decomposition $((X \cap A) \cup S, (T', \mathsf{bag}'))$ can be constructed in $k^{\mathcal{O}(1)}(|V(T)| + m)$ time, where k is the width of $(X, (T, \mathsf{bag}))$.

Proof. Index the vertices of S by $S = \{s_1, s_2, \ldots, s_{|S|}\}$. Because S is linked into the set $\mathsf{bag}(r) \cap (S \cup B)$, there are vertex-disjoint paths $P_1, \ldots, P_{|S|}$, so that for each $i \in [|S|]$, P_i is a path from s_i to $\mathsf{bag}(r) \cap (S \cup B)$, and all vertices of P_i are contained in $S \cup B$.

To construct (T', bag'), we set $T' = T$, and for each $t \in V(T)$ we set

$$\mathsf{bag}'(t) = (\mathsf{bag}(t) \setminus (S \cup B)) \cup \{s_i \mid P_i \cap \mathsf{bag}(t) \neq \emptyset\}.$$

We have that $|\mathsf{bag}'(t)| \leq |\mathsf{bag}(t)|$, because for each inserted vertex s_i we removed a vertex in P_i (note that the inserted vertex and the removed vertex could both be the same vertex s_i). By definition every P_i intersects $\mathsf{bag}(r)$, and thus $S \subseteq \mathsf{bag}'(r)$. Denote $X' = (X \cap A) \cup S$. It remains to show that (T', bag') is a tree decomposition of $\mathsf{torso}(X')$.

First, the tree decomposition (T', bag') satisfies the vertex condition because no vertices in $X \cap A$ were removed, and as argued before $S \subseteq \mathsf{bag}'(r)$. Second, (T', bag') satisfies the connectedness condition because the occurrences of vertices in $X \cap A$ were not altered, and by Lemma 5.7 the sets $\{t \in V(T) \mid P_i \cap \mathsf{bag}(t) \neq \emptyset\}$ induce connected subtrees of T.

For the edge condition, consider an edge $uv \in E(\mathsf{torso}(X'))$. There is a path between u and v whose intermediate vertices are contained in $V(G) \setminus X'$. If there would be an intermediate vertex in B, then $u, v \in S$, implying $\{u, v\} \subseteq \mathsf{bag}'(r)$, so it remains to consider the cases where there are no intermediate vertices or all intermediate vertices are in $A \setminus X' = A \setminus X$. It follows that if in this case $u, v \in X$, then $uv \in E(\mathsf{torso}(X))$, so the edge condition of (T', bag') in this case holds by the edge condition of (T, bag). Also if $u, v \in S$, then again $\{u, v\} \subseteq \mathsf{bag}'(r)$, so the remaining case is $uv = s_i v$, where $s_i \in S \setminus X$ and $v \in X \setminus S$. Now, s_i and the intermediate vertices on the path between s_i and v are in a connected component C of $G \setminus X$. Because $v \in X$ and $\mathsf{bag}(r) \subseteq X$, this implies that $N(C)$ contains both v and at least one vertex on the path P_i, and therefore

as $N(C)$ is a clique in $\mathsf{torso}(X)$ there is a node $t \in V(T)$ with $N(C) \subseteq \mathsf{bag}(t)$ and it will hold that $\{s_i, v\} \subseteq \mathsf{bag}'(t)$.

Because (T, bag) has width k and $|S| \leq k+1$, the construction can be implemented in $k^{\mathcal{O}(1)}(|V(T)| + m)$ time. □

Note that the condition $|\mathsf{bag}'(t)| \leq |\mathsf{bag}(t)|$ implies that the width of (T', bag') is at most the width of (T, bag).

5.2.3 Improving a tree decomposition

We will define a weighted version of linkedness. For a weight function $d : V(G) \to \mathbb{Z}$ and a set $S \subseteq V(G)$, we denote $d(S) = \sum_{v \in S} d(v)$.

Definition 5.9 (d-linked)**.** *Let G be a graph, $A, B \subseteq V(G)$, and $d : V(G) \to \mathbb{Z}$ a weight function. The set A is d-linked into B if for every (A, B)-separator S it holds either that $|S| > |A|$, or that $|S| = |A|$ and $d(S) \geq d(A)$.*

Note that if A is d-linked into B then A is linked into B. We say that an (A, B)-separator S with $|S| < |A|$, or with $|S| = |A|$ and $d(S) < d(A)$ *witnesses* that A is not d-linked into B. Then, we say that a torso tree decomposition $(X, (T, \mathsf{bag}))$ is d-linked into a set of vertices $W \subseteq V(G)$ if for every node $t \in V(T)$ it holds that $\mathsf{bag}(t)$ is d-linked into W. We say that a pair (t, S), where $t \in V(T)$ and S is a $(\mathsf{bag}(t), W)$-separator witnessing that $\mathsf{bag}(t)$ is not d-linked into W witnesses that $(X, (T, \mathsf{bag}))$ is not d-linked into W.

Next we will show that if $(X, \mathcal{T})$ is a torso tree decomposition that covers W, then given a pair (t, S) that witnesses that $(X, \mathcal{T})$ is not d-linked into W, we can, in some sense, improve $(X, \mathcal{T})$ while maintaining that it covers W and not increasing its width. We define $\Phi_d(X) = |X| \cdot n(k+1) + d(X)$ as the measure in which sense we will improve $(X, \mathcal{T})$, where n is the number of vertices of G and k is the width of $\mathcal{T}$.

Lemma 5.10. *There is an algorithm, that takes as an input a graph G, a set of vertices $W \subseteq V(G)$, a torso tree decomposition $(X, \mathcal{T})$ in G of width k that covers W, a weight function $d : V(G) \to [n]$, and a pair (t, S) that witnesses that $(X, \mathcal{T})$ is not d-linked into W, and in time $k^{\mathcal{O}(1)}(|V(T)| + m)$ returns a torso tree decomposition $(X', \mathcal{T}')$ that covers W, has width at most k, has at most $|V(T)|$ nodes, and has $\Phi_d(X') < \Phi_d(X)$.*

Proof. Denote $\mathcal{T} = (T, \mathsf{bag})$. After a $k^{\mathcal{O}(1)}m$ time flow computation we may assume that S is a minimum size $(\mathsf{bag}(t), W)$-separator, because if S was not a minimum size

$(\mathsf{bag}(t), W)$-separator then any minimum size $(\mathsf{bag}(t), W)$-separator also witnesses that $\mathsf{bag}(t)$ is not d-linked into W. This implies that S is linked into $\mathsf{bag}(t)$.

Let (A, S, B) be a separation with $W \subseteq A \cup S$ and $\mathsf{bag}(t) \subseteq B \cup S$. Denote $X' = (X \cap A) \cup S$. We apply the Pulling Lemma (Lemma 5.8) with the torso tree decomposition $(X, (T, \mathsf{bag}))$, the separation (A, S, B), and the node t to construct a torso tree decomposition $(X', \mathcal{T}')$ of width at most k and at most $|V(T)|$ nodes. As $W \subseteq X$ and $W \subseteq A \cup S$, we have that $W \subseteq X'$, so $(X', \mathcal{T}')$ covers W. It remains to prove that $\Phi_d(X') < \Phi_d(X)$.

Because $\mathsf{bag}(t) \subseteq S \cup B$ and $\mathsf{bag}(t) \subseteq X$, we have that $|X'| \leq |X| - |\mathsf{bag}(t)| + |S|$ and $d(X') \leq d(X) - d(\mathsf{bag}(t)) + d(S)$. Therefore, if $|S| < |\mathsf{bag}(t)|$, then $|X'| < |X|$, implying $\Phi_d(X') < \Phi_d(X)$ because $d(S) < n(k+1)$. If $|S| = |\mathsf{bag}(t)|$ and $d(S) < d(\mathsf{bag}(t))$, then $|X'| \leq |X|$ and $d(X') < d(X)$, implying $\Phi_d(X') < \Phi_d(X)$. □

Then, our goal is to show that either a torso tree decomposition $(X, \mathcal{T}_X)$ of width $k-1$ that covers a largest bag W of a tree decomposition $\mathcal{T}$ of width k can be used to improve $\mathcal{T}$, or we find a pair (t, S) witnessing that $(X, \mathcal{T}_X)$ is not d-linked into W for a certain function d, in which case we can improve $(X, \mathcal{T}_X)$ by applying Lemma 5.10. The proof will use similar ideas as we used in Section 4.2.

Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition of G rooted at a node $r \in V(T)$. We define the weight function $d_{\mathcal{T}} \colon V(G) \to [|V(T)|]$ so that $d_{\mathcal{T}}(v) = \mathsf{depth}_T(\mathsf{forget}_{\mathcal{T}}(v)) + 1$, i.e., as the vertex-depth function plus one. Note that this is similar to the function used for the definition of a minimum split in Subsection 4.2.1.

Lemma 5.11. *Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition of G of width k, rooted at a node r with $\mathsf{bag}(r) = W$ and $|W| = k+1$. There is an algorithm that given a torso tree decomposition $(X, \mathcal{T}_X)$, where $\mathcal{T}_X = (T_X, \mathsf{bag}_X)$, that covers W and has width at most $k-1$, in time $k^{\mathcal{O}(1)}(|\mathcal{T}| + |\mathcal{T}_X|)$ either*

1. *constructs a tree decomposition of G of width at most k, having strictly less bags of size $k+1$ than $\mathcal{T}$, and having at most n nodes, or*
2. *returns a pair (t, S) where $t \in V(T_X)$ and $S \subseteq V(G)$ that witnesses that $(X, \mathcal{T}_X)$ is not $d_{\mathcal{T}}$-linked into W.*

Proof. Our goal is to construct a tree decomposition $\mathcal{T}' = (T', \mathsf{bag}')$ of G, and then show that if it does not satisfy the conditions of Item 1, then we find the pair (t, S) of Item 2.

First, for every connected component C of $G \setminus X$, we will construct a tree decomposition $\mathcal{T}_C = (T_C, \mathsf{bag}_C)$ of $N[C]$, so that $N(C)$ is in the root bag of $\mathcal{T}_C$. For a node $t \in V(T)$,

denote by $\mathsf{pull}(t, C)$ the vertices

$$\mathsf{pull}(t, C) = \{v \in N(C) \mid \mathsf{forget}_{\mathcal{T}}(v) \text{ is a strict descendant of } t \text{ in } T\}.$$

To construct the tree decomposition $\mathcal{T}_C$, we first set

$$T_C = T[\{t \in V(T) \mid C \cap \mathsf{bag}(t) \neq \emptyset\}],$$

i.e., T_C is the subtree of T induced by the bags that intersect C. Observe that T_C is connected because $G[C]$ is connected. Then for each $t \in V(T_C)$ we set

$$\mathsf{bag}_C(t) = (\mathsf{bag}(t) \cap N[C]) \cup \mathsf{pull}(t, C).$$

We let the root node of $\mathcal{T}_C$ to be the node $r_C \in V(T_C)$ that is the closest to r in T. Note that because T_C is a connected subtree of T, the node r_C is uniquely defined.

Claim 5.12. *$\mathcal{T}_C$ is a tree decomposition of $N[C]$ and $N(C) \subseteq \mathsf{bag}_C(r_C)$.*

Proof of the claim. First, for the vertices C and edges in $G[C]$ the decomposition clearly satisfies the vertex, edge, and connectedness conditions because $\mathcal{T}$ satisfied these conditions. For the edge condition for edges between C and $N(C)$ and the vertex condition for vertices in $N(C)$, note that each such edge must be in a bag that intersects C, and because for every vertex of $N(C)$ there exists such an edge we have that every vertex of $N(C)$ must occur in some bag that intersects C.

It remains to prove the connectedness condition for vertices in $N(C)$ and the edge condition for edges in $G[N(C)]$. For a vertex $v \in N(C)$, either (1) $v \in \mathsf{bag}(r_C)$ and v is not in $\mathsf{pull}(t, C)$ for any $t \in V(T_C)$, or (2) $\mathsf{forget}_{\mathcal{T}}(v) \in V(T_C) \setminus \{r_C\}$ and $v \in \mathsf{pull}(t, C)$ for all t on the path from the parent of $\mathsf{forget}_{\mathcal{T}}(v)$ to the root r_C. Therefore, the connectedness condition is maintained for vertices in $N(C)$. This also shows that $N(C) \subseteq \mathsf{bag}_C(r_C)$, which implies the edge condition for edges in $G[N(C)]$. ◁

Now, our complete construction of $\mathcal{T}'$ is to attach the tree decompositions $\mathcal{T}_C$ for all components C of $G \setminus X$ from their roots to the tree decomposition $\mathcal{T}_X$. Because $N(C)$ is a clique in $\mathsf{torso}(X)$, the decomposition $\mathcal{T}_X$ contains a bag containing $N(C)$ to which $\mathcal{T}_C$ can be attached.

Next we show that this construction can be implemented in $k^{\mathcal{O}(1)}(|\mathcal{T}| + |\mathcal{T}_X|)$ time. In particular, first, the connected components C and their neighborhoods can be found in $k^{\mathcal{O}(1)}m = k^{\mathcal{O}(1)}|\mathcal{T}|$ time. Then, we observe that the sum of $|\mathcal{T}_C|$ over all components C is at most $(k+1)|\mathcal{T}|$ because $\mathcal{T}$ has width k and the components C are disjoint. By first

computing pointers from vertices of G to bags containing them, and then using the fact that $|N(C)| \leq k+1$, each tree decomposition $\mathcal{T}_C$ can be constructed in $k^{\mathcal{O}(1)}|\mathcal{T}_C|$ time, which sums up to $k^{\mathcal{O}(1)}|\mathcal{T}|$. Then, it remains to attach each tree decomposition $\mathcal{T}_C$ to a node of $\mathcal{T}_X$ whose bag contains $N(C)$. For this, observe that if we consider $\mathcal{T}_X$ rooted, then $N(C)$ is contained in the bag of the node $\mathsf{forget}_{\mathcal{T}_X}(v)$ for the $v \in N(C)$ for which $\mathsf{forget}_{\mathcal{T}_X}(v)$ maximizes $\mathsf{depth}_{T_X}(\mathsf{forget}_{\mathcal{T}_X}(v))$.

Next we give the main argument for extracting the witness of Item 2 if (T', bag') does not satisfy Item 1.

Claim 5.13. *Let C be a component of $G \setminus X$ and $x \in V(T_X)$ a node of $\mathcal{T}_X$ with $N(C) \subseteq \mathsf{bag}_X(x)$. For every node $t \in V(T_C)$ we have either that*

(i) $|\mathsf{bag}_C(t)| < |\mathsf{bag}(t)|$ *or* $\mathsf{bag}_C(t) = \mathsf{bag}(t)$*, or that*

(ii) the set $(\mathsf{bag}_X(x) \setminus \mathsf{pull}(t,C)) \cup (\mathsf{bag}(t) \setminus N[C])$ *witnesses that* $\mathsf{bag}_X(x)$ *is not $d_{\mathcal{T}}$-linked into W.*

Proof of the claim. Item (i) is true if $\mathsf{pull}(t,C)$ is empty, so suppose $\mathsf{pull}(t,C)$ is non-empty and $|\mathsf{bag}_C(t)| \geq |\mathsf{bag}(t)|$. By the definition of $\mathsf{bag}_C(t)$ this implies $|\mathsf{pull}(t,C)| \geq |\mathsf{bag}(t) \setminus N[C]|$. Note that $\mathsf{pull}(t,C) \subseteq \mathsf{bag}_X(x)$. We will show that in this case

$$S = (\mathsf{bag}_X(x) \setminus \mathsf{pull}(t,C)) \cup (\mathsf{bag}(t) \setminus N[C])$$

separates $\mathsf{bag}_X(x)$ from W. Therefore S witnesses that $\mathsf{bag}_X(x)$ is not $d_{\mathcal{T}}$-linked into W, because by $|\mathsf{pull}(t,C)| \geq |\mathsf{bag}(t) \setminus N[C]|$ we have that $|S| \leq |\mathsf{bag}_X(x)|$, and moreover we have $d_{\mathcal{T}}(S) < d_{\mathcal{T}}(\mathsf{bag}_X(x))$ because for every vertex $v_1 \in \mathsf{pull}(t,C)$ and $v_2 \in \mathsf{bag}(t)$, it holds that $d_{\mathcal{T}}(v_1) > d_{\mathcal{T}}(v_2)$ because $\mathsf{forget}_{\mathcal{T}}(v_1)$ is a strict descendant of t while $\mathsf{forget}_{\mathcal{T}}(v_2)$ is an ancestor of t.

To show that S separates $\mathsf{bag}_X(x)$ from W, it is sufficient to show that it separates $\mathsf{pull}(t,C)$ from W because $\mathsf{bag}_X(x) \setminus S = \mathsf{pull}(t,C)$. Consider a shortest path in $G \setminus S$ that starts in $\mathsf{pull}(t,C)$ and ends in W. If this path would intersect $N[C]$ anywhere else than in its first vertex, then it would intersect $\mathsf{pull}(t,C)$ twice because $N(C) \setminus S = \mathsf{pull}(t,C)$ and $W \cap C = \emptyset$, which would contradict that it is a shortest path. Therefore, it intersects $N[C]$ only in its first vertex. Then, because for each $v \in \mathsf{pull}(t,C)$ the node $t \in V(T)$ separates $\mathsf{forget}_{\mathcal{T}}(v)$ from r in T, it holds that $\mathsf{bag}(t)$ separates $\mathsf{pull}(t,C)$ from $\mathsf{bag}(r) = W$. Therefore, the path must intersect $\mathsf{bag}(t)$, and therefore as $\mathsf{bag}(t)$ and $\mathsf{pull}(t,C)$ are disjoint, it must intersect $\mathsf{bag}(t) \setminus N[C]$. However, $\mathsf{bag}(t) \setminus N[C] \subseteq S$, and therefore no such path exists in $G \setminus S$. ◁

Now, for all nodes of the constructed decompositions $\mathcal{T}_C$ we check if Item (i) of Claim 5.13 holds, and if it does not hold we return the pair $(x, (\mathsf{bag}_X(x) \setminus \mathsf{pull}(t, C)) \cup (\mathsf{bag}(t) \setminus N[C]))$. This can be done in $k^{\mathcal{O}(1)}|\mathcal{T}|$ time.

Then, it remains to prove that if Item (i) of Claim 5.13 holds for all nodes of all of the decompositions $\mathcal{T}_C$, then $\mathcal{T}'$ has width at most k and has strictly less bags of size $k+1$ than $\mathcal{T}$. First, clearly $\mathcal{T}'$ has width at most k as none of the decompositions $\mathcal{T}_C$ have larger width than $\mathcal{T}$ and $\mathcal{T}_X$ has smaller width than $\mathcal{T}$. It remains to prove that $\mathcal{T}'$ has less bags of size $k+1$ than $\mathcal{T}$.

Consider any node $t \in V(T)$, and suppose that there are two distinct components C_1 and C_2 of $G \setminus X$ so that both C_1 and C_2 intersect $\mathsf{bag}(t)$ and $|\mathsf{bag}_{C_1}(t)| = |\mathsf{bag}_{C_2}(t)| = |\mathsf{bag}(t)|$. Now, by Item (i) of Claim 5.13 it holds that $\mathsf{bag}_{C_1}(t) = \mathsf{bag}_{C_2}(t) = \mathsf{bag}(t)$. Therefore, as $\mathsf{bag}_{C_1}(t) \subseteq N[C_1]$ and $\mathsf{bag}_{C_2}(t) \subseteq N[C_2]$, it holds that $\mathsf{bag}(t) \subseteq N(C_1) \cap N(C_2)$, which implies that $\mathsf{bag}(t)$ is a clique in $\mathsf{torso}(X)$, and therefore has size at most k. Therefore, for any node $t \in V(T)$ with a bag of size $|\mathsf{bag}(t)| = k+1$, there is at most one corresponding node t in the decompositions $\mathcal{T}_C$ across all components C with a bag of size $|\mathsf{bag}_C(t)| = k+1$. For the root node r, as $\mathsf{bag}(r) \subseteq X$, none of the components C intersect $\mathsf{bag}(r)$, and therefore no decomposition $\mathcal{T}_C$ contains a node corresponding to it. All other bags of $\mathcal{T}'$ are from $\mathcal{T}_X$ and have size at most k, so as $|\mathsf{bag}(r)| = k+1$, it follows that $\mathcal{T}'$ has strictly less bags of size $k+1$ than $\mathcal{T}$.

Finally, by Lemma 2.13 we can reduce the number of nodes of $\mathcal{T}'$ to at most n within the same time without increasing the width or the number of bags of size $k+1$. □

Then, we combine Lemmas 5.10 and 5.11 into a single lemma showing that to improve $\mathcal{T}$ it is sufficient to find a torso tree decomposition in G that covers a largest bag of $\mathcal{T}$ and has width smaller than $\mathcal{T}$.

Lemma 5.14. *Let $\mathcal{T} = (T, \mathsf{bag})$ be a tree decomposition of G of width k and $|\mathcal{T}| \leq n$, rooted at a node $r \in V(T)$ with $\mathsf{bag}(r) = W$ and $|W| = k+1$. There is an algorithm that given a torso tree decomposition $(X, \mathcal{T}_X)$ that covers W and has width at most $k-1$, in time $k^{\mathcal{O}(1)}(|\mathcal{T}_X| + n^3)$ constructs a tree decomposition of G of width at most k, having strictly less bags of size $k+1$ than $\mathcal{T}$, and having at most n nodes.*

Proof. First, we apply Lemma 2.13 to reduce the number of nodes of $\mathcal{T}_X$ to at most n. Then, we repeatedly apply Lemma 5.11 together with Lemma 5.10. In particular, if Lemma 5.11 returns the tree decomposition of Item 1 we are done, and if it returns a pair (t, S) that witnesses that $(X, \mathcal{T}_X)$ is not $d_{\mathcal{T}}$-linked into W then we apply Lemma 5.10, which decreases $\Phi_{d_{\mathcal{T}}}(X)$ by at least one. Because $\Phi_{d_{\mathcal{T}}}(X)$ is initially $\mathcal{O}(kn^2)$ and $\Phi_{d_{\mathcal{T}}}(X)$

must be non-negative, the total number of iterations is at most $\mathcal{O}(kn^2)$, giving a total running time of $k^{\mathcal{O}(1)}n^3$ plus $k^{\mathcal{O}(1)}|\mathcal{T}_X|$ from the application of Lemma 2.13. □

5.2.4 Reducing treewidth to Subset Treewidth

Now we can prove Theorem 5.2, in particular, that algorithms for Subset Treewidth imply algorithms for treewidth.

Theorem 5.2. *Given an algorithm for Subset Treewidth with running time $T(k,n)$, where $T(k,n)$ is increasing in both k and n, an algorithm for treewidth with running time $T(2k,n)\cdot\mathcal{O}(nk)+k^{\mathcal{O}(1)}n^4$ can be constructed. Moreover, if the algorithm for Subset Treewidth runs in space $n^{\mathcal{O}(1)}$, then the algorithm for treewidth runs in space $n^{\mathcal{O}(1)}$.*

Proof. Our goal is to provide the algorithm $\mathcal{A}$ of Theorem 3.8 with $\alpha=1$ and $T_{\mathcal{A}}(k,n)=T(2k,n)\cdot\mathcal{O}(nk)+k^{\mathcal{O}(1)}n^4$. In particular, by Theorem 3.8 we can assume that the input contains a tree decomposition of width at most $2k+1$, and our goal is to improve its width to k or to determine that the treewidth of the input graph G is larger than k. Let $\mathcal{T}=(T,\mathsf{bag})$ be the input tree decomposition. By Lemma 2.13 we assume that $|\mathcal{T}|\leq n$.

Then, we repeat the following process as long as the width of $\mathcal{T}$ is larger than k. Let $W=\mathsf{bag}(r)$ be a largest bag of $\mathcal{T}$, and note that in this case $|W|\geq k+2$. We use the algorithm for Subset Treewidth to either get a torso tree decomposition that covers W and has width at most $|W|-2$ or to conclude that the treewidth of G is larger than $|W|-2\geq k$. If we conclude that the treewidth of G is larger than k we are ready and can immediately return. If the algorithm returns such a torso tree decomposition, we apply Lemma 5.14 to improve $\mathcal{T}$, in particular, to decrease the number of bags of size $|W|$ and not increase the width, and maintain that $|\mathcal{T}|\leq n$.

We can decrease the number of largest bags while not increasing the width at most $n\cdot(k+1)$ times before the width decreases from $2k+1$ to k, and therefore the algorithm works with $\mathcal{O}(nk)$ applications of the algorithm for Subset Treewidth and Lemma 5.14. In all of the applications, the parameter k for Subset Treewidth is at most $2k$, where k is the original parameter for treewidth. The total running time from the applications of Subset Treewidth is therefore $T(2k,n)\cdot\mathcal{O}(nk)$, and the total running time from the applications of Lemma 5.14 is $k^{\mathcal{O}(1)}n^3\cdot\mathcal{O}(kn)=k^{\mathcal{O}(1)}n^4$, giving the desired running time. □

We then turn to Theorem 5.6, in particular, to proving that algorithms for Partitioned Subset Treewidth imply approximation algorithms for treewidth. The main idea for this

is to show that we can always partition W into t parts and make them cliques, while increasing the treewidth of G by only $\mathcal{O}(|W|/t)$.

Lemma 5.15. *Let G be a graph of treewidth at most k, ε a rational with $0 < \varepsilon < 1$, and $W \subseteq V(G)$ a set of vertices of size $|W| \leq 4k + 4$. There exists a partition of W into $t = \mathcal{O}(1/\varepsilon)$ parts $W_1, \ldots, W_t$, so that after making each part into a clique the treewidth of G is at most $k + \varepsilon k$.*

Proof. If $\varepsilon < 1/k$ we can return the trivial partition of W into single vertices. Therefore we can assume that $\varepsilon k \geq 1$.

Consider a rooted tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ of G of width k, and assume that $\mathcal{T}$ is a nice tree decomposition (recall the definition from Subsection 2.3.1). Note that because $\mathcal{T}$ is nice, each node of $\mathcal{T}$ is a forget-node of at most one vertex of G. By further subdividing $\mathcal{T}$, we also assume that each forget-node has exactly one child. We say that a node is a W-forget node if it is a forget-node of a vertex $w \in W$. Let us process $\mathcal{T}$ from the leaves towards the root, i.e., in the post-order, and maintain a set of "deleted" nodes $D \subseteq V(T)$.

Suppose we are processing a node $t \in V(T)$ and let $R \subseteq \mathsf{desc}_T(t)$ be the nodes of T that are descendants of t and reachable from t in $T \setminus D$. Note that $t \in R$ and $R \subseteq V(T) \setminus D$. Now, if R contains at least $\varepsilon k/2$ W-forget-nodes or t is the root we add a part to the partition of W and modify the tree decomposition as follows. We let $W' \subseteq W$ be the vertices in W whose forget-nodes are in R. We add W' as a part of the partition, and add W' to the bags of all nodes in R. Then, we add all nodes in R to D.

Observe that $|W'| \leq \varepsilon k$ follows from the facts that we process the tree in post-order, each node can have at most two children, each node can be a forget-node of at most one vertex, each forget-node has one child, and $\varepsilon k \geq 1$. Therefore, the sizes of the bags of nodes in R increased by at most εk, and moreover they will not increase again because they were added to D. Therefore, the resulting tree decomposition has width at most $k + \varepsilon k$. We also observe that the resulting tree decomposition is indeed a tree decomposition after making such W' into a clique: All the new edges are contained in the bags of all nodes in R, and the subtree condition is maintained because the forget-nodes of vertices in W' are in R.

Now, each created part of the partition except the part corresponding to the root has size at least $\varepsilon k/2$, so in total the number of parts is at most $|W|/(\varepsilon k/2)+1 \leq \frac{16}{\varepsilon}+1 = \mathcal{O}(1/\varepsilon)$. □

Now, by using Lemma 5.15 we can prove Theorem 5.6 similarly to Theorem 5.2.

Theorem 5.6. *Given an algorithm for Partitioned Subset Treewidth with running time $T(k,t,n)$, where $T(k,t,n)$ is increasing on all k,t, and n, a $(1+\varepsilon)$-approximation algorithm for treewidth with running time $T(\mathcal{O}(k),\mathcal{O}(1/\varepsilon),n)\cdot\mathcal{O}(kn)\cdot(1+1/\varepsilon)^{\mathcal{O}(k)}+k^{\mathcal{O}(1)}n^4$ for $0<\varepsilon<1$ can be constructed. Moreover, if the algorithm for Partitioned Subset Treewidth runs in space $n^{\mathcal{O}(1)}$, then the algorithm for treewidth runs in space $n^{\mathcal{O}(1)}$.*

Proof. Our goal is to provide the algorithm $\mathcal{A}$ of Theorem 3.8 with $\alpha=(1+\varepsilon)$ and $T_{\mathcal{A}}(k,n)=T(\mathcal{O}(k),\mathcal{O}(1/\varepsilon),n)\cdot\mathcal{O}(nk)+k^{\mathcal{O}(1)}n^4$. In particular, by Theorem 3.8 we can assume that the input contains a tree decomposition of width at most $2\cdot(1+\varepsilon)\cdot(k+1)-1\leq 4k+3$, and our goal is to improve its width to at most $(1+\varepsilon)k$ or to determine that the treewidth of the input graph G is larger than k. Let $\mathcal{T}=(T,\mathsf{bag})$ be the input tree decomposition. By Lemma 2.13 we assume that $|\mathcal{T}|\leq n$.

Then, we repeat the following process as long as the width of (T,bag) is larger than $k+\varepsilon k$. Let W be a largest bag of (T,bag), and note that in this case $k+\varepsilon k+2\leq|W|\leq 4k+4$. We try all partitions of W into $t=\mathcal{O}(1/\varepsilon)$ parts (where the bound for t is from Lemma 5.15). For each partition $W_1,\ldots,W_t$, we make the parts $W_1,\ldots,W_t$ into cliques in G, and then use the algorithm for Partitioned Subset Treewidth with this partition of W. By Lemma 5.15, there exists such a partition so that after making $W_1,\ldots,W_t$ into cliques the treewidth of G is at most $k+\varepsilon k$, and therefore if the algorithm for Partitioned Subset Treewidth returns for every partition that the treewidth of G is larger than $|W|-2\geq k+\varepsilon k$, we can return that the treewidth of G is larger than k. Otherwise, the algorithm for Partitioned Subset Treewidth returned a torso tree decomposition that covers W and has width at most $|W|-2$, and we proceed applying Lemma 5.14 similarly as in the proof of Theorem 5.2.

The running time follows from the fact that there are at most $(1+1/\varepsilon)^{\mathcal{O}(k)}$ partitions of W into $t=\mathcal{O}(1/\varepsilon)$ parts, and we can decrease the number of largest bags while not increasing the width at most $\mathcal{O}(nk)$ times, and therefore there we use in total $\mathcal{O}(nk)\cdot(1+1/\varepsilon)^{\mathcal{O}(k)}$ applications of the algorithm for Partitioned Subset Treewidth, with $t=\mathcal{O}(1/\varepsilon)$ and k at most $4k+2$, where k is the original parameter for treewidth. □

5.3 Important separators

Before going into the algorithms for Subset Treewidth and Partitioned Subset Treewidth, in this section we provide preliminary results about objects called *important separators*. The algorithms for Subset Treewidth and Partitioned Subset Treewidth will extensively make use of them. Most of the results given in this section are from the prior literature,

but we believe that Lemmas 5.23 and 5.24 are new, although they are not very difficult to prove.

The notion of important separator was introduced by Marx [2006]. Before giving the definition, let us start with some auxiliary definitions. Let G be a graph, $A, B \subseteq V(G)$, and S an (A,B)-separator in G. We say that S is a *minimal* (A,B)*-separator* if no subset of S is an (A,B)-separator. When $A, S \subseteq V(G)$, we denote by $\mathsf{reach}_G(A,S) \subseteq V(G) \setminus S$ the set of vertices reachable from $A \setminus S$ in $G \setminus S$. Note that if $A \subseteq S$, then $\mathsf{reach}_G(A,S) = \emptyset$. We define $\mathsf{reach}^N_G(A,S) = (A \cap S) \cup N_G(\mathsf{reach}_G(A,S)) \subseteq S$ to denote the subset of S that can be seen from A. Note that if S is an (A,B)-separator, then $\mathsf{reach}^N_G(A,S)$ is also an (A,B)-separator and $\mathsf{reach}_G(A, \mathsf{reach}^N_G(A,S)) = \mathsf{reach}_G(A,S)$. It follows that if S is a minimal (A,B)-separator, then $S = \mathsf{reach}^N_G(A,S) = \mathsf{reach}^N_G(B,S)$. We may omit the graph G from the subscript if it is clear from the context. Then we are ready to define important separators.

Definition 5.16 (Important separator)**.** *Let $A, B \subseteq V(G)$ be two sets of vertices. A minimal (A,B)-separator S is called an important (A,B)-separator if there exists no (A,B)-separator S' with $|S'| \leq |S|$ and $\mathsf{reach}(A,S) \subset \mathsf{reach}(A,S')$.*

We remark that Definition 5.16 allows an important (A,B)-separator to be the empty set in the case when B is not reachable from A in G, or when A or B is empty.

The following lemma is a straightforward consequence of Definition 5.16.

Lemma 5.17. *For every (A,B)-separator S, there exists an important (A,B)-separator S' so that $|S'| \leq |S|$ and $\mathsf{reach}(A,S) \subseteq \mathsf{reach}(A,S')$.*

Proof. Take an (A,B)-separator S' with $|S'| \leq |S|$ and $\mathsf{reach}(A,S) \subseteq \mathsf{reach}(A,S')$ that maximizes $|\mathsf{reach}(A,S')|$, and subject to that minimizes $|S'|$. The separator S' is a minimal (A,B)-separator because deleting vertices from S' does not make $|\mathsf{reach}(A,S')|$ smaller. Then, if S' was not an important separator, the definition of important separators would give an (A,B)-separator S'' that would contradict the choice of S'. □

We say that an important (A,B)-separator S' *dominates* an (A,B)-separator S if S' satisfies the conditions of Lemma 5.17. For our algorithm, we need a property that a smallest important separator that dominates S is linked into S in a certain way.

Lemma 5.18. *Let S be an (A,B)-separator and S' a smallest important (A,B)-separator that dominates S. It holds that S' is linked into $S \cap (\mathsf{reach}(A,S') \cup S')$.*

Proof. Denote $S'' = S \cap (\mathsf{reach}(A,S') \cup S')$. Note that $\mathsf{reach}^N(A,S) \subseteq \mathsf{reach}(A,S') \cup S'$, which implies that $\mathsf{reach}^N(A,S) \subseteq S''$, which implies that S'' is an (A,S')-separator

and $\mathsf{reach}(A, S'') = \mathsf{reach}(A, S)$. Suppose that S' is not linked into S'', and let $S^\star$ be a minimum-size (S'', S')-separator. Now, $S^\star$ is an (A, B)-separator of size $|S^\star| < |S'|$ and because it is minimal (S'', S')-separator and S' is disjoint from $\mathsf{reach}(A, S'')$, it is disjoint from $\mathsf{reach}(A, S'') = \mathsf{reach}(A, S)$, implying $\mathsf{reach}(A, S) \subseteq \mathsf{reach}(A, S^\star)$. Therefore, an important (A, B)-separator that dominates $S^\star$ would also dominate S and be smaller than S', which contradicts the choice of S'. □

We also need the following observation about minimal separators.

Lemma 5.19. *If S is a minimal (A, B)-separator and S' an (A, B)-separator with $\mathsf{reach}(A, S) \subseteq \mathsf{reach}(A, S')$, then S' is an (S, B)-separator.*

Proof. This is implied by $S = \mathsf{reach}^N(A, S)$ and $\mathsf{reach}^N(A, S) \subseteq \mathsf{reach}(A, S') \cup S'$. □

We will then prove upper bounds on the numbers of important separators and give enumeration algorithms for them. The proofs of Lemmas 5.20 and 5.22 are from the literature, but we present them here to make this chapter self-contained and to give context for the similar proofs of Lemmas 5.23 and 5.24. The basic tool for proving bounds on important separators will be the following property of important separators of minimum size given by Marx [2006].

Lemma 5.20 (Marx [2006])**.** *For any $A, B \subseteq V(G)$, there exists exactly one important (A, B)-separator S of size $|S| = \mathsf{flow}(A, B)$. Moreover, for this S, the set of important (A, B)-separators is equal to the set of important (S, B)-separators. Also, this S can be found in time $\mathcal{O}(|S| \cdot m)$.*

Proof. Assume there exists an important (A, B)-separator S_1 of size $\mathsf{flow}(A, B)$, and an important (A, B)-separator S_2 so that $\mathsf{reach}(A, S_1) \not\subseteq \mathsf{reach}(A, S_2)$. Define $S_A = \mathsf{reach}^N(A, S_1 \cup S_2)$ and $S_B = \mathsf{reach}^N(B, S_1 \cup S_2)$. We will show that S_B contradicts the fact that S_2 is an important (A, B)-separator.

We observe that if a vertex v is in both S_A and S_B, then v is in both S_1 and S_2, because if v was not in S_i for $i \in [2]$, then S_i would not be an (A, B)-separator because there would be an A-B-path in $G \setminus S_i$ by first taking the path in $\mathsf{reach}(A, S_1 \cup S_2)$ from A to v, and then taking the path in $\mathsf{reach}(B, S_1 \cup S_2)$ from v to B. As $S_A \cup S_B \subseteq S_1 \cup S_2$, it follows that $|S_A| + |S_B| \leq |S_1| + |S_2|$. Because S_A is an (A, B)-separator we have $|S_A| \geq |S_1|$, implying $|S_B| \leq |S_2|$.

We observe that $\mathsf{reach}(A, S_B) \supseteq \mathsf{reach}(A, S_1) \cup \mathsf{reach}(A, S_2)$. By the assumption $\mathsf{reach}(A, S_1) \not\subseteq \mathsf{reach}(A, S_2)$, it follows that $\mathsf{reach}(A, S_2) \subset \mathsf{reach}(A, S_B)$. This contradicts the fact that S_2 is an important (A, B)-separator.

Therefore, if S is an important (A, B)-separator of size $\mathsf{flow}(A, B)$, then for every other important (A, B)-separator S' it holds that $\mathsf{reach}(A, S) \subseteq \mathsf{reach}(A, S')$. It follows that there is a unique important (A, B)-separator S of size $\mathsf{flow}(A, B)$. By Lemma 5.19 all other important (A, B)-separators are (S, B)-separators. As $S = \mathsf{reach}^N(A, S)$ and for every minimal (S, B)-separator S' it holds that $\mathsf{reach}(A, S') \supseteq \mathsf{reach}(A, S)$, it follows that the set of important (A, B)-separators is equal to the set of important (S, B)-separators.

For finding such S of size at most k in $\mathcal{O}(|S| \cdot m)$ time, we note that if we apply the Ford-Fulkerson algorithm for computing $\mathsf{flow}(B, A)$ with a reduction that makes two copies of each vertex, and so that the source corresponds to B, then such S corresponds to the vertices whose one copy is reachable from the source in the residual graph and the another copy is not. □

All three main lemmas of this section (Lemmas 5.22 to 5.24) will be based on the recursion provided by the following lemma.

Lemma 5.21 (Chen et al. [2009]; Marx and Razgon [2014])**.** *Let S be the important (A, B)-separator of size $|S| = \mathsf{flow}(A, B)$ and $v \in S \setminus B$. The set of important (A, B)-separators S' with $v \notin S'$ is equal to the set of important $(S \cup N(v), B)$-separators.*

Proof. As $\mathsf{reach}(A, S) \subset \mathsf{reach}(A, S')$ and $v \in \mathsf{reach}^N(A, S)$, we have $v \in \mathsf{reach}(A, S')$, implying that every important (A, B)-separator S' with $v \notin S'$ is a $(S \cup N(v), B)$-separator, and in particular an important $(S \cup N(v), B)$-separator, because every $(S \cup N(v), B)$-separator is also an (A, B)-separator. Moreover, because no important $(S \cup N(v), B)$-separator contains v, no such separator S' can be dominated by a separator S'' with $v \in S''$, implying that the important $(S \cup N(v), B)$-separators are exactly the important (A, B)-separators S' with $v \notin S'$. □

The following upper bound on the number of important separators was given implicitly by Chen et al. [2009] and explicitly in Marx and Razgon [2014].

Lemma 5.22 (Chen et al. [2009]; Marx and Razgon [2014])**.** *For any $A, B \subseteq V(G)$, there are at most 4^k important (A, B)-separators of size at most k, and they can be enumerated in time $4^k k^{\mathcal{O}(1)} m$ and space $n^{\mathcal{O}(1)}$.*

Proof. We will prove by that the number of important (A, B)-separators of size at most k is at most $2^{2k-\mathsf{flow}(A,B)} \leq 4^k$. The proof is by induction on $2k - \mathsf{flow}(A, B)$, and will naturally give a recursive algorithm for enumerating them in time $2^{2k-\mathsf{flow}(A,B)}$. Note that if $\mathsf{flow}(A, B) > k$, then there are no important (A, B)-separators of size at most k, and if

$\mathsf{flow}(A,B) = 0$, then the empty set is the unique important (A,B)-separator. Therefore, the base case of $2k - \mathsf{flow}(A,B) \leq 0$ holds.

Then assume $k \geq \mathsf{flow}(A,B) \geq 1$, and by Lemma 5.20 let S be the unique important (A,B)-separator of size $|S| = \mathsf{flow}(A,B)$. If $S \subseteq B$, then S is the only important (A,B)-separator. Otherwise, choose $v \in S \setminus B$ arbitrarily. For every important (A,B)-separator S' it holds either that $v \in S'$ or that $v \notin S'$. If S' is of the former type, then $S' \setminus \{v\}$ is an important $(S \setminus \{v\}, B \setminus \{v\})$-separator in $G \setminus \{v\}$. As $\mathsf{flow}_{G\setminus\{v\}}(S \setminus \{v\}, B \setminus \{v\}) = \mathsf{flow}(A,B) - 1$ and $|S' \setminus \{v\}| = |S'| - 1$, the number of such separators S' of size at most k is by induction at most $2^{2(k-1)-(\mathsf{flow}(A,B)-1)} = 2^{2k-\mathsf{flow}(A,B)}/2$. If S' is of the latter type, then by Lemma 5.21 S' is an important $(S \cup N(v), B)$-separator. As $\mathsf{flow}(S \cup N(v), B) \geq \mathsf{flow}(A,B) + 1$, the number of such separators S' of size at most k is by induction at most $2^{2k-(\mathsf{flow}(A,B)+1)} = 2^{2k-\mathsf{flow}(A,B)}/2$. Therefore, the total number of important (A,B)-separators of size at most k is at most $2^{2k-\mathsf{flow}(A,B)}$. □

We will next show that there exists a set of size at most k that intersects every important (A,B)-separator of size at most k, i.e., a *hitting set* for important (A,B)-separators of size at most k, and that such a hitting set can be computed efficiently. To the best of our knowledge this is a novel lemma about important separators, though its proof is only a small variant of the proof of Lemma 5.22.

Lemma 5.23 (Important Separator Hitting Lemma)**.** *There is an algorithm that given two sets $A, B \subseteq V(G)$ and an integer k, in time $k^{\mathcal{O}(1)}m$ outputs a set H of size at most k so that H intersects every non-empty important (A,B)-separator of size at most k.*

Proof. When B is not reachable from A, we can let H be the empty set. When B is reachable from A, we show by induction that there exists such a set H of size at most $\max(0, k - \mathsf{flow}(A,B) + 1)$, which implies the lemma because in this case $\mathsf{flow}(A,B) \geq 1$.

This holds in the base case $k < \mathsf{flow}(A,B)$ because then there exists no (A,B)-separators of size at most k so we can take H as the empty set. Now assume that $k \geq \mathsf{flow}(A,B)$ and that this holds when the difference of k and $\mathsf{flow}(A,B)$ is smaller.

By Lemma 5.20, let S be the unique important (A,B)-separator of size $\mathsf{flow}(A,B)$. If S intersects B, then all important (A,B)-separators intersect $S \cap B$ and we are done by outputting any vertex of $S \cap B$. Otherwise, assume that S does not intersect B and let v be any vertex $v \in S$. By Lemma 5.21, all important (A,B)-separators that do not intersect v are important $(S \cup N(v), B)$-separators. As $\mathsf{flow}(S \cup N(v), B) > \mathsf{flow}(A,B)$, by induction assumption we construct H by taking the union of v and the hitting set for important $(S \cup N(v), B)$-separators of size at most k. □

We will also need the following bound on important separators, which is also proven by a slight variation of the proof of Lemma 5.22. Note that we do not provide an enumeration algorithm in this case, the combinatorial bound is enough for us.

Lemma 5.24. *For any two sets $A, B \subseteq V(G)$, there are at most $k^{k-\mathsf{flow}(A,B)}$ important (A,B)-separators of size at most k.*

Proof. Again, we prove the lemma by induction on $k - \mathsf{flow}(A,B)$. By Lemma 5.20, it holds in the base case $k - \mathsf{flow}(A,B) = 0$, so assume $k - \mathsf{flow}(A,B) \geq 1$ and that the lemma holds for smaller values of $k - \mathsf{flow}(A,B)$.

Let S be the unique important (A,B)-separator of size $|S| = \mathsf{flow}(A,B) < k$. For any important (A,B)-separator S' distinct from S there exists $v \in S \setminus S'$, and by Lemma 5.21 such S' is an important $(S \cup N(v), B)$-separator. As $\mathsf{flow}(S \cup N(v), B) > \mathsf{flow}(A,B)$, we get by induction that the total number of important (A,B)-separators of size at most k is

$$1 + |S| \cdot k^{k-(\mathsf{flow}(A,B)+1)} \leq 1 + (k-1) \cdot k^{k-\mathsf{flow}(A,B)}/k \leq k^{k-\mathsf{flow}(A,B)}.$$

□

5.4 Algorithm for Partitioned Subset Treewidth

This section is devoted to proving Theorem 5.5, in particular to giving a $k^{\mathcal{O}(kt)}n^2$ time algorithm for Partitioned Subset Treewidth. The algorithm will be a branching algorithm that uses important separators.

Throughout this section we assume that the number of edges in the input graph is at most kn, as if this would not hold, we could immediately return that the treewidth is more than k. To make the problem suitable for branching, we now introduce a more general definition of Partitioned Subset Treewidth than given in Section 5.1.

An instance of Partitioned Subset Treewidth is a triple $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$, where G is a graph, t and k are positive integers with $t \leq k+2$, and $\{W_1, \ldots, W_t\}$ is a set of t *terminal cliques*, each $W_i \subseteq V(G)$ being a clique of size at most $k+1$ in G. We denote the union of the terminal cliques by $\hat{W}_{\mathcal{I}} = \bigcup_{i=1}^{t} W_i$. Note that unlike in the definition of Section 5.1, we do not require that $\{W_1, \ldots, W_t\}$ is a partition of $\hat{W}_{\mathcal{I}}$, and neither we require that $|\hat{W}_{\mathcal{I}}| = k+2$. While these properties hold in the initial inputs, in the recursive steps the set $\hat{W}_{\mathcal{I}}$ can become larger, and the terminal cliques can become overlapping. The parameters t and k will not increase in the recursive steps, implying that $|\hat{W}_{\mathcal{I}}| \leq t(k+1)$ always holds.

A solution of an instance of Partitioned Subset Treewidth is a torso tree decomposition $(X, \mathcal{T})$ that covers $\hat{W}_{\mathcal{I}}$ and has width at most k. We say that an instance $\mathcal{I}$ is a *yes-instance* if there exists a solution of it and a *no-instance* otherwise. Our algorithm will either return a solution or conclude that $\mathcal{I}$ is a no-instance, in particular, it will not use the freedom in the definition to return that the treewidth of G is larger than k without concluding that $\mathcal{I}$ is a no-instance (other than for the bound on the number of edges in the initial instance).

5.4.1 Overview

We now give an informal overview of the algorithm, without detailed proofs. All claims that are only sketched here are proven with details in the subsequent subsections. We will first sketch a $k^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithm in the case when there are only two terminal cliques W_1 and W_2. This showcases the most important ideas of our algorithm.

Reduction rule

Let W_1,W_2 be the two terminal cliques, S a minimum size (W_1, W_2)-separator, and (A, S, B) the corresponding separation with $W_1 \subseteq A \cup S$ and $W_2 \subseteq B \cup S$. We will argue that we can make S into a new terminal clique and recursively solve the problem on the graphs $G[A \cup S]$ and $G[B \cup S]$. More formally, we denote by $G \otimes S$ the graph obtained from G by making S a clique, and then denote by $\mathcal{I} \triangleleft (A, S)$ the instance $(G[A \cup S] \otimes S, \{W_1, S\}, k)$ and by $\mathcal{I} \triangleleft (B, S)$ the instance $(G[B \cup S] \otimes S, \{W_2, S\}, k)$. We argue that there exists a solution of $\mathcal{I}$ if and only if there exists solutions of both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$.

Observe that because both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ contain the separator S as a terminal clique but their graphs are disjoint otherwise, any solution of $\mathcal{I} \triangleleft (A, S)$ can be combined with any solution of $\mathcal{I} \triangleleft (B, S)$ into a solution of $\mathcal{I}$ by simply connecting the tree decompositions by an edge between the bags containing S. To argue that if there exists a solution of $\mathcal{I}$ then there exists solutions of both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$, we apply the Pulling Lemma (Lemma 5.8). Because S is a minimum size (W_1, W_2)-separator, S is linked into W_1 and into W_2. Therefore, in order to show that a solution of $\mathcal{I} \triangleleft (A, S)$ exists, we consider a hypothetical solution $(X, (T, \mathsf{bag}))$ of $\mathcal{I}$, and apply the Pulling Lemma (Lemma 5.8) with the separation (A, S, B) and $r \in V(T)$ being a node with $W_2 \subseteq \mathsf{bag}(r)$. In particular, note that S is linked into $\mathsf{bag}(r) \cap (S \cup B) \supseteq W_2$. This constructs a torso tree decomposition $((X \cap A) \cup S, (T', \mathsf{bag}'))$ of width at most k where S is a bag, which can be observed to be a torso tree decomposition also in $G[A \cup S] \otimes S$ because S is a bag

of (T', bag'), and to cover $W_1 \cup S$ because $W_1 \subseteq A \cup S$ and $W_1 \subseteq X$, and therefore is a solution of $\mathcal{I} \triangleleft (A, S)$. The existence of a solution of $\mathcal{I} \triangleleft (B, S)$ is proven similarly.

Observe that this reduction rule makes progress as long as $S \neq W_1$ and $S \neq W_2$, and thus we apply the rule as long as there exists such minimum size (W_1, W_2)-separator S. Recall that we say that W_1 is strongly linked into W_2 if W_1 is linked into W_2 and the only minimum size (W_1, W_2)-separators are W_1, and W_2 in the case when $|W_1| = |W_2|$.

Leaf pushing

Assume now that we cannot make any more progress by the reduction rule, and let $|W_1| \leq |W_2|$, implying that W_1 is strongly linked into W_2. Our goal is now to make progress by increasing the size of W_1. We observe that for any solution $(X, (T, \mathsf{bag}))$ that minimizes $|X|$, it holds that if ℓ is a leaf node of T and p is the parent of ℓ, then $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p) \subseteq W_1 \cup W_2$. Furthermore, we can assume that $\mathsf{bag}(p) = \mathsf{bag}(\ell) \setminus \{w\}$, where w is a forget-vertex of ℓ, and therefore $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p) \subseteq W_1$ or $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p) \subseteq W_2$. Then, observe that if $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$ intersects W_i, it must hold that $W_i \subseteq \mathsf{bag}(\ell)$ because W_i is a clique. Therefore, (T, bag) either contains a bag that contains both W_1 and W_2, in which case $|W_1 \cup W_2| \leq k + 1$ and there is a trivial single-bag solution, or (T, bag) has exactly two leaves and for one of them it holds that $W_1 \subseteq \mathsf{bag}(\ell)$ and $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p) \subseteq W_1 \setminus W_2$.

Now, our goal will be, informally, to increase the size of W_1 by guessing a vertex in $\mathsf{bag}(\ell) \setminus W_1$ and adding it to W_1. We let w be the forget-vertex of ℓ, and observe that the parent bag $\mathsf{bag}(p) = \mathsf{bag}(\ell) \setminus \{w\}$ is a (W_1, W_2)-separator. This shows that $\mathsf{bag}(\ell) \setminus W_1$ must be non-empty, because otherwise $\mathsf{bag}(p)$ would be a (W_1, W_2)-separator of size $|W_1| - 1$, contradicting that W_1 is linked into W_2. Denote $G' = G \setminus (W_1 \setminus \{w\})$, and

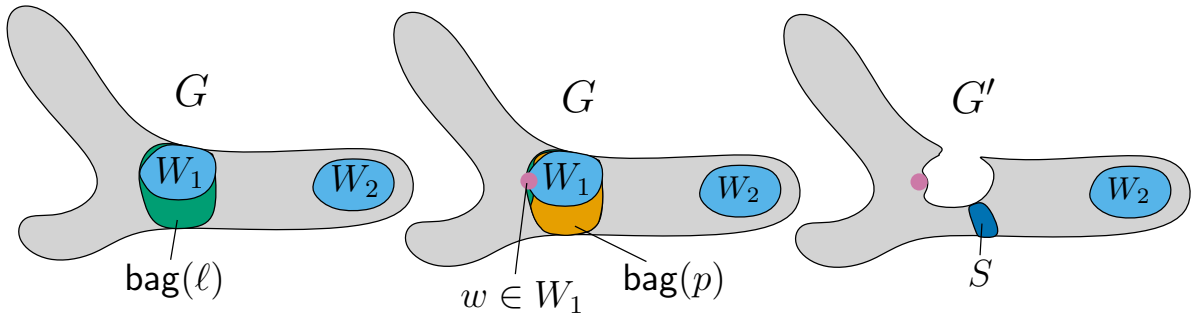

Figure 5.2: Illustration of the leaf pushing argument. The picture on the left depicts the graph G along with the terminal cliques W_1 and W_2, and the leaf bag $\mathsf{bag}(\ell) \supseteq W_1$. The picture in the middle depicts the parent bag $\mathsf{bag}(p) = \mathsf{bag}(\ell) \setminus \{w\}$ and the forget-vertex $w \in W_1$. The picture on the right depicts the graph $G' = G \setminus (W_1 \setminus \{w\})$ and an important $(\{w\}, W_2 \setminus W_1)$-separator S.

observe that in the graph G' the set $\mathsf{bag}(\ell) \setminus W_1 = \mathsf{bag}(p) \setminus W_1$ is a $(\{w\}, W_2 \setminus W_1)$-separator. We will then show that the subset $\mathsf{bag}(\ell) \setminus W_1$ of $\mathsf{bag}(\ell)$ can be replaced by an important $(\{w\}, W_2 \setminus W_1)$-separator. In particular, we will argue that there is an important $(\{w\}, W_2 \setminus W_1)$-separator $S \neq \{w\}$ in the graph G' so that there exists a solution containing a bag $W_1 \cup S$. See Figure 5.2 for an illustration of these objects.

Let S be an important $(\{w\}, W_2 \setminus W_1)$-separator in the graph G' so that it dominates $\mathsf{bag}(\ell) \setminus W_1$ and minimizes $|S|$ among all such important separators. Denote the separation corresponding to S by $(A, S, B) = (\mathsf{reach}_{G'}(\{w\}, S), S, V(G') \setminus (S \cup \mathsf{reach}_{G'}(\{w\}, S)))$. By Lemma 5.18, S is linked into $(A \cup S) \cap (\mathsf{bag}(\ell) \setminus W_1)$. Then, by adding $W_1 \setminus \{w\}$ back to the graph and to the separation, we get that $(A, S \cup W_1 \setminus \{w\}, B)$ is a separation of G and $S \cup W_1 \setminus \{w\}$ is linked into $(A \cup S \cup W_1 \setminus \{w\}) \cap \mathsf{bag}(\ell)$ (the vertices in $W_1 \setminus \{w\}$ are linked by trivial one-vertex paths). We then apply the Pulling Lemma (Lemma 5.8) with the hypothetical solution $(X, (T, \mathsf{bag}))$, the separation $(B, S \cup W_1 \setminus \{w\}, A)$, and the node ℓ, to argue that there exists a torso tree decomposition $((X \cap B) \cup S \cup W_1 \setminus \{w\}, (T', \mathsf{bag}'))$ of width at most k, containing a bag $S \cup W_1 \setminus \{w\}$. As $|S| \leq |\mathsf{bag}(\ell) \setminus W_1|$, this can be turned into a solution of $\mathcal{I}$ by inserting w into the bag $S \cup W_1 \setminus \{w\}$. Therefore there exists a solution of $\mathcal{I}$ with a bag $W_1 \cup S$, and in particular it is safe to replace the terminal clique W_1 by $W_1 \cup S$, also replacing G by $G \otimes (W_1 \cup S)$.

Now, we are able to increase the size of W_1 by guessing the forget-vertex $w \in W_1$ and an important separator S and branching to $(G \otimes (W_1 \cup S), \{W_1 \cup S, W_2\}, k)$. However, by applying the reduction rule we might immediately lose most of the progress by finding a $(W_1 \cup S, W_2)$-separator S' of size $|S'| < |W_1 \cup S|$ and ending up with an instance with terminal cliques $\{S', W_2\}$. Nevertheless, we can ensure that such S' must have size $|S'| > |W_1|$ by using the facts that W_1 is strongly linked into W_2 and the way S was selected. In particular, in the end, after applying the reduction rule possibly several times, we can guarantee that if initially $|W_1| = |W_2|$, then each resulting instance has terminal cliques of sizes at least $|W_1| + 1$ and $|W_2|$, and if initially $|W_1| < |W_2|$, then each resulting instance has terminal cliques of sizes at least $|W_1| + 1$ and $|W_1| + 1$. It follows that the depth of the resulting branching tree is at most $2k$, as the sizes of the terminal cliques are bounded by $k + 1$.

Then, as the number of important separators of size at most k is at most 4^k (Lemma 5.22), this results in a branching tree of degree $k4^k$ and depth $2k$, resulting in a $(k4^k)^{2k} n^{\mathcal{O}(1)} = 2^{\mathcal{O}(k^2)} n^{\mathcal{O}(1)}$ time algorithm. To improve this to $k^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ time, we observe that in order to make progress, it is sufficient to guess only one vertex of the important separator S and add it to W_1, instead of guessing the whole important separator S. To this end, we use the Important Separator Hitting Lemma (Lemma 5.23) that gives a set of size k that intersects all important separators of size at most k, and therefore allows to guess one

vertex in an important separator of size at most k by a branching degree of k instead of 4^k, resulting in a $(k^2)^{2k}n^{\mathcal{O}(1)} = k^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithm.

More than two terminal cliques

Generalizing the just sketched $k^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithm for two terminal cliques into the $k^{\mathcal{O}(kt)}n^{\mathcal{O}(1)}$ algorithm for t terminal cliques of Theorem 5.5 does not require major new ideas, but requires several technical considerations. In the algorithm for t terminal cliques, we will in addition to the leaf pushing branching do branching on merging two different terminal cliques into one, which should be done whenever we guess that there exists a solution where the two terminal cliques are in a same bag. The "real" definition of the measure of the instance will also be more involved, in particular, instead of depending on the sizes of terminal cliques, the measure depends on a notion of "flow potential" of a terminal clique. The flow potential has a technical definition, but for all terminal cliques W_i except for a uniquely largest one it will be equal to the flow from W_i into the union of the other terminal cliques. The measure of a uniquely largest terminal clique must be special to encode that we make progress, for example, in the case when there are two terminal cliques W_1 and W_2 with $|W_1| = |W_2|$ and after branching we end up with two terminal cliques of sizes $|W_1| + 1$ and $|W_2|$. The measure will also take into account the number of terminal cliques, in particular, it will "encode" that decreasing the number of terminal cliques with the expense of making the flow potential of one terminal clique worse still means making overall progress.

The rest of this section is organized as follows. In Subsection 5.4.2 we introduce the measure flow potential for quantifying the progress of the algorithm. In Subsection 5.4.3 we give a reduction rule to simplify instances by safe separations. In Subsection 5.4.4 we give the two branching rules of the algorithm, and in Subsection 5.4.5 we describe the algorithm and put together its correctness proof. Finally, in Subsection 5.4.6 we analyze the running time.

5.4.2 Flow potential

We define the measure *flow potential* that will be used for quantifying the progress of the algorithm. We warn the reader that this definition can be unintuitive, but we have not managed to come up with a simpler alternative.

Let G be a graph and $X, Y \subseteq V(G)$ two sets of vertices. If $X \subset V(G)$, then the flow potential from X to Y, denoted by $\mathsf{flow}\text{-}\Phi_G(X, Y)$ is the minimum order of a separation

(A, S, B), where $X \subseteq A \cup S$, $Y \subseteq B \cup S$, and $B \neq \emptyset$. If $X = V(G)$, then we define $\mathsf{flow}\text{-}\Phi_G(X, Y) = |X|$. We observe that

$$\mathsf{flow}_G(X, Y) \leq \mathsf{flow}\text{-}\Phi_G(X, Y) \leq |X|.$$

Note that $\mathsf{flow}_G(X, Y) < \mathsf{flow}\text{-}\Phi_G(X, Y)$ if and only if $|X| > |Y|$, Y is strongly linked into X, and X intersects all connected components of $G \setminus Y$. We will also use the property that if $Y_1 \subseteq Y_2$, then $\mathsf{flow}\text{-}\Phi_G(X, Y_1) \leq \mathsf{flow}\text{-}\Phi_G(X, Y_2)$.

Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance. For a terminal clique W_i, we denote by $\overline{W}_{\mathcal{I}}(W_i) = \bigcup_{j \in [t] \setminus \{i\}} W_j$ the union of the other terminal cliques in $\mathcal{I}$. We define the flow potential of W_i in $\mathcal{I}$ to be

$$\mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i) = \mathsf{flow}\text{-}\Phi_G(W_i, \overline{W}_{\mathcal{I}}(W_i)).$$

Note that $\mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i) \neq \mathsf{flow}_G(W_i, \overline{W}_{\mathcal{I}}(W_i))$ can hold only if W_i is the unique largest terminal clique of $\mathcal{I}$.

5.4.3 Safe separations

We say that a separation (A, S, B) is a *strict separation* if both A and B are non-empty. In this subsection we introduce a reduction rule based on identifying a strict separation (A, S, B), making S a clique and enforcing S to be covered, and then solving the different sides of S independently of each other. In particular, we show that this reduction is safe if (A, S, B) satisfies certain conditions that we define next.

Definition 5.25 (Safe separation)**.** *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a strict separation of G. We say that (A, S, B) is a safe separation of $\mathcal{I}$ if there exists terminal cliques W_a and W_b (possibly $a = b$) so that S is linked into $(A \cup S) \cap W_a$ and into $(B \cup S) \cap W_b$.*

We say that such terminal cliques W_a and W_b are the *spanning terminal cliques* of the safe separation. Note that safe separations can be classified into two types: those where S is a subset of some terminal clique W_i and we can take $W_a = W_b = W_i$, and those where $a \neq b$ and S is a minimum size (W_a, W_b)-separator. The purpose of our definition is to be a common generalization of these two types.

Next we introduce notation for reduction by safe separations. In Section 5.5 this notation will also be used with separations that are not necessarily safe.

Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a separation with $|S| \leq k+1$. We denote by $\{W_1, \ldots, W_t\} \triangleleft (A, S)$ the set obtained from $\{W_1, \ldots, W_t\}$ by first removing each W_i with $W_i \cap A = \emptyset$, and then inserting S if no superset of S is present. Recall that $G \otimes S$ denotes the graph obtained from G by making S a clique. We define $G \triangleleft (A, S) = G[A \cup S] \otimes S$, and then $\mathcal{I} \triangleleft (A, S) = (G \triangleleft (A, S), \{W_1, \ldots, W_t\} \triangleleft (A, S), k)$.

Note that if (A, S, B) is a safe separation, then $|S| \leq k+1$ because S is linked into a spanning terminal clique W_a and $|W_a| \leq k+1$. Also, observe that if (A, S, B) is a safe separation, then $\mathcal{I} \triangleleft (A, S)$ has at most t terminal cliques. This is because if all terminal cliques of $\mathcal{I}$ intersect A, then (A, S, B) can be a safe separation only if the spanning terminal clique W_b is a superset of S.

The reduction rule used in the algorithm will be that if there exists a safe separation (A, S, B), then solve the instances $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ independently of each other. If either of them returns NO, then return NO, and if both of them return a solution, denoted by $(X_A, \mathcal{T}_A)$ and $(X_B, \mathcal{T}_B)$, respectively, then return the solution obtained by combining these solutions on the separator S. More formally, if $\mathcal{T}_A$ and $\mathcal{T}_B$ are tree decompositions that both contain a bag containing S, then we denote by $\mathcal{T}_A \cup_S \mathcal{T}_B$ the tree decomposition obtained by taking the disjoint union of $\mathcal{T}_A$ and $\mathcal{T}_B$ and connecting them by an edge between bags containing S. Then, the combined solution is denoted by $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$.

In the next two lemmas we show that this reduction is correct. We start by proving that if both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ return a solution, then the constructed solution is a solution of $\mathcal{I}$. This holds in fact for any separation (A, S, B).

Lemma 5.26. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a separation. If $(X_A, \mathcal{T}_A)$ is a solution of $\mathcal{I} \triangleleft (A, S)$ and $(X_B, \mathcal{T}_B)$ a solution of $\mathcal{I} \triangleleft (B, S)$, then $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ is a solution of $\mathcal{I}$.*

Proof. First, note that both $\{W_1, \ldots, W_t\} \triangleleft (A, S)$ and $\{W_1, \ldots, W_t\} \triangleleft (B, S)$ contain a superset of S, so both X_A and X_B are supersets S. Also, any terminal clique in $\{W_1, \ldots, W_t\}$ is either in $\{W_1, \ldots, W_t\} \triangleleft (A, S)$, in $\{W_1, \ldots, W_t\} \triangleleft (B, S)$, or is a subset of S, so $X_A \cup X_B$ is a superset of $\hat{W}_{\mathcal{I}}$.

Because S is a clique in both $G \triangleleft (A, S)$ and in $G \triangleleft (B, S)$, it is contained in a bag of both $\mathcal{T}_A$ and $\mathcal{T}_B$, and therefore the construction $\mathcal{T}_A \cup_S \mathcal{T}_B$ is well-defined. Because $G \triangleleft (A, S)$ and $G \triangleleft (B, S)$ intersect only in S, it holds that $X_A \cap X_B = S$, which implies that $\mathcal{T}_A \cup_S \mathcal{T}_B$ satisfies the connectedness condition to be a tree decomposition of $\mathsf{torso}_G(X_A \cup X_B)$. The vertex condition is trivially satisfied. For the edge condition, consider an edge $uv \in E(\mathsf{torso}_G(X_A \cup X_B))$. Because (A, S, B) is a separation and $S \subseteq X_A \cup X_B$, it

must hold that $u, v \in A \cup S$ or $u, v \in B \cup S$. If $u, v \in S$, then there clearly is a bag containing them, so by assume without loss of generality that $u \in A$ and $v \in A \cup S$. The internal vertices of the path between u and v must be in $A \setminus X_A$, and therefore $uv \in E(\mathsf{torso}_{G \triangleleft (A,S)}(X_A))$, implying that uv is in some bag of $\mathcal{T}_A$. Therefore $\mathcal{T}_A \cup_S \mathcal{T}_B$ satisfies the edge condition. □

We then show the other direction of correctness by applying the Pulling Lemma (Lemma 5.8).

Lemma 5.27. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a safe separation. If $\mathcal{I}$ is a yes-instance, then both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are yes-instances.*

Proof. By the symmetry of the definition of safe separation, it suffices to show that $\mathcal{I} \triangleleft (A, S)$ is a yes-instance.

Let W_a and W_b be the spanning terminal cliques of (A, S, B) and $(X, (T, \mathsf{bag}))$ a solution of $\mathcal{I}$. Because W_b is a clique in G and is contained in X, there exists a node $r \in V(T)$ with $W_b \subseteq \mathsf{bag}(r)$, which implies that S is linked into $\mathsf{bag}(r) \cap (B \cup S)$. We use the Pulling Lemma (Lemma 5.8) with the separation (A, S, B) and the node r to obtain a torso tree decomposition $((X \cap A) \cup S, (T', \mathsf{bag}'))$ of width at most k containing a bag containing S. We observe that $((X \cap A) \cup S, (T', \mathsf{bag}'))$ is a torso tree decomposition also in $G[A \cup S]$, and because it covers S and (T', bag') contains a bag containing S, it is also a torso tree decomposition in $G \triangleleft (A, S)$. Because every terminal clique in $\{W_1, \ldots, W_t\} \triangleleft (A, S)$ is a subset of $(X \cap A) \cup S$, we have that $((X \cap A) \cup S, (T', \mathsf{bag}'))$ covers $\hat{W}_{\mathcal{I} \triangleleft (A,S)}$. □

Next we will give three lemmas arguing that the flow potentials "behave well" when breaking the instance by safe separations. In particular, properties that naturally hold for flow also hold for flow potential. We start by considering a situation where the breaking does not change the terminal cliques.

Lemma 5.28. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a separation. If $\{W_1, \ldots, W_t\} \triangleleft (A, S) = \{W_1, \ldots, W_t\}$, then $\mathsf{flow}\text{-}\Phi_{\mathcal{I} \triangleleft (A,S)}(W_i) \geq \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i)$ for all terminal cliques W_i.*

Proof. Observe that in this case, $\overline{W}_{\mathcal{I} \triangleleft (A,S)}(W_i) = \overline{W}_{\mathcal{I}}(W_i)$, and therefore it suffices to prove that $\mathsf{flow}\text{-}\Phi_G(W_i, \overline{W}_{\mathcal{I}}(W_i)) \leq \mathsf{flow}\text{-}\Phi_{G \triangleleft (A,S)}(W_i, \overline{W}_{\mathcal{I}}(W_i))$.

We argue by the definition of flow potential. First, if $W_i = A \cup S$, this holds trivially, so assume that $W_i \subset A \cup S$. Let (A', S', B') be a separation in $G \triangleleft (A, S)$ of minimum order so that $W_i \subseteq A' \cup S'$, $\overline{W}_{\mathcal{I}}(W_i) \subseteq B' \cup S'$, and B' is non-empty. Now, S is a clique

in $G \triangleleft (A, S)$, so either $S \subseteq A' \cup S'$ or $S \subseteq B' \cup S'$. If $S \subseteq A' \cup S'$ then $(A' \cup B, S', B')$ is a separation in G, and if $S \subseteq B' \cup S'$ then $(A', S', B' \cup B)$ is a separation in G. In either case, $\mathsf{flow\text{-}}\Phi_G(W_i, \overline{W}_{\mathcal{I}}(W_i)) \leq |S'| = \mathsf{flow\text{-}}\Phi_{G \triangleleft (A,S)}(W_i, \overline{W}_{\mathcal{I}}(W_i))$. □

We then argue that for most of the terminal cliques the flow potential does not decrease when going from $\mathcal{I}$ to $\mathcal{I} \triangleleft (A, S)$.

Lemma 5.29. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance, (A, S, B) a separation, and W_i a terminal clique of both $\mathcal{I}$ and $\mathcal{I} \triangleleft (A, S)$ so that there exists some other terminal clique $W_j \neq W_i$ of $\mathcal{I} \triangleleft (A, S)$ that is a superset of S. Then, $\mathsf{flow\text{-}}\Phi_{\mathcal{I} \triangleleft (A,S)}(W_i) \geq \mathsf{flow\text{-}}\Phi_{\mathcal{I}}(W_i)$.*

Proof. We observe that because $W_j \supseteq S$, it holds that $\overline{W}_{\mathcal{I} \triangleleft (A,S)}(W_i) = (\overline{W}_{\mathcal{I}}(W_i) \cap A) \cup S$. Therefore, it suffices to prove that

$$\mathsf{flow\text{-}}\Phi_G(W_i, \overline{W}_{\mathcal{I}}(W_i)) \leq \mathsf{flow\text{-}}\Phi_{G \triangleleft (A,S)}(W_i, (\overline{W}_{\mathcal{I}}(W_i) \cap A) \cup S).$$

We argue by the definition of flow potential. First, if $W_i = A \cup S$, the lemma holds trivially, so assume that $W_i \subset A \cup S$. Let (A', S', B') be a separation in $G \triangleleft (A, S)$ of minimum order so that $W_i \subseteq A' \cup S'$, $(\overline{W}_{\mathcal{I}}(W_i) \cap A) \cup S \subseteq B' \cup S'$, and B' is non-empty. Now, because $S \subseteq B' \cup S'$, we have that $(A', S', B' \cup B)$ is a separation in G. Note that $\overline{W}_{\mathcal{I}}(W_i) \subseteq S' \cup B' \cup B$, so it follows that

$$\mathsf{flow\text{-}}\Phi_G(W_i, \overline{W}_{\mathcal{I}}(W_i)) \leq |S'| = \mathsf{flow\text{-}}\Phi_{G \triangleleft (A,S)}(W_i, (\overline{W}_{\mathcal{I}}(W_i) \cap A) \cup S).$$

□

Then, we reduce the task of analyzing the flow potentials of $\mathcal{I} \triangleleft (A, S)$ into three cases.

Lemma 5.30. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a safe separation. At least one of the following holds:*

1. *$\mathcal{I} \triangleleft (A, S)$ has less terminal cliques than $\mathcal{I}$,*
2. *$\{W_1, \ldots, W_t\} \triangleleft (A, S) = \{W_1, \ldots, W_t\}$, or*
3. *S is a terminal clique of $\mathcal{I} \triangleleft (A, S)$ but not of $\mathcal{I}$, and there exists a terminal clique W_i of $\mathcal{I}$ with $W_i \cap B \neq \emptyset$ and $\mathsf{flow\text{-}}\Phi_{\mathcal{I} \triangleleft (A,S)}(S) \geq \mathsf{flow\text{-}}\Phi_{\mathcal{I}}(W_i)$.*

Proof. Let W_a and W_b be the spanning terminal cliques of (A, S, B). If $W_b \subseteq A \cup S$, then W_b is also a superset of S because S is linked into $W_b \cap (S \cup B)$, and either Item 1 or

Item 2 holds. Then, if W_b intersects B and also some other terminal clique intersects B, Item 1 holds. It remains to prove that if W_b is the only terminal clique that intersects B and $\mathcal{I} \triangleleft (A, S)$ has the same number of terminal cliques as $\mathcal{I}$, then Item 3 holds.

In this case, S is a terminal clique of $\mathcal{I} \triangleleft (A, S)$ but not of $\mathcal{I}$, and $\overline{W}_{\mathcal{I} \triangleleft (A,S)}(S) = \overline{W}_{\mathcal{I}}(W_b)$, implying that it suffices to prove that $\mathsf{flow\text{-}}\Phi_G(W_b, \overline{W}_{\mathcal{I}}(W_b)) \leq \mathsf{flow\text{-}}\Phi_{G \triangleleft (A,S)}(S, \overline{W}_{\mathcal{I}}(W_b))$. We argue by the definition of flow potential. Because (A, S, B) is a strict separation, it holds that $S \subset A \cup S$. Let (A', S', B') be a separation in $G \triangleleft (A, S)$ of minimum order so that $S \subseteq A' \cup S'$, $\overline{W}_{\mathcal{I}}(W_b) \subseteq B' \cup S'$, and B' is non-empty. Now, because $S \subseteq A' \cup S'$, we have that $(A' \cup B, S', B')$ is a separation in G. As $W_b \subseteq A' \cup B \cup S'$, it follows that $\mathsf{flow\text{-}}\Phi_G(W_b, \overline{W}_{\mathcal{I}}(W_b)) \leq |S'| = \mathsf{flow\text{-}}\Phi_{G \triangleleft (A,S)}(S, \overline{W}_{\mathcal{I}}(W_b))$. □

We then show that safe separations can be found efficiently.

Lemma 5.31. *There is a $k^{\mathcal{O}(1)}m$ time algorithm for finding a safe separation or deciding that none exist.*

Proof. We try all pairs of terminal cliques W_a and W_b and find safe separations whose spanning terminal cliques W_a and W_b are.

Let (A, S, B) be a safe separation and W_a and W_b its spanning terminal cliques. First, consider safe separators where (A, S, B) where $S \subseteq W_a$ or $S \subseteq W_b$. Assume without loss of generality that $S \subseteq W_a$. We can find such safe separations by checking if $G \setminus W_a$ has at least two connected components, and also trying all $w \in W_a$ and checking if $G \setminus (W_a \setminus \{w\})$ has at least two connected components.

Then, consider safe separations (A, S, B) spanned by W_a and W_b so that S is not a subset of W_a or W_b. In this case W_a intersects A and W_b intersects B, and S is a (W_a, W_b)-separator. By the symmetry of the definition we may assume that $|W_a| \leq |W_b|$. If W_a is strongly linked into W_b, then no such safe separators exists. Then, if W_a is not strongly linked into W_b, any minimum size (W_a, W_b)-separator S with $S \neq W_a$ and $S \neq W_b$ corresponds to a safe separator, and can be found by standard flow computations in $k^{\mathcal{O}(1)}m$ time. □

Then, we state the fact that applying safe separations makes all pairs of terminal cliques strongly linked into each other.

Lemma 5.32. *If $\mathcal{I}$ has no safe separations, then for each pair of terminal cliques W_i, W_j with $|W_i| \leq |W_j|$, it holds that W_i is strongly linked into W_j.*

Proof. If W_i would not be strongly linked into W_j, then the separator contradicting strong linkedness would give a safe separation. □

5.4.4 Branching

We do two types of branching in our algorithm, terminal clique merging and leaf pushing.

Terminal clique merging

The first type of branching is that we guess that two terminal cliques W_i and W_j are in a same bag in some solution, and therefore we can actually merge them into one terminal clique. We introduce some notation for this operation and analyze the flow potential under it.

Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance. For two distinct terminal cliques W_i and W_j with $|W_i \cup W_j| \leq k+1$, we denote by $\{W_1, \ldots, W_t\} \times (W_i, W_j)$ the set obtained by removing W_i and W_j from $\{W_1, \ldots, W_t\}$ and inserting $W_i \cup W_j$ (if $W_i \cup W_j$ is already present, nothing is inserted). We make $W_i \cup W_j$ into a clique in this operation so we denote $G \times (W_i, W_j) = G \otimes (W_i \cup W_j)$ and by $\mathcal{I} \times (W_i, W_j)$ we denote the instance $(G \times (W_i, W_j), \{W_1, \ldots, W_t\} \times (W_i, W_j), k)$.

Next we observe that the terminal clique merging does not decrease the flow potentials of the other terminal cliques.

Lemma 5.33. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be instance and $W_i, W_j, W_\ell \in \{W_1, \ldots, W_t\}$ three distinct terminal cliques. It holds that* $\mathsf{flow}\text{-}\Phi_{\mathcal{I} \times (W_i, W_j)}(W_\ell) \geq \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_\ell)$.

Proof. This follows directly from the facts that $\overline{W}_{\mathcal{I}}(W_\ell) = \overline{W}_{\mathcal{I} \times (W_i, W_j)}(W_\ell)$ and any separation of $G \times (W_i, W_j)$ is also a separation of G. □

We also observe that any solution of $\mathcal{I} \times (W_i, W_j)$ is also a solution of $\mathcal{I}$.

We say that $\mathcal{I}$ is *maximally merged* if for any pair of two distinct terminal cliques W_i and W_j it holds that either $|W_i \cup W_j| > k+1$ or $\mathcal{I} \times (W_i, W_j)$ is a no-instance. In particular, we can conclude that $\mathcal{I}$ is maximally merged after branching on all different ways to merge two terminal cliques and not finding a solution.

Leaf pushing

A terminal clique W_i is a *potential forget-clique* of $\mathcal{I}$ if there exists a solution $(X, (T, \mathsf{bag}))$ of $\mathcal{I}$ so that T contains a leaf node ℓ with a parent p so that $W_i \subseteq \mathsf{bag}(\ell)$ and $\mathsf{bag}(p) = \mathsf{bag}(\ell) \setminus \{w\}$ for some $w \in W_i \setminus \overline{W}_{\mathcal{I}}(W_i)$. The leaf pushing operation will make progress by adding a vertex to a potential forget-clique.

Next we show that a maximally merged yes-instance has at least two potential forget-cliques. The fact that there are at least two of them will be important since we do not necessarily make progress by leaf pushing a uniquely largest terminal clique.

Lemma 5.34. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be a yes-instance that is maximally merged and has $t \geq 2$. The instance $\mathcal{I}$ has at least two potential forget-cliques.*

Proof. Let $(X, (T, \mathsf{bag}))$ be a solution of $\mathcal{I}$ that first minimizes $|X|$ and subject to that minimizes $|V(T)|$. First, if $|V(T)| = 1$, then all terminal cliques would be contained in the only bag of (T, bag), and $\mathcal{I}$ would not be maximally merged. Therefore $|V(T)| \geq 2$ and T has at least two leaves.

Claim 5.35. *For any leaf node ℓ of T with parent p, it holds that $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p) \subseteq \hat{W}_{\mathcal{I}}$.*

Proof of the claim. Suppose that $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$ contains a vertex $x \in V(G) \setminus \hat{W}_{\mathcal{I}}$. Let (T', bag') be the tree decomposition obtained from (T, bag) by removing x from $\mathsf{bag}(\ell)$. We claim that then $(X \setminus \{x\}, (T', \mathsf{bag}'))$ is a solution of $\mathcal{I}$ that would contradict the minimality of $|X|$. It holds that $X \setminus \{x\}$ covers $\hat{W}_{\mathcal{I}}$ and that the width of (T', bag') is at most k, so it remains to argue that (T', bag') is a tree decomposition of $\mathsf{torso}_G(X \setminus \{x\})$. Because x occurred only in the bag $\mathsf{bag}(\ell)$, (T', bag') satisfies the vertex condition and the connectedness condition. For the edge condition, it suffices to prove that any path from x to $X \setminus \{x\}$ intersects $\mathsf{bag}(\ell) \setminus \{x\}$. This follows from Lemma 5.7 by considering a modified version of (T, bag) where a bag containing $\mathsf{bag}(\ell) \setminus \{x\}$ is inserted between ℓ and p. ◁

Now, let $\ell, p \in V(T)$ be some leaf-parent pair. We have that $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$ is non-empty because otherwise we could decrease $|V(T)|$ by removing ℓ. Therefore, by Claim 5.35 there exists a terminal clique W_i that intersects $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$. Because W_i is a clique and the decomposition covers W_i, we know that $W_i \subseteq \mathsf{bag}(\ell)$. We can modify (T, bag) by adding nodes between ℓ and p so that the vertices in $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$ are forgotten one-by-one, and that a vertex $w \in W_i \cap (\mathsf{bag}(\ell) \setminus \mathsf{bag}(p))$ is the first to be forgotten. In particular, this results in a decomposition where the parent of ℓ is a node p' with $\mathsf{bag}(p') = \mathsf{bag}(\ell) \setminus \{w\}$. Now, if w would be in some other terminal clique $W_j \neq W_i$, then

$W_j \subseteq \mathsf{bag}(\ell)$ would hold because $\mathsf{bag}(\ell)$ is the only bag containing w, but then $\mathcal{I}$ would not be maximally merged. Therefore, W_i is a potential forget-clique.

Finally, to show that there are at least two potential forget-cliques, note that if W_i intersects $\mathsf{bag}(\ell) \setminus \mathsf{bag}(p)$ for two different leaf-parent pairs ℓ_1, p_1 and ℓ_2, p_2, then because $W_i \subseteq \mathsf{bag}(\ell_1)$ and $W_i \subseteq \mathsf{bag}(\ell_2)$, by the connectedness condition it would hold that $W_i \subseteq \mathsf{bag}(p_1)$, contradicting that W_i intersects $\mathsf{bag}(\ell_1) \setminus \mathsf{bag}(p_1)$. Therefore, for every leaf-parent pair ℓ, p we can assign a unique terminal clique W_i, and therefore as there are at least two leaves there are at least two potential forget-cliques. □

We introduce notation for the leaf pushing operation. Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance, W_i a terminal clique, and $A \subseteq V(G)$ a set of vertices that is disjoint from W_i and $|W_i \cup A| \leq k+1$. We denote by $\{W_1, \ldots, W_t\} + (W_i, A)$ the set obtained from $\{W_1, \ldots, W_t\}$ by replacing the terminal clique W_i by $W_i \cup A$. Again, if $W_i \cup A$ already exists, we just remove W_i. We denote $G + (W_i, A) = G \otimes (W_i \cup A)$ and by $\mathcal{I} + (W_i, A)$ we denote the instance $(G + (W_i, A), \{W_1, \ldots, W_t\} + (W_i, A), k)$. Observe that any solution of $\mathcal{I} + (W_i, A)$ is also a solution of $\mathcal{I}$. Observe also that if $\mathcal{I} + (W_i, A)$ is a yes-instance and $A' \subseteq A$, then $\mathcal{I} + (W_i, A')$ is also a yes-instance.

Next we prove the main leaf pushing lemma, in particular that we can increase the size of a potential forget-clique by guessing an important separator.

Lemma 5.36. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be a maximally merged yes-instance with $t \geq 2$ and no safe separators and W_i a potential forget-clique of $\mathcal{I}$. There exists a vertex $w \in W_i \setminus \overline{W}_{\mathcal{I}}(W_i)$ and in the graph $G \setminus (W_i \setminus \{w\})$ a non-empty important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator S disjoint from W_i so that $\mathcal{I} + (W_i, S)$ is a yes-instance.*

Proof. By the definition of potential forget-clique, let $(X, (T, \mathsf{bag}))$ be a solution so that (T, bag) contains a leaf node ℓ with a parent p so that $W_i \subseteq \mathsf{bag}(\ell)$ and $\mathsf{bag}(p) = \mathsf{bag}(\ell) \setminus \{w\}$ for $w \in W_i \setminus \overline{W}_{\mathcal{I}}(W_i)$. Denote $W_i^f = W_i \setminus \{w\}$.

By Lemma 5.7 it holds that $\mathsf{bag}(p) = \mathsf{bag}(\ell) \setminus \{w\}$ separates w from $X \setminus \{w\}$, and therefore separates w from $\hat{W}_{\mathcal{I}} \setminus \{w\}$. Therefore, in the graph $G \setminus W_i^f$, the set $\mathsf{bag}(\ell) \setminus W_i$ is a $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator. The set $\hat{W}_{\mathcal{I}} \setminus W_i$ is non-empty because $\mathcal{I}$ is maximally merged and $t \geq 2$.

Let S be a smallest important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator in $G \setminus W_i^f$ that dominates $\mathsf{bag}(\ell) \setminus W_i$. The separator S does not contain w because $w \in \mathsf{reach}_{G \setminus W_i^f}(\{w\}, \mathsf{bag}(\ell) \setminus W_i)$, and therefore S is disjoint from W_i. The separator S is non-empty because otherwise W_i^f would separate w from $\hat{W}_{\mathcal{I}} \setminus W_i$ and be a safe separator. Let (A, S, B) be the separation

in $G \setminus W_i^f$ with $B = \mathsf{reach}_{G \setminus W_i^f}(\{w\}, S)$ and $A = V(G) \setminus (W_i^f \cup B \cup S)$, which implies $\hat{W}_{\mathcal{I}} \setminus W_i \subseteq A \cup S$. By Lemma 5.18, S is linked into $(\mathsf{bag}(\ell) \setminus W_i) \cap (B \cup S)$ in $G \setminus W_i^f$.

By adding W_i^f back to the graph and to the separator, we get that $(A, S \cup W_i^f, B)$ is a separation of G, and moreover $S \cup W_i^f$ is linked into $\mathsf{bag}(\ell) \cap (B \cup S \cup W_i^f)$ (the vertices in W_i^f are linked by trivial one-vertex paths). Let $X' = (X \cap A) \cup S \cup W_i^f$ and $(X', (T', \mathsf{bag}'))$ be the torso tree decomposition obtained by applying the Pulling Lemma (Lemma 5.8) with $(X, (T, \mathsf{bag}))$, the separation $(A, S \cup W_i^f, B)$, and the node ℓ of T.

Now, $(X', (T', \mathsf{bag}'))$ has width at most k and it covers $\hat{W}_{\mathcal{I}} \setminus \{w\}$. Let t be the node of T' so that $S \cup W_i^f \subseteq \mathsf{bag}'(t)$. We construct a torso tree decomposition $(X' \cup \{w\}, (T'', \mathsf{bag}''))$ by adding a leaf t' adjacent to t with $\mathsf{bag}''(t') = S \cup W_i$. Because $|S| \leq |\mathsf{bag}(\ell) \setminus W_i|$, it follows that $|\mathsf{bag}''(t')| \leq |\mathsf{bag}(\ell)| \leq k+1$. Also, $(X' \cup \{w\}, (T'', \mathsf{bag}''))$ covers $S \cup \hat{W}_{\mathcal{I}}$, and therefore it remains to prove that (T'', bag'') is indeed a tree decomposition of $\mathsf{torso}(X' \cup \{w\})$. It satisfies the vertex condition because (T', bag') satisfied the vertex condition for X' and the vertex w is in the bag $\mathsf{bag}''(t')$. It satisfies the connectedness condition because (T', bag') satisfied the connectedness condition, the vertex w is in no bag of (T', bag'), and $S \cup W_i^f \subseteq \mathsf{bag}'(t)$. It remains to prove that (T'', bag'') satisfies the edge condition, which follows from the edge condition of (T', bag') and the fact that $S \cup W_i^f$ separates w from X'. □

In the algorithm, we will apply Lemma 5.36 together with the Important Separator Hitting Lemma (Lemma 5.23). In particular, we will add only a single vertex of S to W_i in the actual leaf pushing branching. Next we show that if W_i is not a unique largest terminal clique, adding any vertex to W_i increases its flow potential.

Lemma 5.37. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance with $t \geq 2$ that has no safe separations. Let also $i \in [t]$ so that $|W_i| \leq k$ and exists $j \neq i$ so that $|W_j| \geq |W_i|$ and W_j is not a superset of W_i, and let $v \in V(G) \setminus W_i$. Then $\mathsf{flow}\text{-}\Phi_{\mathcal{I}+(W_i,\{v\})}(W_i \cup \{v\}) \geq \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i) + 1$.*

Proof. It suffices to show that $W_i \cup \{v\}$ has flow potential $\mathsf{flow}\text{-}\Phi_{\mathcal{I}+(W_i,\{v\})}(W_i \cup \{v\}) = |W_i \cup \{v\}|$. Suppose otherwise, in particular suppose that there exists a separation (A, S, B) with $W_i \cup \{v\} \subseteq A \cup S$, $\overline{W}_{\mathcal{I}+(W_i,\{v\})}(W_i \cup \{v\}) \subseteq B \cup S$, B non-empty, and $|S| < |W_i \cup \{v\}|$. Note that $|S| < |W_i \cup \{v\}|$ implies that also A is non-empty, and note that because $W_j \neq W_i \cup \{v\}$, we have that $W_j \subseteq \overline{W}_{\mathcal{I}+(W_i,\{v\})}(W_i \cup \{v\}) \subseteq B \cup S$. In particular, S is a (W_i, W_j)-separator. Because $\mathcal{I}$ has no safe separations, by Lemma 5.32, W_i is strongly linked into W_j. Therefore, because $|S| \leq |W_i| \leq |W_j|$, either $S = W_i$ or $S = W_j$. However, in either case (A, S, B) would be a safe separation of $\mathcal{I}$, which is a contradiction. □

Algorithm 1 A $k^{\mathcal{O}(kt)}n^2$ time algorithm for Partitioned Subset Treewidth.

Input: Instance $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$.
Output: Either a solution of $\mathcal{I}$ or NO.

1: **if** $t = 1$ or $|V(G)| \leq k + 2$ **then return** Case-analysis($\mathcal{I}$) ▷ Lemma 5.38
2: **if** Exists a safe separation (A, S, B) **then**
3: **return** Combine(Solve($\mathcal{I} \triangleleft (A, S)$), Solve($\mathcal{I} \triangleleft (B, S)$))
4: **for all** $i, j \in [t]$ with $i \neq j$ and $|W_i \cup W_j| \leq k + 1$ **do**
5: $sol \leftarrow$ Solve($\mathcal{I} \times (W_i, W_j)$)
6: **if** $sol \neq$ NO **then return** sol
7: **if** Exists $i, j \in [t]$ with $W_i \subset W_j$ **then return** NO
8: **for all** $i \in [t]$ so that $|W_i| \leq k$ and exists $j \neq i$ with $|W_j| \geq |W_i|$ **do**
9: **for all** $w \in W_i \setminus \overline{W}_{\mathcal{I}}(W_i)$ **do**
10: $H \leftarrow \mathrm{ImpSepHittingSet}_{G \setminus (W_i \setminus \{w\})}(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i, k)$ ▷ Lemma 5.23
11: **for all** $v \in H \setminus \{w\}$ **do**
12: $sol \leftarrow$ Solve($\mathcal{I} + (W_i, \{v\})$)
13: **if** $sol \neq$ NO **then return** sol
14: **return** NO

5.4.5 The algorithm

In this subsection, we put the reduction rules and branching together to a complete algorithm for Partitioned Subset Treewidth, and analyze the algorithm.

First, we need the following lemma to handle corner cases.

Lemma 5.38. *Instances with $t = 1$ or $|V(G)| \leq k + 2$ can be solved in $\mathcal{O}(m)$ time.*

Proof. If $t = 1$, then because $|W_1| \leq k + 1$, there is a trivial solution where $X = W_1$ and the tree decomposition has a single bag containing X. When $|V(G)| \leq k + 2$, we consider the following cases. First, if $|\hat{W}_{\mathcal{I}}| \leq k + 1$, then again the trivial single-bag solution suffices. Otherwise, we have that $X = \hat{W}_{\mathcal{I}} = V(G)$, and there exists a solution if and only if G is not a complete graph. □

The algorithm for Partitioned Subset Treewidth is presented in the pseudocode Algorithm 1 and described in detail below. In the pseudocode, we denote the recursive calls of the algorithm by the function "Solve".

First, on Line 1, Algorithm 1 handles the special cases of $t = 1$ and $|V(G)| \leq k + 2$ by Lemma 5.38. Then, on Lines 2 and 3 the reduction by safe separations discussed in Subsection 5.4.3 is implemented. In particular, if there exists a safe separation (A, S, B), then the instances $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are solved recursively, and if both of them return a solution the solutions are combined to a solution of $\mathcal{I}$, and if either of them

returns NO, then we return NO. The function "Combine" on Line 3 denotes an operation that returns NO if either of its arguments is NO, and if its arguments are a solution $(X_A, \mathcal{T}_A)$ of $\mathcal{I} \triangleleft (A, S)$ and a solution $(X_B, \mathcal{T}_B)$ of $\mathcal{I} \triangleleft (B, S)$ then it returns the solution $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ of $\mathcal{I}$.

Then, the terminal clique merging branching discussed in Subsection 5.4.4 is implemented on Lines 4 to 6. In particular, the algorithm branches on merging all pairs of terminal cliques W_i, W_j with $|W_i \cup W_j| \leq k+1$ and returns a solution if any of the resulting instances were yes-instances. After this, $\mathcal{I}$ is maximally merged, and this is immediately used on Line 7 to return NO if some terminal clique is a subset of some other terminal clique.

Then, the leaf pushing branching discussed in Subsection 5.4.4 is implemented on Lines 8 to 13. The algorithm branches on all candidates for a potential forget-clique W_i that is not a uniquely largest terminal clique, and a vertex $w \in W_i$ for which there exists an important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator S in the graph $G \setminus (W_i \setminus \{w\})$ so that w and S satisfy the conditions of Lemma 5.36. The algorithm does not branch on all such important separators S, but instead uses the Important Separator Hitting Lemma (Lemma 5.23) to obtain a set of vertices H of size at most k that intersects all important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separators S of size at most k in the graph $G \setminus (W_i \setminus \{w\})$. Then, a single vertex of such an important separator can be guessed by guessing a single vertex in H, so the algorithm branches on all vertices in $H \setminus \{w\}$ to add to W_i. Finally, on Line 14 the algorithm concludes that $\mathcal{I}$ is a no-instance if none of the branches returned a solution.

The algorithm runs in space $n^{\mathcal{O}(1)}$, because each individual recursive call clearly runs in time and space $n^{\mathcal{O}(1)}$, and the depth of the recursion is $n^{\mathcal{O}(1)}$ because at each recursive call either the number of vertices of the graph decreases, or the number of terminal cliques decreases, or the sum of the sizes of the terminal cliques increases, but the number of terminal cliques never increases.

We then prove the correctness of Algorithm 1. Its running time will be analyzed in Subsection 5.4.6.

We start by proving that Algorithm 1 is correct when it returns a solution.

Lemma 5.39. *If Algorithm 1 returns a solution, then it is a solution of $\mathcal{I}$.*

Proof. For the case analysis of Line 1 this is by Lemma 5.38. Then, we use induction on the recursion tree, assuming that the lemma holds for recursive calls of the algorithm.

Now, whenever Algorithm 1 returns on Line 3 after finding a safe separation (A, S, B) and combining solutions of $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ into a solution of $\mathcal{I}$, it is correct

by induction and Lemma 5.26. The cases when the algorithm returns a solution after terminal clique merging on Line 6 or after leaf pushing on Line 13 are correct by induction and the facts that any solution of $\mathcal{I} \times (W_i, W_j)$ is also a solution of $\mathcal{I}$ and any solution of $\mathcal{I} + (W_i, \{v\})$ is also a solution of $\mathcal{I}$. □

We then show that Algorithm 1 is correct when it returns NO.

Lemma 5.40. *If Algorithm 1 returns NO, then $\mathcal{I}$ is a no-instance.*

Proof. For the case analysis of Line 1 this is by Lemma 5.38. Then we use induction on the recursion tree, assuming that the lemma holds for recursive calls of the algorithm. The correctness of safe separation reduction on Line 3 follows from induction and Lemma 5.27.

Then, after the safe separation reduction of Lines 2 and 3 we can assume that $\mathcal{I}$ has no safe separations, and by the terminal clique merging of Lines 4 to 6 and induction we can assume that $\mathcal{I}$ is maximally merged. The correctness of returning NO on Line 7 if there are terminal cliques $W_i \subset W_j$ follows from the facts that $\mathcal{I}$ is maximally merged and if $\mathcal{I}$ would be a yes-instance, then $\mathcal{I} \times (W_i, W_j)$ would also be a yes-instance.

It remains to argue that if Algorithm 1 returns from the final Line 14, then $\mathcal{I}$ is a no-instance. For the sake of contradiction, assume that Algorithm 1 returns NO from Line 14 but $\mathcal{I}$ is a yes-instance. Now, by Lemma 5.34, $\mathcal{I}$ has at least two potential forget-cliques. Let W_i be a smallest potential forget-clique of $\mathcal{I}$. By Lemma 5.36 we have that $|W_i| \leq k$, and therefore W_i satisfies the conditions of Line 8. By Lemma 5.36, there exists a vertex $w \in W_i$ and in $G \setminus (W_i \setminus \{w\})$ a non-empty important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator S disjoint from W_i so that $\mathcal{I} + (W_i, S)$ is a yes-instance. Some iteration of Line 9 will choose this vertex $w \in W_i$, and it holds that $H \cap S \neq \emptyset$, so some iteration of Line 11 will choose a vertex $v \in S$. Because $\mathcal{I} + (W_i, S)$ is a yes-instance, $\mathcal{I} + (W_i, \{v\})$ is also a yes-instance, so by induction we get that Algorithm 1 would return on Line 13, which is a contradiction. □

5.4.6 Running time analysis

We then prove that the running time of Algorithm 1 is $k^{\mathcal{O}(kt)}n^2$. For this, we introduce the measures $\Phi_{\mathcal{I}}(W_i)$ of a terminal clique and $\Phi(\mathcal{I})$ of the instance.

We define the measure of a terminal clique based on its flow potential as

$$\Phi_{\mathcal{I}}(W_i) = 3k + 3 - \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i).$$

Observe that $2k+2 \leq \Phi_{\mathcal{I}}(W_i) \leq 3k+3$, which in particular implies that the sum of measures of two terminal cliques is always at least $k+1$ larger than the measure of a single terminal clique.

Then, the measure of the instance is defined as

$$\Phi(\mathcal{I}) = \sum_{i=1}^{t} \Phi_{\mathcal{I}}(W_i).$$

Note that $(2k+2)t \leq \Phi(\mathcal{I}) \leq (3k+3)t$. We will show the running time of the algorithm to be of form $k^{\mathcal{O}(\Phi(\mathcal{I}))}n^2 = k^{\mathcal{O}(kt)}n^2$.

To this end, we will show that breaking the instance by a safe separation does not increase the measure, and that both the terminal clique merging branching of Line 5 and the leaf pushing branching of Line 12 decrease the measure by at least one. We start by proving the property for safe separations, using Lemmas 5.28 to 5.30.

Lemma 5.41. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and (A, S, B) a safe separation. It holds that $\Phi(\mathcal{I} \triangleleft (A,S)) \leq \Phi(\mathcal{I})$.*

Proof. We consider the three cases of Lemma 5.30. First, if $\mathcal{I} \triangleleft (A,S)$ has less terminal cliques than $\mathcal{I}$, let $t' < t$ be the number of terminal cliques of $\mathcal{I} \triangleleft (A,S)$. Recall from the definition of $\mathcal{I} \triangleleft (A,S)$ that all terminal cliques of $\mathcal{I} \triangleleft (A,S)$ except possibly S are also terminal cliques of $\mathcal{I}$, and moreover $\mathcal{I} \triangleleft (A,S)$ has at least one terminal clique that is a superset of S. Therefore, the conditions of Lemma 5.29 apply for at least $t'-1$ terminal cliques W_i of $\mathcal{I} \triangleleft (A,S)$, in particular, for them $\mathsf{flow}\text{-}\Phi_{\mathcal{I}\triangleleft(A,S)}(W_i) \geq \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i)$ holds by Lemma 5.29 and therefore for them $\Phi_{\mathcal{I}\triangleleft(A,S)}(W_i) \leq \Phi_{\mathcal{I}}(W_i)$. Because the measure of a terminal clique is at least $2k+2$ and at most $3k+3$, it follows that

$$\begin{aligned}
\Phi(\mathcal{I} \triangleleft (A,S)) &\leq \Phi(\mathcal{I}) + 3k + 3 - (2k+2)(t - (t'-1)) \\
&\leq \Phi(\mathcal{I}) + 3k + 3 - (2k+2) \cdot 2 \\
&\leq \Phi(\mathcal{I}) - k - 1.
\end{aligned}$$

Then, if $\{W_1, \ldots, W_t\} \triangleleft (A,S) = \{W_1, \ldots, W_t\}$, the lemma follows directly from Lemma 5.28.

Then, if both $\mathcal{I}$ and $\mathcal{I} \triangleleft (A,S)$ have t terminal cliques and there is a terminal clique W_i of $\mathcal{I}$ with $W_i \cap B \neq \emptyset$, then $\mathcal{I}$ does not contain any terminal clique that is a subset of $A \cup S$ and a superset of S and therefore for all terminal cliques W_j of $\mathcal{I} \triangleleft (A,S)$ except S

we have by Lemma 5.29 that $\mathsf{flow}\text{-}\Phi_{\mathcal{I}\triangleleft(A,S)}(W_j) \geq \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_j)$. Therefore, we have that

$$\Phi(\mathcal{I} \triangleleft (A,S)) \leq \Phi(\mathcal{I}) + \Phi_{\mathcal{I}\triangleleft(A,S)}(S) - \Phi_{\mathcal{I}}(W_i),$$

which by $\mathsf{flow}\text{-}\Phi_{\mathcal{I}\triangleleft(A,S)}(S) \geq \mathsf{flow}\text{-}\Phi_{\mathcal{I}}(W_i)$ (Lemma 5.30) implies $\Phi(\mathcal{I} \triangleleft (A,S)) \leq \Phi(\mathcal{I})$. □

Next, we observe that the terminal clique merging branching of Line 5 decreases the measure of the instance.

Lemma 5.42. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance and W_i, W_j two distinct terminal cliques. It holds that $\Phi(\mathcal{I} \times (W_i, W_j)) \leq \Phi(\mathcal{I}) - k - 1$.*

Proof. This follows from Lemma 5.33 and the facts that $\Phi_{\mathcal{I}}(W_i) + \Phi_{\mathcal{I}}(W_j) \geq 4k + 4$ and $\Phi_{\mathcal{I}\times(W_i,W_j)}(W_i \cup W_j) \leq 3k + 3$. □

Then, we observe that Lemma 5.37 implies that the leaf pushing branching of Line 12 decreases the measure of the instance.

Lemma 5.43. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance with $t \geq 2$ that has no safe separations. Let also $i \in [t]$ so that $|W_i| \leq k$ and exists $j \neq i$ so that $|W_j| \geq |W_i|$ and W_j is not a superset of W_i, and let $v \in V(G) \setminus W_i$. It holds that $\Phi(\mathcal{I} + (W_i, \{v\})) \leq \Phi(\mathcal{I}) - 1$.*

Proof. Observe that adding a vertex to a terminal clique does not decrease the flow potentials of other terminal cliques. Therefore, the lemma holds by Lemma 5.37. □

Then for the reduction by safe separations we have to argue that the sum of the sizes of the instances $\mathcal{I} \triangleleft (A,S)$ and $\mathcal{I} \triangleleft (B,S)$ is less than the size of $\mathcal{I}$. For this, we formally define the *size* of $\mathcal{I}$ to be

$$\mathsf{size}(\mathcal{I}) = \max(1, (k+2)|V(G)| - (k+2)^2).$$

Note that $\mathsf{size}(\mathcal{I}) = 1$ if and only if $|V(G)| \leq k + 2$.

We show that if (A, S, B) is a strict separation and $|S| \leq k + 1$, then with respect to the $\mathsf{size}(\mathcal{I})$ measure, the instances $\mathcal{I} \triangleleft (A,S)$ and $\mathcal{I} \triangleleft (B,S)$ are in total smaller than $\mathcal{I}$ if $|V(G)| \geq k + 3$.

Lemma 5.44. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be instance with $|V(G)| \geq k+3$ and (A, S, B) a strict separation with $|S| \leq k + 1$. Then $\mathsf{size}(\mathcal{I} \triangleleft (A,S)) + \mathsf{size}(\mathcal{I} \triangleleft (B,S)) \leq \mathsf{size}(\mathcal{I}) - 1$.*

Proof. First, because (A, S, B) is a strict separation, $|V(G)| \geq k+3$, and $k \geq 1$, both of $\mathsf{size}(\mathcal{I} \triangleleft (A, S)) + 2 \leq \mathsf{size}(\mathcal{I})$ and $\mathsf{size}(\mathcal{I} \triangleleft (B, S)) + 2 \leq \mathsf{size}(\mathcal{I})$ hold. This implies that the lemma holds whenever $\mathsf{size}(\mathcal{I} \triangleleft (A, S)) = 1$ or $\mathsf{size}(\mathcal{I} \triangleleft (B, S)) = 1$. It remains to consider the case where $\mathsf{size}(\mathcal{I} \triangleleft (A, S)) > 1$ and $\mathsf{size}(\mathcal{I} \triangleleft (B, S)) > 1$, in particular where $|A \cup S| \geq k+3$ and $|B \cup S| \geq k+3$.

In this case

$$\begin{aligned}
&\mathsf{size}(\mathcal{I} \triangleleft (A, S)) + \mathsf{size}(\mathcal{I} \triangleleft (B, S)) \\
&\qquad = (k+2)|A \cup S| - (k+2)^2 + (k+2)|B \cup S| - (k+2)^2 \\
&\qquad = (k+2)(|A \cup S| + |B \cup S|) - 2(k+2)^2 \\
&\qquad = (k+2)(|V(G)| + |S|) - 2(k+2)^2 \\
&\qquad = (k+2)(|V(G)| + |S| - k - 2) - (k+2)^2 \\
&\qquad \leq \mathsf{size}(\mathcal{I}) - 1.
\end{aligned}$$

□

We then put the running time analysis together.

Lemma 5.45. *Algorithm 1 runs in time $k^{\mathcal{O}(kt)} n^2$.*

Proof. First, we observe that all of the operations in a single call of the recursive procedure can be performed in $k^{\mathcal{O}(1)} m'$ time, where m' is the number of edges in the instance given to the recursive call. In particular, the case analysis of Line 1 can be implemented in $k^{\mathcal{O}(1)} m'$ time by Lemma 5.38, reducing by safe separations on Lines 2 and 3 can be implemented in $k^{\mathcal{O}(1)} m'$ time by Lemma 5.31, for terminal clique merging on Lines 4 to 6 this is trivial, and for leaf pushing on Lines 8 to 13 it is an application of the Important Separator Hitting Lemma (Lemma 5.23).

By the definition of $\mathcal{I} \triangleleft (A, S)$, observe that at each recursive call the current graph can be obtained from an induced subgraph of the original graph by adding all edges inside the terminal cliques, and therefore we can bound $m' \leq k^{\mathcal{O}(1)} m \leq k^{\mathcal{O}(1)} n$, where m is the number of original edges. Therefore, the running time of the algorithm can be bounded by $k^{\mathcal{O}(1)} n \cdot R(\mathcal{I})$, where $R(\mathcal{I})$ is the total number of recursive calls.

We show by induction that the number of recursive calls is bounded by

$$R(\mathcal{I}) \leq \mathsf{size}(\mathcal{I}) \cdot ((k+2)^3)^{\Phi(\mathcal{I})} = k^{\mathcal{O}(1)} n \cdot k^{\mathcal{O}(kt)} = k^{\mathcal{O}(kt)} n.$$

First, when the algorithm returns on Line 1, this holds because $\mathsf{size}(\mathcal{I}) \geq 1$ and $\Phi(\mathcal{I}) \geq 1$ always. Then we can assume that $t \geq 2$ and $|V(G)| \geq k+3$. If there exists a safe separation (A, S, B), then the number of recursive calls is

$$\begin{aligned} R(\mathcal{I}) =& 1 + R(\mathcal{I} \triangleleft (A,S)) + R(\mathcal{I} \triangleleft (B,S)) \\ \leq& 1 + (\mathsf{size}(\mathcal{I} \triangleleft (A,S)) + \mathsf{size}(\mathcal{I} \triangleleft (B,S))) \cdot ((k+2)^3)^{\Phi(\mathcal{I})} && \text{(by Lemma 5.41)} \\ \leq& \mathsf{size}(\mathcal{I}) \cdot ((k+2)^3)^{\Phi(\mathcal{I})}. && \text{(by Lemma 5.44)} \end{aligned}$$

If no safe separators exist, then all recursive calls are from terminal clique merging on Line 5 and leaf pushing on Line 12. By Lemma 5.42, for recursive calls made from terminal clique merging on Line 5 it holds that $\Phi(\mathcal{I} \times (W_i, W_j)) \leq \Phi(\mathcal{I}) - 1$, and by Lemma 5.43, for recursive calls made from leaf pushing on Line 12 it holds that $\Phi(\mathcal{I} + (W_i, \{v\})) \leq \Phi(\mathcal{I}) - 1$. The total number of recursive calls from Lines 5 and 12 is at most $t^2 + tk^2 \leq (k+2)^3 - 1$, so we get that

$$R(\mathcal{I}) \leq 1 + ((k+2)^3 - 1) \cdot \mathsf{size}(\mathcal{I}) \cdot ((k+2)^3)^{\Phi(\mathcal{I})-1} \leq \mathsf{size}(\mathcal{I}) \cdot ((k+2)^3)^{\Phi(\mathcal{I})}.$$

□

This finishes the proof of Theorem 5.5, and together with Theorem 5.6 they imply Theorem 1.3.

5.5 Faster algorithm for Subset Treewidth

This section is devoted to proving Theorem 5.3, in particular, to giving a $2^{\mathcal{O}(k^2)}n^2$ time algorithm for Subset Treewidth. The algorithm of the previous section already gives a $k^{\mathcal{O}(k^2)}n^2$ time algorithm for Subset Treewidth, so this section can be seen as an optimization of the base of the exponent from $k^{\mathcal{O}(1)}$ to $\mathcal{O}(1)$. We will re-use many definitions and lemmas of Section 5.4. In particular, we use the definition of an instance of Partitioned Subset Treewidth from Section 5.4, observing that an instance of Subset Treewidth can be seen as an instance of Partitioned Subset Treewidth having initially $t = |W| = k+2$ terminal cliques of size 1.

The algorithm for Subset Treewidth will use similar concepts to the algorithm for Partitioned Subset Treewidth, but a different approach for making progress in the branching. The main measure of progress will be parameter q that states that there are no solutions that contain "internal separations" of order $< q$. Here, an internal separation

of a solution $(X, \mathcal{T})$ means a separation (A, S, B) so that S is a subset of some bag of $\mathcal{T}$, and the terminal cliques intersect both A and B. The goal will be to increase q, by first pushing two terminal cliques to be of size at least $\geq q$ by using a version of leaf pushing that guesses the whole important separator instead of only one vertex, and then guessing how a hypothetical internal separation of order q would split the terminal cliques and breaking the instance by an important separator of size q pushed towards the side with two terminal cliques of size $\geq q$. We will also argue about internal separations that contain only a small number of "original" terminal vertices behind them, in particular, we will use an observation that if a solution has an internal separation (A, S, B) so that at most $k + 1 - |S|$ original terminal vertices are "behind" the terminal cliques intersecting A, then the whole A-side of the solution can be replaced by just a single bag containing S and the original terminal vertices behind it.

The rest of this section is organized as follows. In Subsection 5.5.1 we introduce the concept of a terminal clique covering an original terminal vertex and based on that the concept of a degenerate separation. In Subsection 5.5.2 we introduce a new parameter to measure the progress of the algorithm and argue how different operations on instances preserve so called "valid instances". In Subsection 5.5.3 we discuss the branching rules of the algorithm and in Subsection 5.5.4 we describe the algorithm and put together its correctness proof. In Subsection 5.5.5 we analyze the running time.

5.5.1 Terminal covers and degenerate separations

We extend the definition of an instance of Partitioned Subset Treewidth given in Section 5.4. We now keep track also of the original input graph G_O and the set of original terminal vertices W_O, which were the original input to the Subset Treewidth problem. In particular, $|W_O| = k + 2$. Observe that the recursive algorithm of Section 5.4 maintains that if $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ is an instance in some recursive call, then $V(G) \subseteq V(G_O)$ and $E(G) \supseteq E(\mathsf{torso}_{G_O}(V(G)))$. The operations $\mathcal{I} \times (W_i, W_j)$ and $\mathcal{I} + (W_i, A)$ trivially maintain this because they change G only by adding edges, and the $\mathcal{I} \triangleleft (A, S)$ operation with a separation (A, S, B) maintains this because S becomes a clique in $G \triangleleft (A, S)$.

Terminal covers

Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k)$ be an instance, G_O the original graph, and W_O the set of original terminal vertices. We say that a terminal clique W_i *covers* an original terminal vertex $w \in W_O$ if W_i is a $(\{w\}, V(G))$-separator in G_O. We observe that in the algorithm of the previous section, at every point for every original terminal vertex there exists a

terminal clique that covers it, and moreover every terminal clique covers at least one original terminal vertex.

In the algorithm of this section we maintain a mapping $\mathsf{tc} : W_O \to \{W_1, \ldots, W_t\}$ from the original terminal vertices to the current terminal cliques, so that for all $w \in W_O$, the terminal clique $\mathsf{tc}(w)$ covers w. We extend the definition of an instance to include the mapping tc, in particular an instance is now a 4-tuple $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc})$. (The instance also implicitly contains G_O and W_O, but we do not write them explicitly because they do not change in the branching.)

Let us now define how the mapping is maintained under the operation $\mathcal{I} \triangleleft (A, S)$, where (A, S, B) is a separation. Let S' be a terminal clique of $\mathcal{I} \triangleleft (A, S)$ that is a superset of S. If there are multiple such terminal cliques, then let S' be the lexicographically first choice. Then, we define

$$\mathsf{tc}(w) \triangleleft (A, S) = \begin{cases} S' \text{ if } \mathsf{tc}(w) \subseteq B \cup S \\ \mathsf{tc}(w) \text{ otherwise.} \end{cases}$$

Now, we define $\mathcal{I} \triangleleft (A, S) = (G \triangleleft (A, S), \{W_1, \ldots, W_t\} \triangleleft (A, S), k, \mathsf{tc} \triangleleft (A, S))$. The following lemma shows that this correctly maintains the mapping tc.

Lemma 5.46. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc})$ be an instance, G_O the original graph, and $w \in W_O$ an original terminal vertex. Let also (A, S, B) be a separation of G. If $\mathsf{tc}(w) \subseteq B \cup S$, then any terminal clique of $\mathcal{I} \triangleleft (A, S)$ that is a superset of S covers w in $\mathcal{I} \triangleleft (A, S)$. Otherwise, $\mathsf{tc}(w)$ is a terminal clique of $\mathcal{I} \triangleleft (A, S)$ and covers w in $\mathcal{I} \triangleleft (A, S)$.*

Proof. First, if $\mathsf{tc}(w)$ intersects A, in which case $\mathsf{tc}(w) \in \{W_1, \ldots, W_t\} \triangleleft (A, S)$, the fact that $\mathsf{tc}(w)$ covers w in $\mathcal{I} \triangleleft (A, S)$ holds directly by the fact that $\mathsf{tc}(w)$ covers w in $\mathcal{I}$.

Then, consider the case when $\mathsf{tc}(w) \subseteq B \cup S$. Recall that by definition $\mathsf{tc}(w)$ is a $(\{w\}, V(G))$-separator in G_O. We will show that S is a $(\{w\}, V(G \triangleleft (A, S)))$-separator in G_O. Consider any path from w to $V(G \triangleleft (A, S)) = A \cup S$ in G_O. Because $\mathsf{tc}(w)$ covers w in $\mathcal{I}$, this path intersects $\mathsf{tc}(w)$, and therefore it has a suffix that is a $\mathsf{tc}(w)$-$A \cup S$-path in G_O. Now, because $E(G) \supseteq E(\mathsf{torso}_{G_O}(V(G)))$, we can map the suffix into a $\mathsf{tc}(w)$-$A \cup S$-path in G by just removing vertices in $G_O \setminus V(G)$ from it. Then, because S is a $(\mathsf{tc}(w), A \cup S)$-separator in G, this path must intersect S, and therefore the path from w to $A \cup S$ in G_O must also intersect S, and therefore S is a $(\{w\}, A \cup S)$-separator in G_O. □

We also define the maintenance of tc under terminal clique merging by

$$\mathsf{tc}(w) \times (W_i, W_j) = \begin{cases} W_i \cup W_j \text{ if } \mathsf{tc}(w) = W_i \text{ or } \mathsf{tc}(w) = W_j \\ \mathsf{tc}(w) \text{ otherwise.} \end{cases}$$

Now, $\mathcal{I} \times (W_i, W_j) = (G \times (W_i, W_j), \{W_1, \ldots, W_t\} \times (W_i, W_j), k, \mathsf{tc} \times (W_i, W_j))$. This maintains the mapping tc correctly because if W_i is a $(\{w\}, V(G))$-separator in G_O, then also $W_i \cup W_j$ is a $(\{w\}, V(G))$-separator in G_O.

Similarly, when adding vertices to terminal cliques it is defined by

$$\mathsf{tc}(w) + (W_i, A) = \begin{cases} W_i \cup A \text{ if } \mathsf{tc}(w) = W_i \\ \mathsf{tc}(w) \text{ otherwise.} \end{cases}$$

Then, $\mathcal{I} + (W_i, A) = (G + (W_i, A), \{W_1, \ldots, W_t\} + (W_i, A), k, \mathsf{tc} + (W_i, A))$. Again, this clearly maintains the mapping tc correctly.

For a terminal clique W_i, we denote by $\#\mathsf{tc}_{\mathcal{I}}(W_i)$ the number of original terminal vertices mapped to W_i by tc, i.e., $\#\mathsf{tc}_{\mathcal{I}}(W_i) = |\{w \in W_O \mid \mathsf{tc}(w) = W_i\}|$. Note that the operations $\mathcal{I} \times (W_i, W_j)$ and $\mathcal{I} + (W_i, A)$ preserve the invariant that $\#\mathsf{tc}_{\mathcal{I}}(W_i) \geq 1$ for all terminal cliques W_i, and the operation $\mathcal{I} \triangleleft (A, S)$ for a separation (A, S, B) preserves this if there is at least one terminal clique that is a subset of $B \cup S$ or a superset of S, which will be always the case when this operation is used.

Degenerate separations

Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc})$ be an instance, and $(X, (T, \mathsf{bag}))$ a solution of $\mathcal{I}$. We say that a separation (A, S, B) of G is an *internal separation* of the solution $(X, (T, \mathsf{bag}))$ for $\mathcal{I}$ if $S \subseteq \mathsf{bag}(t)$ for some $t \in V(T)$ and $\hat{W}_{\mathcal{I}}$ intersects both A and B.

We say that an internal separation (A, S, B) is *degenerate* if

$$|S| + \sum_{W_i \mid W_i \cap A \neq \emptyset} \#\mathsf{tc}_{\mathcal{I}}(W_i) \leq k + 1.$$

Note that if a solution contains a degenerate internal separation (A, S, B), then for the purpose of obtaining a solution of the original instance we can, slightly informally speaking, replace the decomposition on the A-side of the separation by just a single bag $S \cup \{w \in W_O \mid \mathsf{tc}(w) \cap A \neq \emptyset\}$ because the definition states that this bag has size at most

$k+1$. In particular, we observe that for the original instance there always exists a solution where every degenerate internal separation is a separation between a leaf bag and the rest of the decomposition, with the leaf bag containing only the separator S and the original terminal vertices "behind" S. Now, our goal is to perform a *pre-branching* step that by using important separators guesses, in some sense, a maximal set of degenerate internal separations, and after that arrives to an instance where no solution has a degenerate internal separation.

We say that an instance $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc})$ is *valid* if it is a yes-instance and has no solution $(X, (T, \mathsf{bag}))$ that has a degenerate internal separation for $\mathcal{I}$. Otherwise, we say that $\mathcal{I}$ is *invalid*. Next we show that by performing the pre-branching step, we can assume that we start with a valid instance.

Lemma 5.47 (Pre-branching)**.** *There is an algorithm, that given a graph G, integer k, and a set W with $|W| = k+2$, in time $2^{\mathcal{O}(k^2)}m$ and space $n^{\mathcal{O}(1)}$ enumerates $2^{\mathcal{O}(k^2)}$ instances $(G, \{W_1, \ldots, W_t\}, k, \mathsf{tc})$ with $t \leq k+2$ so that any solution of any of the instances can in time $k^{\mathcal{O}(1)}m$ be turned into a torso tree decomposition of width at most k in G that covers W, and moreover if such a torso tree decomposition exists, then at least one of the returned instances is valid.*

Proof. Let the initial instance be $\mathcal{I} = (G, \{W_1, \ldots, W_{|W|}\}, k, \mathsf{tc})$, where $\{W_1, \ldots, W_{|W|}\}$ is a partition of $W = W_O$ into single vertices and $\mathsf{tc}(w) = \{w\}$. We then branch on all possible ways to perform terminal clique merging operations (recall the definitions from Subsection 5.4.4). There are $k^{\mathcal{O}(k)}$ possible sequences of terminal clique merging.

Observe that any solution of any of the resulting instances is a torso tree decomposition of width at most k in G that covers W, and moreover if a solution exists, then at least one of the resulting instances is a maximally merged yes-instance. Notice also that $\#\mathsf{tc}_{\mathcal{I}}(W_i) = |W_i|$ holds for all terminal cliques W_i in instances obtained in this manner. We will then prove the lemma with the assumption that we start with an instance for which $\#\mathsf{tc}_{\mathcal{I}}(W_i) = |W_i|$ holds, and in particular, if the starting instance is a maximally merged yes-instance, then at least one of the outputs will be a valid instance.

We will do branching that maintains a partition of terminal cliques into *processed* terminal cliques and *unprocessed* terminal cliques. Initially, all of the terminal cliques are unprocessed. This branching will always maintain that for unprocessed terminal cliques W_i it holds that $\#\mathsf{tc}_{\mathcal{I}}(W_i) \geq |W_i|$, and in the "success" branches the following invariants will be maintained

1. $\mathcal{I}$ is a yes-instance,

2. for every processed terminal clique W_i there exists no solution that contains a degenerate internal separation (A, S, B) so that A intersects W_i, and

3. there exists no solution that contains a degenerate internal separation (A, S, B) so that more than one terminal clique intersects A.

Initially, the invariant $\#\mathsf{tc}_{\mathcal{I}}(W_i) \geq |W_i|$ holds as discussed earlier. Also, if $\mathcal{I}$ is initially a maximally merged yes-instance, the invariant of Item 1 holds by definition, and the invariant of Item 2 initially holds by the fact that there are no processed terminal cliques. The invariant of Item 3 initially holds if $\mathcal{I}$ is maximally merged, because if there would be a solution $(X, (T, \mathsf{bag}))$ with a degenerate internal separation (A, S, B), so that at least two terminal cliques intersect A, then by the definition $|S| + \sum_{W_i | W_i \cap A \neq \emptyset} \#\mathsf{tc}_{\mathcal{I}}(W_i) \leq k+1$ and the fact that $\#\mathsf{tc}_{\mathcal{I}}(W_i) \geq |W_i|$ we could replace the solution on the subgraph induced by $A \cup S$ by just a single bag $S \cup \bigcup_{W_i | W_i \cap A \neq \emptyset} W_i$, and conclude that $\mathcal{I}$ is not maximally merged.

The branching works as follows. While there exists an unprocessed terminal clique W_i, we branch on the cases that either there exists no solution that contains a degenerate internal separation (A, S, B) with $W_i \cap A \neq \emptyset$, recursing to the case where we just mark W_i as processed, or that there exists a solution that contains a degenerate internal separation (A, S, B) so that $W_i \cap A \neq \emptyset$, and in this case we recurse on all cases that are obtained by taking an important $(W_i, \overline{W}_{\mathcal{I}}(W_i))$-separator S' of size $|S'| \leq k + 1 - \#\mathsf{tc}_{\mathcal{I}}(W_i)$, letting $(A', S', B') = (\mathsf{reach}_G(W_i, S'), S', V(G) \setminus (\mathsf{reach}_G(W_i, S') \cup S'))$, and recursing to the instance $\mathcal{I} \triangleleft (B', S')$ with S' marked as processed if it is a terminal clique of $\mathcal{I} \triangleleft (B', S')$.

Observe that the branching cannot decrease $\#\mathsf{tc}_{\mathcal{I}}(W_i)$ for any unprocessed terminal clique W_i, so the invariant that $\#\mathsf{tc}_{\mathcal{I}}(W_i) \geq |W_i|$ for them is maintained. Note also that in each branch, the number of unprocessed terminal cliques decreases. In particular, in the corner case when S' is not a terminal clique of $\mathcal{I} \triangleleft (B', S')$, it holds that the terminal cliques of $\mathcal{I} \triangleleft (B', S')$ are a subset of the terminal cliques of $\mathcal{I}$ but do not contain W_i. As the number of important separators S' of size at most k is at most 4^k by Lemma 5.22, we branch to at most $4^k + 1$ directions every time, and therefore as initially there are at most $k + 2$ unprocessed terminal cliques, the branching tree has size at most $(4^k + 1)^{k+2} = 2^{\mathcal{O}(k^2)}$.

The leaves of the branching tree have no unprocessed terminal cliques and will be the instances we output. Note that because when taking the important separator S' we impose the condition $|S'| \leq k + 1 - \#\mathsf{tc}_{\mathcal{I}}(W_i)$, which by $\#\mathsf{tc}_{\mathcal{I}}(W_i) \geq |W_i|$ implies that $|S'| + |W_i| \leq k + 1$, we can construct from a solution of $\mathcal{I} \triangleleft (B', S')$ a solution of $\mathcal{I}$ by just attaching a bag $W_i \cup S'$ as a neighbor of a bag containing S'. Therefore, from any solution of the outputted instance we can construct a solution of the original instance.

Note that if the invariants of Items 1 to 3 hold for some outputted instance, then it is a valid instance. Therefore, it remains to argue that if the invariants of Items 1 to 3 initially hold, then they hold for at least one of the branches.

Suppose that the invariants of Items 1 to 3 hold for $\mathcal{I}$, and let W_i be an unprocessed terminal clique that we are branching on. First, if there exists no solution that contains a degenerate internal separation (A, S, B) so that A intersects W_i, the invariants are clearly maintained by just marking W_i processed as it does not change the set of solutions of the instance or the set $\hat{W}$. Then, suppose there exists a solution $(X, (T, \mathsf{bag}))$ that contains a degenerate internal separation (A, S, B) so that A intersects W_i. First, by Item 3, W_i must be the only terminal clique that intersects A. Then, we consider a hypothetical solution $(X, (T, \mathsf{bag}))$ and a degenerate internal separation (A, S, B) of it so that $\mathsf{reach}_G(W_i, S)$ is the largest possible over all such $(X, (T, \mathsf{bag}))$ and (A, S, B). We note that $W_i \subseteq A \cup S$ and $\overline{W}_{\mathcal{I}}(W_i) \subseteq B \cup S$, so S is a $(W_i, \overline{W}_{\mathcal{I}}(W_i))$-separator. Then we let S' to be a smallest important $(W_i, \overline{W}_{\mathcal{I}}(W_i))$-separator that dominates S, and will argue that the branch that selects S' as the important separator will maintain the invariants. We denote $(A', S', B') = (\mathsf{reach}_G(W_i, S'), S', V(G) \setminus (\mathsf{reach}_G(W_i, S') \cup S'))$. Note that W_i intersects A' because W_i intersects $\mathsf{reach}_G(W_i, S)$. We will argue that $\mathcal{I} \triangleleft (B', S')$ satisfies the invariants of Items 1 to 3.

Item 1. First, to show that $\mathcal{I} \triangleleft (B', S')$ is a yes-instance, we use that by Lemma 5.18, S' is linked into $S \cap (A' \cup S')$. We apply the Pulling Lemma (Lemma 5.8) with the separation (B', S', A'), the torso tree decomposition $(X, (T, \mathsf{bag}))$, and the bag of (T, bag) that contains S, and obtain a torso tree decomposition $((X \cap B') \cup S', (T', \mathsf{bag}'))$ of no larger width that covers $\hat{W}_{\mathcal{I} \triangleleft (B', S')}$, showing that $\mathcal{I} \triangleleft (B', S')$ is a yes-instance.

Item 3. Then, to argue that $\mathcal{I} \triangleleft (B', S')$ does not have solutions with degenerate internal separations (A, S, B) with multiple terminal cliques intersecting A, suppose that $(X_d, (T_d, \mathsf{bag}_d))$ is a solution of $\mathcal{I} \triangleleft (B', S')$ that contains a degenerate internal separation (A_d, S_d, B_d) so that at least two terminal cliques of $\mathcal{I} \triangleleft (B', S')$ intersect A_d. Now, because W_i is the only terminal clique of $\mathcal{I}$ that intersects A' and by the fact that $|S'| \leq |S|$ and $|W_i| + |S| \leq k + 1$ we can turn $(X_d, (T_d, \mathsf{bag}_d))$ into a solution $(X_d \cup W_i, (T'_d, \mathsf{bag}'_d))$ of $\mathcal{I}$ by just attaching a bag $W_i \cup S'$ to a bag of (T'_d, bag'_d) that contains S'.

Then, if $S' \subseteq S_d \cup B_d$, we consider the separation $(A_d, S_d, B_d \cup A')$ of G. The set $\hat{W}_{\mathcal{I}}$ intersects A_d because at least two terminal cliques of $\mathcal{I} \triangleleft (B', S')$ intersect A_d, and it intersects $B_d \cup A'$ because W_i intersects A', and therefore $(A_d, S_d, B_d \cup A')$ is an internal separation for the solution $(X_d \cup W_i, (T'_d, \mathsf{bag}'_d))$ of $\mathcal{I}$. In this case, as $A_d \subseteq B'$, all terminal cliques of $\mathcal{I} \triangleleft (B', S')$ that intersect A_d are also terminal cliques of $\mathcal{I}$ that intersect A_d and

vice versa, and therefore at least two terminal cliques of $\mathcal{I}$ intersect A_d and $(A_d, S_d, B_d \cup A')$ is degenerate also for $\mathcal{I}$, which would contradict that $\mathcal{I}$ satisfies the invariant of Item 3.

The other case is that S' intersects A_d and is a subset of $A_d \cup S_d$, in which case we consider the separation $(A_d \cup A', S_d, B_d)$ of G. The set $\hat{W}_{\mathcal{I}}$ intersects $A_d \cup A'$ because W_i intersects A', and it intersects B_d because $B_d \subseteq B'$ and $\hat{W}_{\mathcal{I}} \cap B' = \hat{W}_{\mathcal{I} \triangleleft (B', S')} \cap B'$, and therefore it is an internal separation for the solution $(X_d \cup W_i, (T'_d, \mathsf{bag}'_d))$ of $\mathcal{I}$. At least two terminal cliques of $\mathcal{I}$ intersect $A_d \cup A'$ because W_i intersects A', and some terminal clique of $\mathcal{I} \triangleleft (B', S')$ that is also a terminal clique of $\mathcal{I}$ must intersect A_d because all but at most one terminal clique of $\mathcal{I} \triangleleft (B', S')$ is a terminal clique of $\mathcal{I}$ and at least two terminal cliques of $\mathcal{I} \triangleleft (B', S')$ intersect A_d. Now, because S' intersects A_d and all terminal vertices covered by terminal cliques of $\mathcal{I}$ that are subsets of $A' \cup S'$ were mapped into S' or to the superset of S' in $\mathcal{I} \triangleleft (B', S')$, the internal separation $(A_d \cup A', S_d, B_d)$ is degenerate for $\mathcal{I}$, which would contradict that $\mathcal{I}$ satisfies the invariant of Item 3.

Item 2. Then, to argue that $\mathcal{I} \triangleleft (B', S')$ has no solution with a degenerate internal separation (A, S, B) with a processed terminal clique intersecting A, let W_j be a processed terminal clique of $\mathcal{I} \triangleleft (B', S')$ and suppose that $(X_d, (T_d, \mathsf{bag}_d))$ is a solution of $\mathcal{I} \triangleleft (B', S')$ that contains a degenerate internal separation (A_d, S_d, B_d) so that W_j intersects A_d but no other terminal clique of $\mathcal{I} \triangleleft (B', S')$ intersects A_d. (Note that if also some other terminal clique intersects A_d, then we are in the already proven case of Item 3.) Again, because W_i is the only terminal clique of $\mathcal{I}$ that intersects A' and by the facts that $|S'| \leq |S|$ and $|W_i| + |S| \leq k+1$ we can turn $(X_d, (T_d, \mathsf{bag}_d))$ into a solution $(X_d \cup W_i, (T'_d, \mathsf{bag}'_d))$ of $\mathcal{I}$ just by attaching a bag $W_i \cup S'$ to a bag of (T_d, bag_d) that contains S'.

Then, if $S' \subseteq S_d \cup B_d$, we consider the separation $(A_d, S_d, B_d \cup A')$ of G. Now, because $S' \subseteq S_d \cup B_d$ but W_j intersects A_d, we have that $W_j \neq S'$, implying that W_j is also a terminal clique of $\mathcal{I}$ and therefore A_d intersects $\hat{W}_{\mathcal{I}}$, and $B_d \cup A'$ intersects $\hat{W}_{\mathcal{I}}$ because W_i intersects A', and therefore $(A_d, S_d, B_d \cup A')$ is an internal separation of the solution $(X_d \cup W_i, (T'_d, \mathsf{bag}'_d))$ for $\mathcal{I}$. Because W_j is a processed terminal clique of $\mathcal{I} \triangleleft (B', S')$ and a terminal clique of $\mathcal{I}$, and $W_j \neq W_i$ and $W_j \neq S'$, W_j is also a processed terminal clique of $\mathcal{I}$, and moreover because $A_d \subseteq B'$, also no other terminal cliques of $\mathcal{I}$ intersect A_d, and therefore we contradict that $\mathcal{I}$ satisfies the invariant of Item 2.

The other case is that S' intersects A_d and is a subset of $A_d \cup S_d$. Note that in this case $S' \subseteq W_j$ and W_j is the only terminal clique of $\mathcal{I} \triangleleft (B', S')$ that is a superset of S'. We then consider the separation $(A_d \cup A', S_d, B_d)$ of G. Because W_i intersects A' and $B_d \subseteq B'$ this is an internal separation for $\mathcal{I}$. Note that because $W_i \subseteq A' \cup S'$, the original terminal vertices mapped to W_i in $\mathcal{I}$ are mapped to W_j in $\mathcal{I} \triangleleft (B', S')$.

If any other terminal clique of $\mathcal{I}$ than W_i intersects $A_d \cup A'$, then it must either be also a terminal clique of $\mathcal{I} \triangleleft (B', S')$ that intersects A_d (in particular, W_j) or a subset of $A' \cup S'$ whose covered original terminal vertices are mapped to W_j in $\mathcal{I} \triangleleft (B', S')$. In that case, we contradict that $\mathcal{I}$ satisfies the invariant of Item 3. Then, if the only terminal clique of $\mathcal{I}$ that intersects $A_d \cup A'$ is W_i, we observe that S_d is a $(W_i, \overline{W}_{\mathcal{I}}(W_i))$-separator, and moreover because S_d does not intersect A' and S' intersects A_d we have that $\mathsf{reach}_G(W_i, S') \subset \mathsf{reach}_G(W_i, S_d)$. Because $\#\mathsf{tc}_{\mathcal{I} \triangleleft (B', S')}(W_j) \geq \#\mathsf{tc}_{\mathcal{I}}(W_i)$, it holds that $|W_i| + |S_d| \leq k+1$, and therefore $(A_d \cup A', S_d, B_d)$ is a degenerate internal separation for $\mathcal{I}$, and therefore as $\mathsf{reach}_G(W_i, S) \subseteq \mathsf{reach}_G(W_i, S') \subset \mathsf{reach}_G(W_i, S_d)$, it contradicts the choice of (A, S, B). □

5.5.2 Maintaining valid instances

We introduce a new parameter of the instance based on the minimum order of a an internal separation in a solution. In particular, we maintain a integer q so that, informally speaking, in the success branches it is guaranteed that there exists no solution that contains an internal separation of order less than q. More formally, we further extend the definition of an instance to now be a 5-tuple $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$, re-using all previous definitions but now also including an integer $q \in [0, k+2]$. We now say that $\mathcal{I}$ is valid if it is a yes-instance, there exists no solution that contains a degenerate internal separation, and there exists no solution that contains an internal separation of order less than q. Otherwise, we say that $\mathcal{I}$ is invalid. Note that a valid instance in the sense of Subsection 5.5.1 can be turned into valid instance of this sense by just setting $q = 0$. The definitions $\triangleleft$, $\times$, and $+$ used for manipulating the instance are extended so that they do not change q. The rest of this section will be devoted to designing a branching algorithm for either finding a solution of $\mathcal{I}$ or concluding that $\mathcal{I}$ is invalid.

We first give a general lemma that will be used for arguing that if we break the instance by a separation (A, S, B), then the resulting instances $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid.

Lemma 5.48. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be a valid instance and (A, S, B) a separation of G so that at least one terminal clique either intersects A or is a superset of S and at least one terminal clique either intersects B or is a superset of S. If both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are yes-instances, then both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid.*

Proof. By symmetry it suffices to prove that $\mathcal{I} \triangleleft (A, S)$ is valid, so for the sake of contradiction suppose that $\mathcal{I} \triangleleft (A, S)$ is invalid. Because $\mathcal{I} \triangleleft (A, S)$ is a yes-instance, it has a solution that contains a degenerate internal separation or an internal separation of

order $< q$. Let $(X_A, \mathcal{T}_A)$ be such a solution of $\mathcal{I} \triangleleft (A, S)$ and $(X_B, \mathcal{T}_B)$ any solution of $\mathcal{I} \triangleleft (B, S)$.

By Lemma 5.26, $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ is a solution of $\mathcal{I}$. Let (A', S', B') be the internal separation of $(X_A, \mathcal{T}_A)$. Now, as S is a clique in $G \triangleleft (A, S)$, we have two cases, either $S \subseteq B' \cup S'$ or $S \subseteq A' \cup S'$ and S intersects A'.

First, if $S \subseteq B' \cup S'$, then consider the separation $(A', S', B \cup B')$ of G. In this situation we have that $\hat{W}_{\mathcal{I}}$ intersects A' because $\hat{W}_{\mathcal{I} \triangleleft (A,S)}$ intersects A' and $S \subseteq B' \cup S'$, and that $\hat{W}_{\mathcal{I}}$ intersects $B \cup B'$ because if it does not intersect B, then $\hat{W}_{\mathcal{I} \triangleleft (A,S)} = \hat{W}_{\mathcal{I}}$, in which case it must intersect B'. Therefore, $(A', S', B \cup B')$ is an internal separation of the solution $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ of $\mathcal{I}$, and therefore if $|S'| < q$ we are done in this case. If (A', S', B') is degenerate internal separation of $(X_A, \mathcal{T}_A)$ in $\mathcal{I} \triangleleft (A, S)$, then it is also a degenerate internal separation of $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ because the original terminal vertices mapped to terminal cliques that intersect A' are the same in both $\mathcal{I}$ and $\mathcal{I} \triangleleft (A, S)$ because $A' \subseteq A$.

Then, if $S \subseteq A' \cup S'$ and S intersects A', consider the separation $(A' \cup B, S', B')$ of G. By the same arguments as in the earlier case, we get that $(A' \cup B, S', B')$ is an internal separation of the solution $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ of $\mathcal{I}$, and again if $|S'| < q$ we are immediately done. Now, if (A', S', B') is a degenerate internal separation of $(X_A, \mathcal{T}_A)$ in $\mathcal{I} \triangleleft (A, S)$, then $(A' \cup B, S', B')$ is degenerate in $\mathcal{I}$ because all original terminal vertices mapped to terminal cliques intersecting B in $\mathcal{I}$ are mapped to a superset of S in $\mathcal{I} \triangleleft (A, S)$, which intersects A'. □

It follows that breaking the instance by safe separations preserves the validity.

Lemma 5.49. *If $\mathcal{I}$ is valid and (A, S, B) is a safe separation, then both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid.*

Proof. By Lemma 5.27 both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are yes-instances. Because (A, S, B) is a safe separation, at least one terminal clique intersects A or is a superset of S and at least one terminal clique intersects B or is a superset of S. Therefore by Lemma 5.48 both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid. □

Then, we observe that safe separations (A, S, B) of order $< q$ can be turned into internal separations of order $< q$ if $\hat{W}_{\mathcal{I}}$ intersects both A and B.

Lemma 5.50. *If an instance $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ has a safe separation (A, S, B) of order $< q$ so that $\hat{W}_{\mathcal{I}}$ intersects both A and B, then $\mathcal{I}$ is invalid.*

Proof. If $\mathcal{I}$ would be valid, then by Lemma 5.27 both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are yes-instances. Let $(X_A, \mathcal{T}_A)$ be a solution of $\mathcal{I} \triangleleft (A, S)$ and $(X_B, \mathcal{T}_B)$ be a solution of $\mathcal{I} \triangleleft (B, S)$. By Lemma 5.26, $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ is a solution of $\mathcal{I}$. However, now (A, S, B) is an internal separation of order $< q$ and thus $\mathcal{I}$ is invalid. □

Then, we show that the notion of maximally merged plays well together with the definition of valid instances.

Lemma 5.51. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance. If for all pairs of distinct terminal cliques W_i, W_j either $|W_i \cup W_j| > k+1$ holds or $\mathcal{I} \times (W_i, W_j)$ is invalid, then $\mathcal{I}$ is either maximally merged or invalid.*

Proof. Suppose this holds and $\mathcal{I}$ is valid but not maximally merged. Now, there exists a pair of distinct terminal cliques W_i, W_j so that $|W_i \cup W_j| \leq k+1$ and $\mathcal{I} \times (W_i, W_j)$ is a yes-instance but invalid. Let $(X, (T, \mathsf{bag}))$ be a solution of $\mathcal{I} \times (W_i, W_j)$ and (A, S, B) an internal separation of $(X, (T, \mathsf{bag}))$ that is either degenerate or has $|S| < q$. Now, $(X, (T, \mathsf{bag}))$ is also a solution of $\mathcal{I}$, and because $\hat{W}_{\mathcal{I}} = \hat{W}_{\mathcal{I} \times (W_i, W_j)}$, (A, S, B) is also an internal separation for $\mathcal{I}$. First, if $|S| < q$, then $\mathcal{I}$ is invalid. Then, if (A, S, B) is degenerate for $\mathcal{I} \times (W_i, W_j)$, then for $\mathcal{I}$ only a smaller number of original terminal vertices are mapped into terminal cliques that intersect A, so (A, S, B) is degenerate for $\mathcal{I}$. □

5.5.3 Branching

In our algorithm, if there are less than 2 terminal cliques of size $\geq q$ we will perform leaf pushing branching to create more terminal cliques of size $\geq q$. Unlike in the leaf pushing of Section 5.4, in the leaf pushing of this section we will add the whole important separator to the terminal clique. We show that like this, we can make a terminal clique of size $< q$ into size $\geq q$ by guessing a single important separator.

Lemma 5.52. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be a maximally merged valid instance with $t \geq 2$, no safe separators, and W_i a potential forget-clique of $\mathcal{I}$. There is a vertex $w \in W_i \setminus \overline{W}_{\mathcal{I}}(W_i)$ and in the graph $G \setminus (W_i \setminus \{w\})$ a non-empty important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator S disjoint from W_i so that $\mathcal{I} + (W_i, S)$ is a valid instance and $q + 1 \leq |W_i \cup S| \leq k + 1$.*

Proof. Let w and S be chosen so that $\mathcal{I} + (W_i, S)$ is a yes-instance, which can be done by Lemma 5.36. Now, suppose that $\mathcal{I} + (W_i, S)$ is invalid and let $(X, \mathcal{T})$ and (A', S', B') be the solution and the internal separation that show that $\mathcal{I} + (W_i, S)$ is invalid. Note that $(X, \mathcal{T})$ is also a solution of $\mathcal{I}$.

First, if $\hat{W}_{\mathcal{I}}$ intersects both A' and B', then $(X, \mathcal{T})$ and (A', S', B') directly show that also $\mathcal{I}$ is invalid. In particular, in the case when (A', S', B') is degenerate for $\mathcal{I} + (W_i, S)$, note that if a terminal clique of $\mathcal{I}$ intersects A' then also the corresponding terminal clique of $\mathcal{I} + (W_i, S)$ also intersects A', and therefore (A', S', B') is also degenerate for $\mathcal{I}$.

Then, if $\hat{W}_{\mathcal{I}}$ does not intersect A' but S does, we have that $W_i \subseteq S'$ and $S \subseteq A' \cup S'$. In this case, denote $R_w = \mathsf{reach}_G(\{w\}, S \cup W_i \setminus \{w\})$ and consider the separation $(A'', S'', B'') = (A' \cup R_w, S' \setminus R_w, B' \setminus R_w)$. This is a separation because the neighborhood of R_w is a subset of $S \cup W_i \subseteq A' \cup S'$. Moreover, it is an internal separation of $(X, \mathcal{T})$ for $\mathcal{I}$ because $w \in A''$ and $\hat{W}_{\mathcal{I}} \cap B' = \hat{W}_{\mathcal{I}} \cap B''$ because $R_w \cap \hat{W}_{\mathcal{I}} = \{w\}$ and $w \notin B'$. If $|S'| < q$ then this immediately shows that $\mathcal{I}$ is invalid. If (A', S', B') is degenerate for $\mathcal{I} + (W_i, S)$, then (A'', S'', B'') is degenerate for $\mathcal{I}$, because by the fact that $w \in W_i \setminus \overline{W}_{\mathcal{I}}(W_i)$, the only terminal clique of $\mathcal{I}$ that intersects A'' is W_i, and $\#\mathsf{tc}_{\mathcal{I}}(W_i) = \#\mathsf{tc}_{\mathcal{I}+(W_i,S)}(W_i \cup S)$ in this case.

Then, if $\hat{W}_{\mathcal{I}}$ does not intersect B' but S does, we do an analogous argument, in particular we again denote $R_w = \mathsf{reach}_G(\{w\}, S \cup W_i \setminus \{w\})$ and consider the separation $(A'', S'', B'') = (A' \setminus R_w, S' \setminus R_w, B' \cup R_w)$. By the same argument as previously, this is an internal separation of $(X, \mathcal{T})$ for $\mathcal{I}$. Again, if $|S'| < q$ then $\mathcal{I}$ is invalid. If (A', S', B') is degenerate for $\mathcal{I} + (W_i, S)$, then (A'', S'', B'') is degenerate for $\mathcal{I}$, because if a terminal clique of $\mathcal{I}$ intersects A'' then a corresponding terminal clique of $\mathcal{I} + (W_i, S)$ intersects A'.

Finally, to show that $|W_i \cup S| \geq q + 1$, we have that if $|W_i \cup S| \leq q$ would hold, then $(W_i \setminus \{w\}) \cup S$ would be a $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator of size $< q$. This would give an internal separation $(A, (W_i \setminus \{w\}) \cup S, B)$ with w intersecting A and $\hat{W}_{\mathcal{I}} \setminus (W_i \cup S)$ intersecting B. Note that $\hat{W}_{\mathcal{I}} \setminus (W_i \cup S) \neq \emptyset$ because otherwise $\mathcal{I}$ would not be maximally merged. □

Then, once there are at least two terminal cliques of size $\geq q$, the algorithm will guess how a hypothetical internal separation of order q would separate the terminal cliques, and find a corresponding separation by guessing an important separator. For this argument it will be crucial that the separator S of such an internal separation (A, S, B) will be linked into the terminal cliques of size $\geq q$.

Lemma 5.53. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be a valid instance, and (A, S, B) an internal separation of a solution of $\mathcal{I}$ of order $|S| = q$. Then S is linked into any terminal clique of $\mathcal{I}$ of size at least q.*

Proof. Let W_i be a terminal clique of $\mathcal{I}$ of size $|W_i| \geq q$ and suppose S is not linked into W_i. By symmetry suppose $W_i \subseteq A \cup S$, and by definition of internal separation let W_j

be a terminal clique that intersects B. Now, let S' be a minimum size (W_i, S)-separator, in particular having size $|S'| < q$ and S' linked into both W_i and S. Note that S' also separates W_j from W_i, in particular S' gives a separation (A', S', B') so that $W_i \subseteq A' \cup S'$ and $B \subseteq B'$, implying that $W_i \cap A' \neq \emptyset$ and $W_j \cap B' \neq \emptyset$.

Let $(X, \mathcal{T})$ be the solution of $\mathcal{I}$ whose internal separation (A, S, B) is. Note that $\mathcal{T}$ has a bag containing S, and a bag containing W_i. We use the Pulling Lemma (Lemma 5.8) with $(X, \mathcal{T})$, (A', S', B') and the bag containing S to construct a solution of $\mathcal{I} \triangleleft (A', S')$ and then with $(X, \mathcal{T})$, (B', S', A') and the bag containing W_i to construct a solution of $\mathcal{I} \triangleleft (B', S')$. Now, by combining the solutions of $\mathcal{I} \triangleleft (A', S')$ and $\mathcal{I} \triangleleft (B', S')$ using Lemma 5.26 we get a solution of $\mathcal{I}$ whose internal separation (A', S', B') is. This implies that $\mathcal{I}$ is invalid because $|S'| < q$, W_i intersects A', and W_j intersects B'. □

We then introduce notation for arguing about guessing how a hypothetical internal separation of order q separates the terminal cliques. Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance. For $t' \subseteq [t]$, we denote $\hat{W}_{\mathcal{I}}[t'] = \bigcup_{i \in t'} W_i$. Let (t_L, t_R) be a partition of $[t]$ into two non-empty sets, in particular representing a partition of the terminal cliques. We call (t_L, t_R) q-biased if $|W_i| \geq q$ implies that $i \in t_R$.

We then give the main lemma that asserts how internal separations of order q can be guessed by guessing the partition (t_L, t_R) of terminal cliques induced by them and an important $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator.

Lemma 5.54. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be a maximally merged valid instance that has no safe separations and has at least two terminal cliques of size at least q. Suppose that $\mathcal{I}$ has a solution that has an internal separation of order q. Then, there exists a q-biased partition (t_L, t_R) of $[t]$ and an important $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator S of size $|S| = q$ with $\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \neq \emptyset$, corresponding to a separation $(A, S, B) = (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S), S, V(G) \setminus (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \cup S))$ so that both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid instances.*

Proof. Let (A, S, B) be an internal separation of order q of a solution $(X, \mathcal{T})$ of $\mathcal{I}$. Note that because $\mathcal{I}$ does not have safe separations, either all terminal cliques of size $\geq q$ intersect A or all terminal cliques of size $\geq q$ intersect B. By permuting A and B if necessary, assume that all terminal cliques of size $\geq q$ intersect B. Let (t_L, t_R) be the partition of $[t]$ that is obtained by assigning terminal cliques that intersect B into t_R and the others into t_L. Note that at least one terminal clique intersects A because (A, S, B) is an internal separation and at least two terminal cliques of size $\geq q$ intersect B.

Now, S is a $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator. Moreover, S is a minimal $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator, because otherwise the subset of S would give an internal separation of order

$< q$, meaning that $\mathcal{I}$ would not be valid. Let W_i be a terminal clique of size $\geq q$. By Lemma 5.53, S is linked into W_i. Let S' be a smallest important $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator that dominates S. Because S is a minimal $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator, by Lemma 5.19 S' is a $(S, \hat{W}_{\mathcal{I}}[t_R])$-separator, which implies that $|S'| = q$ and S' is linked into W_i. Also, by minimality of S and Lemma 5.18, we have that S' is linked into S. Moreover, as $\hat{W}_{\mathcal{I}}[t_L]$ intersects A, we have that $\hat{W}_{\mathcal{I}}[t_L]$ intersects $\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S')$ and in particular $\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S') \neq \emptyset$.

Now, let $(A', S', B') = (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S'), S', V(G) \setminus (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S') \cup S'))$. Observe that $S \subseteq A' \cup S'$ and $W_i \subseteq B' \cup S'$. We then use the Pulling Lemma (Lemma 5.8) to construct solutions of $\mathcal{I} \triangleleft (A', S')$ and $\mathcal{I} \triangleleft (B', S')$. A solution of $\mathcal{I} \triangleleft (A', S')$ is constructed by applying the lemma with the torso tree decomposition $(X, \mathcal{T})$, the separation (A', S', B'), and the node of $\mathcal{T}$ whose bag contains W_i. Symmetrically, a solution of $\mathcal{I} \triangleleft (B', S')$ is constructed by applying the lemma with the torso tree decomposition $(X, \mathcal{T})$, the separation (B', S', A'), and the node of $\mathcal{T}$ whose bag contains S.

Now, both $\mathcal{I} \triangleleft (A', S')$ and $\mathcal{I} \triangleleft (B', S')$ are yes-instances, so it remains to prove that they are valid. First, because $\mathcal{I}$ is maximally merged and there are at least two terminal cliques of size $\geq q$, it holds that $|\hat{W}_{\mathcal{I}}[t_R]| \geq q + 1$, implying that $\hat{W}_{\mathcal{I}}[t_R]$ intersects B'. Also, as argued before $\hat{W}_{\mathcal{I}}[t_L]$ intersects A'. Therefore, by Lemma 5.48 both $\mathcal{I} \triangleleft (A', S')$ and $\mathcal{I} \triangleleft (B', S')$ are valid. □

5.5.4 The algorithm

We then describe the $2^{\mathcal{O}(k^2)}n^2$ time algorithm for Subset Treewidth.

Given input (G, W, k), the algorithm first uses pre-branching (Lemma 5.47) to enumerate $2^{\mathcal{O}(k^2)}$ instances of Partitioned Subset Treewidth, so that any solution to any of the instances can in $k^{\mathcal{O}(1)}m$ time be turned into a torso tree decomposition in G of width k that covers W, and moreover if such a torso tree decomposition exists, then at least one of the instances is valid. For each resulting instance, we then use a recursive procedure that either concludes that a given instance is invalid, or returns a solution to the instance. This recursive procedure is described in pseudocode Algorithm 2, and we also give a detailed description of it next.

First, on Line 1 the algorithm uses Lemma 5.38 to handle the corner cases of $t = 1$ and $|V(G)| \leq k + 2$. Then, on Lines 2 to 6 the reduction by safe separations is performed. In particular, if there exists a safe separation (A, S, B), then if $|S| < q$ and $\hat{W}$ intersects both A and B, we can by Lemma 5.50 conclude that the instance is invalid. Otherwise,

Algorithm 2 Recursive procedure of a $2^{\mathcal{O}(k^2)}n^2$ time algorithm for Subset Treewidth.

Input: Instance $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$.
Output: Either a solution of $\mathcal{I}$ or INVALID.

1: **if** $t \leq 1$ or $|V(G)| \leq k+2$ **then return** Case-analysis($\mathcal{I}$) ▷ Lemma 5.38
2: **if** Exists a safe separation (A, S, B) **then**
3: **if** $|S| < q$ and $\hat{W}_{\mathcal{I}}$ intersects both A and B **then**
4: **return** INVALID
5: **else**
6: **return** Combine(Solve($\mathcal{I} \triangleleft (A, S)$), Solve($\mathcal{I} \triangleleft (B, S)$))
7: **for all** $i, j \in [t]$ with $i \neq j$ and $|W_i \cup W_j| \leq k+1$ **do**
8: $sol \leftarrow$ Solve($\mathcal{I} \times (W_i, W_j)$)
9: **if** $sol \neq$ INVALID **then return** sol
10: **if** $q > k+1$ **then return** INVALID
11: **if** Less than 2 terminal cliques of size $\geq q$ **then** ▷ Lemma 5.52
12: **for all** $i \in [t]$ so that $|W_i| < q$ and exists $j \neq i$ with $|W_j| \geq |W_i|$ **do**
13: **for all** $w \in W_i$ **do**
14: **for all** Important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separators S in $G \setminus (W_i \setminus \{w\})$ with $|S| \leq k$ **do**
15: **if** $q+1 \leq |W_i \cup S| \leq k+1$ **then**
16: $sol \leftarrow$ Solve($\mathcal{I} + (W_i, S)$)
17: **if** $sol \neq$ INVALID **then return** sol
18: **else** ▷ Lemma 5.54
19: **for all** q-biased bipartitions (t_L, t_R) of $[t]$ **do**
20: **for all** Important $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separators S with $|S| = q$ **do**
21: Let $(A, S, B) = (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S), S, V(G) \setminus (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \cup S))$
22: **if** $\sum_{W_i | W_i \cap A \neq \emptyset} \#\mathsf{tc}_{\mathcal{I}}(W_i) > k+1-q$ **then**
23: $sol \leftarrow$ Combine(Solve($\mathcal{I} \triangleleft (A, S)$), Solve($\mathcal{I} \triangleleft (B, S)$))
24: **if** $sol \neq$ INVALID **then return** sol
25: **return** Solve($(G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q+1)$)

we recursively solve the instances $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ (recursive application of the algorithm is denoted by the function "Solve" in the pseudocode), and if both of them return a solution then we return the solution obtained from combining them, and if either of them return INVALID then we return INVALID. In particular, the function "Combine" on Line 6 denotes an operation that returns INVALID if either of its arguments is INVALID, and if its arguments are a solution $(X_A, \mathcal{T}_A)$ of $\mathcal{I} \triangleleft (A, S)$ and a solution $(X_B, \mathcal{T}_B)$ of $\mathcal{I} \triangleleft (B, S)$ then it returns the solution $(X_A \cup X_B, \mathcal{T}_A \cup_S \mathcal{T}_B)$ of $\mathcal{I}$.

Then, on Lines 7 to 9 the algorithm does terminal clique merging branching. In particular, the algorithm branches on merging all pairs of terminal cliques W_i, W_j with $|W_i \cup W_j| \leq k+1$ and returns a solution if any of the branches returned a solution. After this, by Lemma 5.51 we can assume that $\mathcal{I}$ is either maximally merged or invalid. This is used on Line 10 to justify that if $q > k+1$ we can return INVALID because any solution of a

maximally merged instance with at least two terminal cliques must contain an internal separation.

For the main branching of the algorithm there are two cases. Either there are less than 2 terminal cliques of size $\geq q$, or there are at least 2 terminal cliques of size $\geq q$. We first describe the case when there are less than 2 terminal cliques of size $\geq q$. In this case, on Lines 11 to 17 the algorithm performs leaf pushing branching according to Lemma 5.52. In particular, the algorithm guesses a potential forget-clique W_i that is not a uniquely largest terminal clique, a vertex $w \in W_i$, and an important $(\{w\}, \hat{W} \setminus W_i)$-separator S in the graph $G \setminus (W_i \setminus \{w\})$ so that $q+1 \leq |W_i \cup S| \leq k+1$, and branches on adding S to W_i. For iterating over such important separators, we use the algorithm of Lemma 5.22 to iterate over all important separators of size at most k and check the conditions. The purpose of this branching is to increase the number of terminal cliques of size $\geq q$.

When there are at least 2 terminal cliques of size $\geq q$, the algorithm branches on Lines 18 to 24 on how the internal separation of order q would partition the terminal cliques in the solution, according to Lemma 5.54. The algorithm guesses the q-biased bipartition (t_L, t_R) of $[t]$ and an important $(\hat{W}[t_L], \hat{W}[t_R])$-separator S of size q. Then we denote the separation corresponding to it by $(A, S, B) = (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S), S, V(G) \setminus (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \cup S))$, and if this separation would not be a degenerate internal separation solves the instances $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ recursively and combines the solutions in the same manner as when recursing on safe separations. Again, we use the algorithm of Lemma 5.22 for iterating over such important separators. Finally, on Line 25, if none of the branches returned a solution the algorithm does a recursive call with an increased value of q.

The algorithm can clearly be implemented in space $n^{\mathcal{O}(1)}$, in particular, on Lines 14 and 20 we use the $n^{\mathcal{O}(1)}$ space enumeration of important separators of Lemma 5.22. We then prove the correctness of Algorithm 2. Its running time will be analyzed in Subsection 5.5.5.

First we show that the algorithm is correct when it returns a solution.

Lemma 5.55. *If Algorithm 2 returns a solution, then it is a solution of $\mathcal{I}$.*

Proof. We prove the lemma by induction on the recursion tree. When the algorithm returns on Line 1 from the case analysis, this follows from the correctness of the case analysis.

When breaking the instance by a safe separation (A, S, B) and returning on Line 6 a solution formed by combining solutions of $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$, the correctness follows from induction and Lemma 5.26. For terminal clique merging on Line 9 and

leaf pushing on Line 17 this follows from induction and the fact that any solution of $\mathcal{I} \times (W_i, W_j)$ or $\mathcal{I} + (W_i, S)$ is also solution of $\mathcal{I}$. When combining solutions on Line 24 the correctness again follows Lemma 5.26 and induction. For the final line Line 25 it follows from induction. □

Then we show that the algorithm is correct when it returns that $\mathcal{I}$ is invalid.

Lemma 5.56. *If Algorithm 2 returns* INVALID*, then $\mathcal{I}$ is invalid.*

Proof. We prove the lemma by induction on the recursion tree. When the algorithm returns on Line 1 from the case analysis, this follows from the correctness of the case analysis.

When returning INVALID on Line 4 if a safe separation (A, S, B) with $|S| < q$ and $\hat{W}_{\mathcal{I}}$ intersecting both A and B exists, the correctness is given in Lemma 5.50. For returning INVALID from the safe separation recursion on Line 6, we have that if $\mathcal{I}$ is valid, then by Lemma 5.49 both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid, and therefore the correctness follows from induction.

After terminal clique merging on Lines 7 to 9, by induction and Lemma 5.51 we may assume that $\mathcal{I}$ is either invalid or maximally merged, and in particular in the rest of this proof we may assume that $\mathcal{I}$ is maximally merged, as the conclusion trivially holds when $\mathcal{I}$ is invalid. Then, for returning INVALID on Line 10 if $q > k + 1$, we have that in this case if $t \geq 2$ and $\mathcal{I}$ is maximally merged it must be invalid because then any solution must either contradict that $\mathcal{I}$ is valid or have all of the terminal cliques in a single bag, which would contradict that $\mathcal{I}$ is maximally merged.

For returning INVALID on the final Line 25 there are two cases depending on the number of terminal cliques of size $\geq q$.

First, if there are less than 2 terminal cliques of size $\geq q$, we use Lemma 5.52. Towards contradiction assume that $\mathcal{I}$ is valid but we return INVALID from Line 25. By Lemma 5.34, $\mathcal{I}$ has at least two potential forget-cliques, and some iteration of Line 12 fixed such potential forget-clique W_i, and some iteration of Lines 13 to 15 fixed a vertex $w \in W_i$ and an important $(\{w\}, \hat{W}_{\mathcal{I}} \setminus W_i)$-separator S in $G \setminus (W_i \setminus \{w\})$ satisfying the conditions of Lemma 5.52. Now, by Lemma 5.52, $\mathcal{I} + (W_i, S)$ is a valid instance, so by induction Algorithm 2 would return on Line 17.

Then, if there are at least 2 terminal cliques of size $\geq q$, we use Lemma 5.54. Towards contradiction assume that $\mathcal{I}$ is valid but we return INVALID from Line 25. By induction, $(G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q + 1)$ is invalid, implying that either $\mathcal{I}$ is in-

valid or there exists a solution of $\mathcal{I}$ that contains an internal separation of order q. Therefore, we can assume that $\mathcal{I}$ satisfies the preconditions of Lemma 5.54. Now, let (t_L, t_R) be the q-biased partition of $[t]$ and S the important $(\hat{W}_{\mathcal{I}}[t_L], \hat{W}_{\mathcal{I}}[t_R])$-separator of size $|S| = q$ with $\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \neq \emptyset$ given by Lemma 5.54, and let $(A, S, B) = (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S), S, V(G) \setminus (\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \cup S))$. By Lemma 5.54, both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid. Some iteration of Lines 19 and 20 will fix such (t_L, t_R) and S, and therefore some iteration of Line 21 will fix such (A, S, B). If $\sum_{W_i | W_i \cap A \neq \emptyset} \#\mathsf{tc}_{\mathcal{I}}(W_i) > k + 1 - q$ holds, we would have returned from Line 24 and obtain a contradiction.

It remains to show that if $\mathcal{I}$ is valid, then $\sum_{W_i | W_i \cap A \neq \emptyset} \#\mathsf{tc}_{\mathcal{I}}(W_i) > k + 1 - q$ indeed holds for such (A, S, B). Because $\mathsf{reach}_G(\hat{W}_{\mathcal{I}}[t_L], S) \neq \emptyset$, we have that $\hat{W}_{\mathcal{I}}[t_L]$ intersects A, and because there are at least two terminal cliques of size $\geq q$ and $\mathcal{I}$ is maximally merged, we have that $\hat{W}_{\mathcal{I}}[t_R]$ intersects B. Therefore, because both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ are valid, we can construct from their solutions a solution of $\mathcal{I}$ so that (A, S, B) is its internal separation. However, if $\sum_{W_i | W_i \cap A \neq \emptyset} \#\mathsf{tc}_{\mathcal{I}}(W_i) \leq k + 1 - q$ would hold, then this would be a degenerate internal separation. □

5.5.5 Running time analysis

We analyze the running time of Algorithm 2. Throughout, we let $\delta = k + 2 - k/\log k$, and consider $q \geq \delta$ to be large and $q < \delta$ to be small (recall that log denotes the base-2 logarithm). In order to simplify the analysis we will also assume that $k \geq 64$. If $k < 64$ we use the algorithm of Section 5.4, which runs in $\mathcal{O}(n^2)$ time in this case.

Informally, the idea of choosing $\delta = k + 2 - k/\log k$ is that once $q \geq \delta$, a single leaf push takes a terminal clique to size at least δ, and after that the terminal clique is only $k/\log k$ vertices away from the maximum size, implying that we can "pay" $k^{\mathcal{O}(1)}$ branching degree for increasing the size by one and still end up with $2^{\mathcal{O}(k^2)} n^{\mathcal{O}(1)}$ running time. In the other case, when $q < \delta$, we use the absence of degenerate internal separations to argue that there should be at least $k/\log k$ original terminal vertices behind any internal separation of order q and amortize the cost of the branching on the original terminal vertices. We note that any choice of δ between $k + 2 - k/\log k$ and $k + 2 - \log k$ would be sufficient for the analysis, but we fix the value $\delta = k + 2 - k/\log k$.

We now define the measure of a terminal clique based on cases depending on q and δ. We note that even though we use similar notation to Section 5.4, the measure of this section is unrelated to the measure of Section 5.4. The measure of a terminal clique W_i is

$$\Phi_{\mathcal{I}}(W_i) = \begin{cases} (k+2-\min(q,|W_i|)) \cdot \log k + 4k & \text{if } q \geq \delta \text{ and } |W_i| \geq \delta \\ 6k & \text{if } q \geq \delta \text{ and } |W_i| < \delta \\ (k+2-\min(q,|W_i|)) \cdot \#\mathsf{tc}_{\mathcal{I}}(W_i) + 6k & \text{if } q < \delta. \end{cases}$$

Note that if $q \geq \delta$ and $|W_i| \geq \delta$, then $4k \leq \Phi_{\mathcal{I}}(W_i) \leq 5k$, implying that when $q \geq \delta$, we have $4k \leq \Phi_{\mathcal{I}}(W_i) \leq 6k$, implying $\sum_{i=1}^{t} \Phi_{\mathcal{I}}(W_i) \leq 6kt \leq \mathcal{O}(k^2)$. If $q < \delta$, then we have a lower bound of $6k \leq \Phi_{\mathcal{I}}(W_i)$ on the measure of a single terminal clique, and the sum is upper bounded by $\sum_{i=1}^{t} \Phi_{\mathcal{I}}(W_i) \leq 6kt + |W_O|(k+2) \leq \mathcal{O}(k^2)$.

We then let $\#\mathsf{q}(\mathcal{I}) = \min(2, \text{number of terminal cliques of size } \geq q)$ and define the measure of the instance to be

$$\Phi(\mathcal{I}) = \begin{cases} (k+2-q) \cdot 3k + (2-\#\mathsf{q}(\mathcal{I})) \cdot k + \sum_{i=1}^{t} \Phi_{\mathcal{I}}(W_i) & \text{if } t \geq 2 \text{ and} \\ 1 & \text{if } t = 1. \end{cases}$$

Note that $\Phi(\mathcal{I}) \leq \mathcal{O}(k^2)$ and if $t \geq 2$ then $\Phi(\mathcal{I}) \geq 8k$.

We then give a general lemma for arguing that breaking the instance by separations of order at least q does not increase the measure, and moreover decreases the measure by at least k if this decreases the number of terminal cliques.

Lemma 5.57. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance with $t \geq 2$ terminal cliques and (A, S, B) a separation with $|S| \geq q$. If the number of terminal cliques of $\mathcal{I} \triangleleft (A, S)$ is t, then $\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I})$, and if the number of terminal cliques of $\mathcal{I} \triangleleft (A, S)$ is less than t, then $\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I}) - k$.*

Proof. First, consider the case when the number of terminal cliques of $\mathcal{I} \triangleleft (A, S)$ is t. In this case $\{W_1, \ldots, W_t\} \triangleleft (A, S) = \{W_1, \ldots, W_t\} \cup \{S'\} \setminus \{W_i\}$ for some $S' \supseteq S$ and $W_i \subseteq B \cup S$ or $W_i = S'$. Note that now, $\#\mathsf{tc}_{\mathcal{I} \triangleleft (A,S)}(S') = \#\mathsf{tc}_{\mathcal{I}}(W_i)$ and $\min(q, |W_i|) \leq \min(q, |S'|)$ because $|S'| \geq q$, so $\Phi_{\mathcal{I} \triangleleft (A,S)}(S') \leq \Phi_{\mathcal{I}}(W_i)$, implying together with $\#\mathsf{q}(\mathcal{I} \triangleleft (A, S)) \geq \#\mathsf{q}(\mathcal{I})$ that $\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I})$.

Then, when the number of terminal cliques of $\mathcal{I} \triangleleft (A, S)$ is less than t, first consider the case when $q \geq \delta$. In this case, $\Phi_{\mathcal{I}}(W_i) \geq 4k$ for any W_i and $\Phi_{\mathcal{I} \triangleleft (A,S)}(S') \leq 5k$ when $S' \supseteq S$ because $|S| \geq q$, so we decrease the measure by at least $3k$ from the sum over terminal cliques, and increase by at most k from the $\#\mathsf{q}(\mathcal{I})$ measure as $\#\mathsf{q}(\mathcal{I}) - \#\mathsf{q}(\mathcal{I} \triangleleft (A, S)) \leq 1$, so in total we decrease the measure by at least $2k$.

Then, when $q < \delta$, first consider the case when $\{W_1, \ldots, W_t\}$ contains a terminal clique $S' \supseteq S$ with $S' \subseteq A \cup S$, in which case $\{W_1, \ldots, W_t\} \triangleleft (A, S) \subset \{W_1, \ldots, W_t\}$. In this case we have that

$$\#\mathsf{tc}_{\mathcal{I}\triangleleft(A,S)}(S') = \#\mathsf{tc}_{\mathcal{I}}(S') + \sum_{W_i \in \{W_1,\ldots,W_t\} \setminus \{W_1,\ldots,W_t\}\triangleleft(A,S)} \#\mathsf{tc}_{\mathcal{I}}(W_i).$$

Therefore, as $\min(|W_i|, q) \leq \min(|S'|, q)$, we have that

$$\Phi_{\mathcal{I}\triangleleft(A,S)}(S') \leq \Phi_{\mathcal{I}}(S') + \sum_{W_i \in \{W_1,\ldots,W_t\} \setminus \{W_1,\ldots,W_t\}\triangleleft(A,S)} \Phi_{\mathcal{I}}(W_i) - 6k,$$

implying

$$\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I}) - 6k \cdot |\{W_1, \ldots, W_t\} \setminus \{W_1, \ldots, W_t\} \triangleleft (A, S)| \leq \Phi(\mathcal{I}) - 6k.$$

In the final case $\{W_1, \ldots, W_t\}$ does not contain a terminal clique $S' \supseteq S$ with $S' \subseteq A \cup S$, so $\{W_1, \ldots, W_t\} \triangleleft (A, S) = \{W_1, \ldots, W_t\} \cup \{S\} \setminus \{W_i \in \{W_1, \ldots, W_t\} \mid W_i \subseteq S \cup B\}$ and

$$\#\mathsf{tc}_{\mathcal{I}\triangleleft(A,S)}(S) = \sum_{W_i \in \{W_1,\ldots,W_t\} | W_i \subseteq S \cup B} \#\mathsf{tc}_{\mathcal{I}}(W_i).$$

By $\min(|W_i|, q) \leq \min(|S|, q)$ this implies that

$$\Phi_{\mathcal{I}\triangleleft(A,S)}(S) \leq 6k + \sum_{W_i \in \{W_1,\ldots,W_t\} | W_i \subseteq S \cup B} \Phi_{\mathcal{I}}(W_i) - 6k,$$

which by $|\{W_i \in \{W_1, \ldots, W_t\} \mid W_i \subseteq S \cup B\}| \geq 2$ and $\#\mathsf{q}(\mathcal{I}) - \#\mathsf{q}(\mathcal{I} \triangleleft (A, S)) \leq 1$ implies that $\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I}) - 5k$. □

We then show that breaking the instance by a safe separation on Line 6, i.e., when the safe separation (A, S, B) has either order $\geq q$ or $\hat{W}_{\mathcal{I}}$ intersects only one of A and B, does not increase the measure.

Lemma 5.58. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance with $t \geq 2$ and (A, S, B) a safe separation with $|S| \geq q$ or $\hat{W}_{\mathcal{I}} \subseteq A \cup S$ or $\hat{W}_{\mathcal{I}} \subseteq B \cup S$. Then $\Phi(\mathcal{I}\triangleleft(A, S)) \leq \Phi(\mathcal{I})$.*

Proof. First, if $\hat{W}_{\mathcal{I}} \subseteq B \cup S$, then $\mathcal{I}\triangleleft(A, S)$ has only one terminal clique so $\Phi(\mathcal{I}\triangleleft(A, S)) = 1$. Then, if $\hat{W}_{\mathcal{I}} \subseteq A \cup S$, there is a terminal clique $S' \supseteq S$ with $S' \subseteq A \cup S$ because (A, S, B) is a safe separator, and therefore we have that $\{W_1, \ldots, W_t\} \triangleleft (A, S) \subseteq \{W_1, \ldots, W_t\}$, and moreover all terminal cliques $W_i \in \{W_1, \ldots, W_t\} \setminus \{W_1, \ldots, W_t\} \triangleleft (A, S)$ are subsets of S and thus have size $|W_i| \leq |S| \leq |S'|$ and the original terminal vertices mapped

to them get mapped to S' in $\mathcal{I} \triangleleft (A, S)$, implying that $\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I})$. Then, if $|S| \geq q$, this follows from Lemma 5.57. □

We then show that terminal clique merging decreases the measure by at least k.

Lemma 5.59. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance with $t \geq 2$. It holds that $\Phi(\mathcal{I} \times (W_i, W_j)) \leq \Phi(\mathcal{I}) - k$.*

Proof. Observe that

$$\Phi(\mathcal{I} \times (W_i, W_j)) \leq \Phi(\mathcal{I}) + \Phi_{\mathcal{I}\times(W_i,W_j)}(W_i \cup W_j) - \Phi_{\mathcal{I}}(W_i) - \Phi_{\mathcal{I}}(W_j) + k$$

(where the $+k$ comes from the fact that $\#\mathsf{q}(\mathcal{I} \times (W_i, W_j)) = \#\mathsf{q}(\mathcal{I}) - 1$ might hold). Therefore, $\Phi(\mathcal{I} \times (W_i, W_j)) \leq \Phi(\mathcal{I}) - k$ holds when $q \geq \delta$ because in that case $\Phi_{\mathcal{I}\times(W_i,W_j)}(W_i \cup W_j) \leq 6k$ and $\Phi_{\mathcal{I}}(W_i) + \Phi_{\mathcal{I}}(W_j) \geq 8k$.

When, $q < \delta$, first if $W_i \cup W_j \in \{W_1, \ldots, W_t\}$, we just map more original terminal vertices into a larger terminal clique and decrease the measure by at least $12k$. In the other case, we have that $\#\mathsf{tc}_{\mathcal{I}\times(W_i,W_j)}(W_i \cup W_j) = \#\mathsf{tc}_{\mathcal{I}}(W_i) + \#\mathsf{tc}_{\mathcal{I}}(W_j)$, which by $|W_i \cup W_j| \geq \max(|W_i|, |W_j|)$ implies that $\Phi_{\mathcal{I}}(W_i) + \Phi_{\mathcal{I}}(W_j) - \Phi_{\mathcal{I}\times(W_i,W_j)}(W_i \cup W_j) \geq 6k$, implying $\Phi(\mathcal{I} \times (W_i, W_j)) \leq \Phi(\mathcal{I}) - 5k$. □

We then show that increasing the value of q decreases the measure by at least k.

Lemma 5.60. *Let $\mathcal{I}_1 = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ and $\mathcal{I}_2 = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q+1)$ and $t \geq 2$. It holds that $\Phi(\mathcal{I}_2) \leq \Phi(\mathcal{I}_1) - k$.*

Proof. Observe that the measures of each terminal cliques do not decrease, in particular, if $q + 1 \geq \delta$ and $q < \delta$, then the measure goes from at least $6k$ to at most $6k$. Then, the measure of the instance decreases $3k$ from the term $(k + 2 - q) \cdot 3k$ and increases by at most $2k$ from the term $(2 - \#\mathsf{q}(\mathcal{I})) \cdot k$. □

We then show that if the instance has less than 2 terminal cliques of size $\geq q$, then increasing the size of a terminal clique from less than q to at least q decreases the measure by at least k.

Lemma 5.61. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance with $t \geq 2$, W_i a terminal clique of $\mathcal{I}$, and $S \subseteq V(G)$. If $\#\mathsf{q}(\mathcal{I}) < 2$, $|W_i| < q$, and $|W_i \cup S| \geq q$, then $\Phi(\mathcal{I} + (W_i, S)) \leq \Phi(\mathcal{I}) - k$.*

Proof. If $W_i \cup S \in \{W_1, \ldots, W_t\}$, then $\mathcal{I} + (W_i, S) = \mathcal{I} \times (W_i, W_i \cup S)$ and this holds by Lemma 5.59. Otherwise, observe that increasing the size of a terminal clique (while keeping the mapping tc same) cannot decrease its measure, and therefore as $\#\mathsf{q}(\mathcal{I} + (W_i, S)) \geq \#\mathsf{q}(\mathcal{I}) + 1$, it holds that $\Phi(\mathcal{I} + (W_i, S)) \leq \Phi(\mathcal{I}) - k$. □

We then argue how the measure behaves when we break the instance by a separation (A, S, B) of order q and $\mathcal{I} \triangleleft (B, S)$ has the same number of terminal cliques as $\mathcal{I}$. In particular, this corresponds to Line 23 of Algorithm 2 when $|t_L| = 1$. This lemma is the main motivation of the somewhat involved definition of the measure and δ.

Lemma 5.62. *Let $\mathcal{I} = (G, \{W_1, \ldots, W_t\}, k, \mathsf{tc}, q)$ be an instance with $t \geq 2$. Let (A, S, B) be a separation so that $|S| \geq q$ and there is a terminal clique W_i with $W_i \subseteq A \cup S$, $|W_i| < q$, and $\#\mathsf{tc}_{\mathcal{I}}(W_i) > k + 1 - q$. It holds that $\Phi(\mathcal{I} \triangleleft (B, S)) \leq \Phi(\mathcal{I}) - \min(k, (q - |W_i|) \log k)$.*

Proof. First, if $\mathcal{I} \triangleleft (B, S)$ has less terminal cliques than $\mathcal{I}$, then $\Phi(\mathcal{I} \triangleleft (B, S)) \leq \Phi(\mathcal{I}) - k$ by Lemma 5.57. Then, we assume that W_i is the only terminal clique that is a subset of $A \cup S$ and $\mathcal{I} \triangleleft (B, S)$ has the same number of terminal cliques as $\mathcal{I}$. In this case, because $\#\mathsf{q}(\mathcal{I} \triangleleft (B, S)) \geq \#\mathsf{q}(\mathcal{I})$, we have $\Phi(\mathcal{I} \triangleleft (B, S)) \leq \Phi(\mathcal{I}) + \Phi_{\mathcal{I} \triangleleft (B,S)}(S) - \Phi_{\mathcal{I}}(W_i)$. We also have that $\#\mathsf{tc}_{\mathcal{I} \triangleleft (B,S)}(S) = \#\mathsf{tc}_{\mathcal{I}}(W_i)$. We consider the cases $q < \delta$ and $q \geq \delta$.

First, when $q < \delta$

$$\#\mathsf{tc}_{\mathcal{I}}(W_i) > k + 1 - \delta > k + 1 - (k + 2 - k/\log k) > k/\log k - 1 \geq \log k,$$

where the last inequality follows from $k \geq 64$. This implies that

$$\begin{aligned}
\Phi(\mathcal{I} \triangleleft (B, S)) \leq & \Phi(\mathcal{I}) + (k + 2 - q) \cdot \#\mathsf{tc}_{\mathcal{I} \triangleleft (B,S)}(S) - (k + 2 - |W_i|) \cdot \#\mathsf{tc}_{\mathcal{I}}(W_i) \\
\leq & \Phi(\mathcal{I}) - (q - |W_i|) \cdot \#\mathsf{tc}_{\mathcal{I}}(W_i) \\
\leq & \Phi(\mathcal{I}) - (q - |W_i|) \cdot \log k.
\end{aligned}$$

Then, consider the case when $q \geq \delta$. If $|W_i| < \delta$, then $\Phi_{\mathcal{I} \triangleleft (B,S)}(S) \leq 5k$ and $\Phi_{\mathcal{I}}(W_i) = 6k$, implying that $\Phi(\mathcal{I} \triangleleft (B, S)) \leq \Phi(\mathcal{I}) - k$. Then, if $|W_i| \geq \delta$,

$$\begin{aligned}
\Phi(\mathcal{I} \triangleleft (B, S)) &\leq \Phi(\mathcal{I}) + (k + 2 - q) \cdot \log k - (k + 2 - |W_i|) \cdot \log k \\
&\leq \Phi(\mathcal{I}) - (q - |W_i|) \cdot \log k.
\end{aligned}$$

□

We then put the lemmas together to prove the running time of Algorithm 2.

Lemma 5.63. *Algorithm 2 runs in time* $2^{\mathcal{O}(k^2)}n^2$.

Proof. First we observe that all of the operations in a single call of the recursive procedure can be performed in $2^{\mathcal{O}(k)}m'$ time, where m' is the number of edges in the instance given to the recursive call. In particular, the case analysis of Line 1 can be implemented in $\mathcal{O}(m')$ time by Lemma 5.38, safe separations can be found in $k^{\mathcal{O}(1)}m'$ time by Lemma 5.31, the terminal clique merging of Lines 7 to 9 can be implemented in $k^{\mathcal{O}(1)}$ time, the branching on Lines 11 to 17 when there are less than 2 terminal cliques of size $\geq q$ can be implemented in $k^{\mathcal{O}(1)}4^k m' = 2^{\mathcal{O}(k)}m'$ time by Lemma 5.22, and also the branching on Lines 18 to 24 when there are at least 2 terminal cliques of size $\geq q$ can be implemented in $k^{\mathcal{O}(1)}2^t 4^k m' = 2^{\mathcal{O}(k)}m'$ time.

By the definition of $\mathcal{I} \triangleleft (A, S)$, observe that at each recursive call the current graph can be obtained from an induced subgraph of the original graph by adding all edges inside the terminal cliques, and therefore we can bound $m' \leq k^{\mathcal{O}(1)}m \leq k^{\mathcal{O}(1)}n$, where m is the number of original edges. Therefore, the running time of the algorithm can be bounded by $2^{\mathcal{O}(k)}n \cdot R(\mathcal{I})$, where $R(\mathcal{I})$ is the total number of recursive calls.

We show by induction that the number of recursive calls is bounded by

$$R(\mathcal{I}) \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})} \leq 2^{\mathcal{O}(k^2)}n$$

(where $\mathsf{size}(\mathcal{I})$ is defined in Subsection 5.4.6), which then implies the conclusion because $2^{\mathcal{O}(k)}n \cdot 2^{\mathcal{O}(k^2)}n = 2^{\mathcal{O}(k^2)}n^2$.

First, when the algorithm returns from the case analysis of Line 1, this holds because $\mathsf{size}(\mathcal{I}) \geq 1$ and $\Phi(\mathcal{I}) \geq 1$. Then we can assume that $t \geq 2$ and $|V(G)| \geq k + 3$. If there exists a safe separation (A, S, B), then the number of recursive calls is

$$\begin{aligned} R(\mathcal{I}) =& 1 + R(\mathcal{I} \triangleleft (A, S)) + R(\mathcal{I} \triangleleft (B, S)) \\ \leq& 1 + (\mathsf{size}(\mathcal{I} \triangleleft (A, S)) + \mathsf{size}(\mathcal{I} \triangleleft (B, S))) \cdot 16^{\Phi(\mathcal{I})} && \text{by Lemma 5.58 and induction} \\ \leq& \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}. && \text{by Lemma 5.44} \end{aligned}$$

Now, let $R_1(\mathcal{I})$ denote the total number of calls in the recursion trees from terminal clique merging on Line 8. By Lemma 5.59, induction, and the fact that $k \geq 64$, we have that

$$R_1(\mathcal{I}) \leq (k+2)^2 \cdot \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})-k} \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/5.$$

Then, let $R_2(\mathcal{I})$ denote the total number of calls in the recursion tree from the final Line 25 where q is incremented. By induction and Lemma 5.60, we have that

$$R_2(\mathcal{I}) \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})-k} \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/5.$$

Now, consider the case when there are less than two terminal cliques of size $\geq q$, and let $R_3(\mathcal{I})$ denote the total number of calls in the recursion tree from Line 16 where the leaf pushing branching is done. By induction, Lemma 5.22, Lemma 5.61, and $k \geq 64$, we have that

$$R_3(\mathcal{I}) \leq (k+2)^2 \cdot 4^k \cdot \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})-k} \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/5.$$

This finishes the running time analysis in the case when there are less than 2 terminal cliques of size $\geq q$, as in this case we have that

$$R(\mathcal{I}) \leq 1 + R_1(\mathcal{I}) + R_2(\mathcal{I}) + R_3(\mathcal{I}) \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}.$$

It remains to consider the case when there are at least two terminal cliques of size $\geq q$. Let $R_4(\mathcal{I})$ denote the total number of calls in the recursion tree from Line 23 when $|t_L| \geq 2$ and $R_5(\mathcal{I})$ the total number of calls when $|t_L| = 1$. Recall that in all cases $|t_R| \geq 2$.

First, let $|t_L| \geq 2$ and consider a single call from Line 23. As $\hat{W}_{\mathcal{I}}[t_L] \subseteq A \cup S$ and $\hat{W}_{\mathcal{I}}[t_R] \subseteq B \cup S$, we have that both $\mathcal{I} \triangleleft (A, S)$ and $\mathcal{I} \triangleleft (B, S)$ have less terminal cliques than $\mathcal{I}$, and therefore by induction, Lemma 5.57, and Lemma 5.44 the number of calls for fixed (A, S, B) is at most

$$\mathsf{size}(\mathcal{I}) \cdot (16^{\Phi(\mathcal{I} \triangleleft (A,S))} + 16^{\Phi(\mathcal{I} \triangleleft (B,S))}) \leq 2 \cdot \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})-k}.$$

Then, the total number of calls from Line 23 in this case is

$$R_4(\mathcal{I}) \leq 2^t \cdot 4^k \cdot 2 \cdot \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})-k} \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/5.$$

Now, let $|t_L| = 1$ and consider a single call from Line 23. As $|t_R| \geq 2$, we again have by Lemma 5.57 that $\Phi(\mathcal{I} \triangleleft (A, S)) \leq \Phi(\mathcal{I}) - k$. Let W_i be the single terminal clique with $i \in t_L$. We have by Lemma 5.62 that $\Phi(\mathcal{I} \triangleleft (B, S)) \leq \Phi(\mathcal{I}) - \min(k, (q - |W_i|) \log k)$.

Because $\mathcal{I}$ has no safe separations, $|W_i| < q$, and there is a terminal clique of size $\geq q$, we have that $\mathsf{flow}_G(W_i, \overline{W}_{\mathcal{I}}(W_i)) = |W_i|$, which implies by Lemma 5.24 that the number of important $(W_i, \overline{W}_{\mathcal{I}}(W_i))$-separators of size q is at most $k^{q-|W_i|}$. Combining

with Lemma 5.22 and the fact that $q \leq k$ here, we actually obtain an upper bound of

$$\min(k^{q-|W_i|}, 4^k) \leq 4^{\min(k,(q-|W_i|)\log k)}.$$

Now, for fixed W_i, by induction and Lemma 5.44 the number of recursive calls is

$$\begin{aligned}
R_5(\mathcal{I}, i) &\leq 4^{\min(k,(q-|W_i|)\log k)} \cdot \mathsf{size}(\mathcal{I}) \cdot (16^{\Phi(\mathcal{I})-\min(k,(q-|W_i|)\log k)} + 16^{\Phi(\mathcal{I})-k}) \\
&\leq \mathsf{size}(\mathcal{I}) \cdot 4^{\min(k,(q-|W_i|)\log k)} \cdot 2 \cdot 16^{\Phi(\mathcal{I})} \cdot 16^{-\min(k,(q-|W_i|)\log k)} \\
&\leq \mathsf{size}(\mathcal{I}) \cdot 2 \cdot 16^{\Phi(\mathcal{I})} \cdot (1/4)^{\min(k,(q-|W_i|)\log k)} \\
&\leq \mathsf{size}(\mathcal{I}) \cdot 2 \cdot 16^{\Phi(\mathcal{I})}/20k \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/10k.
\end{aligned}$$

Then, over all terminal cliques this is

$$R_5(\mathcal{I}) = \sum_{i=1}^{t} R_5(\mathcal{I}, i) \leq (k+2) \cdot \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/10k \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}/5.$$

This finishes the running time analysis of the case when there are at least 2 terminal cliques of size $\geq q$, as in this case we have that

$$R(\mathcal{I}) \leq 1 + R_1(\mathcal{I}) + R_2(\mathcal{I}) + R_4(\mathcal{I}) + R_5(\mathcal{I}) \leq \mathsf{size}(\mathcal{I}) \cdot 16^{\Phi(\mathcal{I})}.$$

□

Putting together with the pre-branching of Lemma 5.47, this finishes the proof of Theorem 5.3, and together with Theorem 5.2 they imply Theorem 1.2.

Chapter 6

Dynamic treewidth

In this chapter we give a data structure for maintaining tree decompositions of bounded width. In particular, we prove the following theorem.

Theorem 1.4. *There is a data structure that is initialized with an initially edgeless n-vertex dynamic graph G and a parameter k. The data structure supports updating G by edge insertions and deletions, and maintains a tree decomposition of G of width at most $6k+5$ whenever the treewidth of G is at most k. When the treewidth of G is more than k, the data structure contains a marker* "Treewidth too large"*. The amortized initialization time is $2^{k^{\mathcal{O}(1)}}n$ and the amortized update time is $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$.*

Moreover, the data structure can be provided a CMSO_2 *sentence φ upon initialization, in which case it maintains whether φ is true in G whenever the marker* "Treewidth too large" *is not present. In this case, the amortized initialization time is $f(k,\varphi)\cdot n$ and the amortized update time is $f(k,\varphi)\cdot 2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$, where f is a computable function.*

The proof of Theorem 1.4 builds on the Subset Treewidth problem and torso tree decompositions introduced in the previous chapter, but also uses several other techniques from the literature of treewidth computing.

6.1 Overview

In this section we give an informal overview of the proof of Theorem 1.4. We first give a high-level description of the data structure in Subsection 6.1.1, and then in Subsections 6.1.2 and 6.1.3 we sketch the proofs of the most important technical ingredients. At the end of this section, we overview the organization of the rest of this chapter.

We remark that some statements made in this section are simplified compared to the their formal versions presented later, and thus may not be formally true. However, their intent is to be "morally correct" in the sense that analogous, but more technical, statements are proved in the later sections. Lemma statements that are not marked as "informal" are formally correct.

6.1.1 High-level description

Let n be the number of vertices and k the given parameter that bounds the treewidth of the dynamic graph G. Our goal is to maintain a binary tree decomposition $\mathcal{T}$ of height at most $h = 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$ and width at most $6k+5$, and at the same time any dynamic programming scheme, or more formally, a "tree decomposition automaton", on $\mathcal{T}$. In this overview, we assume that the treewidth of G is always at most k. Relaxing this assumption to the setting of Theorem 1.4 is relatively straightforward with the technique of "delaying invariant-breaking insertions" of Eppstein et al. [1996].

The goal of maintaining such a tree decomposition is reasonable because of a lemma of Bodlaender and Hagerup [1998] that states that any tree decomposition of width k can be turned into a binary tree decomposition of height $\mathcal{O}(\log n)$ and width at most $3k+2$.

Now, assuming $\mathcal{T}$ is a binary tree decomposition of height $\mathsf{hgt}(\mathcal{T}) \leq h$ and width $\mathcal{O}(k)$, the operations of adding an edge or deleting an edge can be implemented in time $f(k) \cdot \mathsf{hgt}(\mathcal{T})$, for some function f depending on the dynamic programming we are maintaining, as follows. Let us assume that we store the existence of an edge uv at the highest node whose bag contains both u and v, i.e., at the node $\mathsf{forget}_{\mathcal{T}}(uv)$. Now, when deleting the edge uv, it suffices to find the node $\mathsf{forget}_{\mathcal{T}}(uv)$, update information about the existence of this edge stored in this node, and then update dynamic programming tables of the nodes on the path from this node to the root, taking $f(k) \cdot \mathsf{hgt}(\mathcal{T})$ time.

In the edge addition operation between vertices u and v, we let P_u and P_v be the paths in $\mathcal{T}$ from $\mathsf{forget}_{\mathcal{T}}(u)$ and $\mathsf{forget}_{\mathcal{T}}(v)$, respectively, to the root, add u and v to all bags on $P_u \cup P_v$, add the information about the existence of the edge uv to the root node, and update dynamic programming tables on $P_u \cup P_v$, again taking in total $f(k) \cdot \mathsf{hgt}(\mathcal{T})$ time. Let us emphasize that only the node $\mathsf{forget}_{\mathcal{T}}(uv)$ is "aware" of the existence of the edge uv, as opposed to the more intuitive alternative of all the bags containing both u and v being "aware" of uv. This is crucial for the fact that the dynamic programming tables of only $\mathcal{O}(\mathsf{hgt}(\mathcal{T}))$ nodes have to be recomputed after an edge addition or deletion.

Now, the only issue is that the edge addition operation could cause the width of $\mathcal{T}$ to increase to more than $6k+5$. In this case we have to modify $\mathcal{T}$ in order to make its width smaller, while still maintaining small height. The main technical contribution of this chapter is to show that such changes to tree decompositions can indeed be implemented efficiently.

Recall that in the edge addition operation, we increased the sizes of bags in a subtree consisting of the union $P_u \cup P_v$ of two paths, each between a node and the root. In particular, all of the nodes with too large bags are contained in the prefix $P_u \cup P_v$ of $\mathcal{T}$ of size at most $2 \cdot \mathsf{hgt}(\mathcal{T})$ (recall that a prefix of a rooted tree is a connected set of nodes that contains the root). Now, our idea is to generalize the improvement operation of Chapter 5, so that instead of improving only a single bag, it can be given a prefix of $\mathcal{T}$ and it improves the whole prefix. Roughly speaking, we will solve the Subset Treewidth problem with $W = \mathsf{bags}(P_u \cup P_v)$, and then use the torso tree decomposition to improve $\mathcal{T}$ as in Chapter 5. Of course, several additional techniques will be needed for bounding the height, and for implementing the operation in time roughly linear in $|T_{\mathsf{pref}}|$. However, compared to Chapter 5 we have the advantage of being allowed a worse dependence on k, and being required to maintain only an approximately optimal tree decomposition.

This generalization of the improvement operation that we develop for maintaining tree decompositions in the dynamic setting will be called the *refinement operation*. The definition and properties of the operation are technical and will be described in Subsection 6.1.2, but let us give here an informal description of what is achieved by the operation. The refinement operation takes as an input a prefix T_{pref} of the tree decomposition $\mathcal{T}$ that we are maintaining, and informally stated, replaces T_{pref} by a tree decomposition of width at most $6k+5$ and height at most $\mathcal{O}(\log n)$. The operation also edits other parts of $\mathcal{T}$, but in a way that makes them only better in terms of sizes of bags, in a similar fashion as the amortization in Chapter 4. In particular, if we use the refinement operation on $T_{\mathsf{pref}} = P_u \cup P_v$ after an edge addition operation that made the width exceed $6k+5$ in the nodes in $P_u \cup P_v$, the operation brings the width of $\mathcal{T}$ back to at most $6k+5$. The amortized running time of the refinement operation is $2^{k^{\mathcal{O}(1)}}|T_{\mathsf{pref}}| \log n$ (times factors depending on the dynamic programming maintained), and it can increase the height of $\mathcal{T}$ by at most $\mathcal{O}(\log n)$.

With the refinement operation, we have a tool for keeping the width of the maintained tree decomposition $\mathcal{T}$ bounded by $6k+5$. However, each application of the refinement operation can increase the height of $\mathcal{T}$ by $\Omega(\log n)$, so we need also a tool for decreasing the height. We develop such a tool by a combination of a carefully chosen potential function and a strategy to decrease the potential function "for free" by using the refinement

operation if the height is too large. In particular, the potential function we use is

$$\Phi(\mathcal{T}) = \sum_{t \in V(T)} (\gamma \cdot k)^{|\mathsf{bag}(t)|} \cdot \mathsf{hgt}(t),$$

where γ is a fixed constant that we define in Subsection 6.5.1. This function has the properties that it does not increase too much in the edge addition operation (the increase is at most $k^{\mathcal{O}(k)}\mathsf{hgt}(\mathcal{T})^2$), it plays well together with the details of the amortized analysis of the refinement operation (the factor $(\gamma \cdot k)^{|\mathsf{bag}(t)|}$ comes from there in a similar fashion as we had $3^{|\mathsf{bag}(t)|}$ in the potential in Chapter 4), and because of the factor $\mathsf{hgt}(t)$, it naturally admits smaller values on trees of smaller height.

In Subsection 6.1.3 we outline a strategy that, provided the height of $\mathcal{T}$ exceeds $2^{\Omega(k \log k \sqrt{\log n \log\log n})}$, selects a prefix T_{pref} so that applying the refinement operation on T_{pref} decreases the value of $\Phi(\mathcal{T})$, and moreover the running time of the refinement operation can be bounded by this decrease. In particular, this means that as long as the height is more than $2^{\Omega(k \log k \sqrt{\log n \log\log n})}$, we can apply such a refinement operation "for free" in terms of amortized running time, and moreover decrease the value of the potential. As the potential cannot keep decreasing forever, repeated applications of such an operation eventually lead to improving the height to at most $2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$.

6.1.2 The refinement operation

We then overview how the refinement operation works (see Figure 6.1 for an illustration of it). First, given the prefix T_{pref} that we wish to improve, we find a set of vertices $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$ so that $\mathsf{torso}_G(X)$ has treewidth at most $2k+1$ (here we have $2k+1$ instead of k for a technical reason we will explain), i.e., we solve a variant of Subset Treewidth. Then, we compute an optimum-width tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$ and use the lemma of Bodlaender and Hagerup [1998] to make it a binary tree decomposition of height $\mathcal{O}(\log n)$, resulting in $\mathcal{T}^X$ having width at most $6k+5$. Now $\mathcal{T}^X$ will form a prefix of the new refined tree decomposition. It remains to construct tree decompositions $\mathcal{T}^C$ for each connected component C of $G \setminus X$ and attach them to $\mathcal{T}^X$.

Recall that a node of T is an appendix of T_{pref} if it is not in T_{pref} but its parent is. For each appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$ of T_{pref}, let $\mathcal{T}_a = (T_a, \mathsf{bag}_a) = \mathcal{T}\restriction_{\mathsf{desc}(a)}$ be the restriction of $\mathcal{T}$ to the subtree rooted at a. Note that because of $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$, for each connected component C of $G \setminus X$ there exists a unique appendix a of T_{pref} such that all bags containing vertices from C are in $\mathcal{T}_a$. Moreover, the restriction $(T_a, \mathsf{bag}_a\restriction_{N[C]})$ to the closed neighborhood $N[C]$ of C is a tree decomposition of the induced subgraph $G[N[C]]$ minus the edges inside $N(C)$.

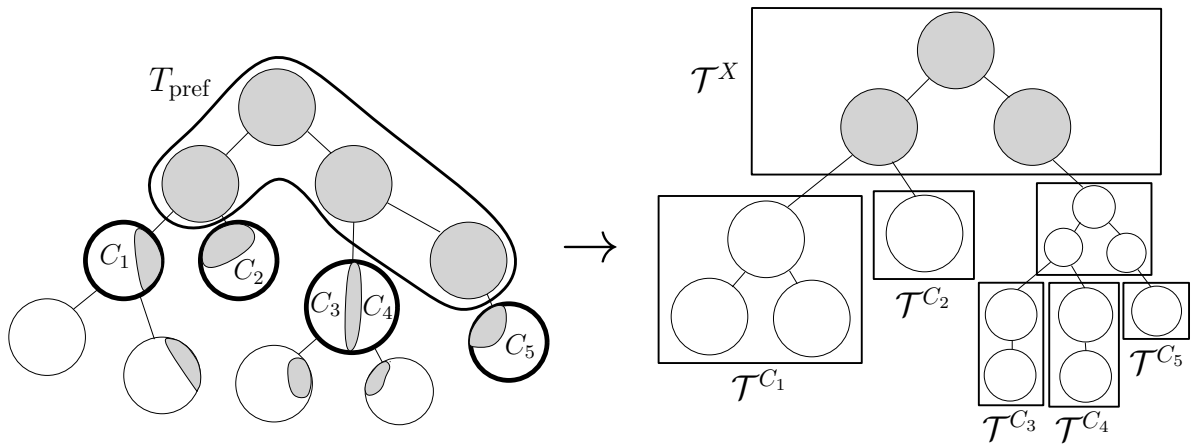

Figure 6.1: The refinement operation. The left picture illustrates the tree decomposition $\mathcal{T}$, with the prefix T_{pref} encircled and the vertices in $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$ depicted in gray. The appendices of T_{pref} are circled by boldface, and the components of $G \setminus X$ are denoted by $C_1, \ldots, C_5$. The right picture illustrates the tree decomposition constructed from $\mathcal{T}$ by the refinement using X, in particular, by taking the tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$, and gluing the tree decompositions $\mathcal{T}^{C_i}$ for components C_i of $G \setminus X$ to it. The subtree consisting of the three nodes above $\mathcal{T}^{C_3}$, $\mathcal{T}^{C_4}$, $\mathcal{T}^{C_5}$ is constructed in order to keep the tree binary after reattaching $\mathcal{T}^{C_3}$, $\mathcal{T}^{C_4}$, $\mathcal{T}^{C_5}$.

Because $\mathcal{T}^X$ is a tree decomposition of $\mathsf{torso}_G(X)$, it has a bag that contains $N(C)$. We wish to attach $(T_a, \mathsf{bag}_a{\restriction_{N[C]}})$ from its root a under this bag. In order to achieve this while satisfying the connectedness condition of tree decompositions, we need to have the set $N(C)$ in the root of $(T_a, \mathsf{bag}_a{\restriction_{N[C]}})$. We denote by $\mathcal{T}^C = (T^C, \mathsf{bag}^C)$ the tree decomposition obtained from $(T_a, \mathsf{bag}_a{\restriction_{N[C]}})$ by "forcing" $N(C)$ to be in the root bag $\mathsf{bag}^C(a)$, in particular, by inserting $N(C)$ to $\mathsf{bag}^C(a)$ and then fixing the connectedness condition by inserting each vertex $v \in N(C)$ to all bags on the unique path from a to $\mathsf{forget}_{\mathcal{T}_a}(v)$. Then, $\mathcal{T}^C$ is a tree decomposition of $G[N[C]]$ whose root bag contains $N(C)$, and therefore it can be attached to the bag of $\mathcal{T}^X$ that contains $N(C)$. These attachments may make the resulting tree decomposition non-binary, so finally we need to expand high-degree nodes into binary trees. This concludes the informal description of the refinement operation. The actual definition is a bit more involved, as it will be necessary to (1) select X in a particular way, (2) treat in some cases multiple different components C in $\mathcal{T}_a$ as if they were one component, and (3) prune out unnecessary bags of $\mathcal{T}^C$ in a similar fashion as in the amortization in Chapter 4.

From the description of the refinement operation sketched above it should be clear that the resulting tree decomposition is indeed a tree decomposition of G. Also, because the height of $\mathcal{T}^X$ is at most $\mathcal{O}(\log n)$ and the height of each attached decomposition $\mathcal{T}^C$ is at most $\mathsf{hgt}(\mathcal{T})$, the height of the resulting tree decomposition is at most $\mathsf{hgt}(\mathcal{T}) + \mathcal{O}(\log n)$. Recall that our goal is that if all bags of size more than $6k + 6$ are contained in T_{pref},

then the width of the refined tree decomposition is at most $6k + 5$. The width of $\mathcal{T}^X$ is clearly at most $6k + 5$. To bound the sizes of the bags of $\mathcal{T}^C$ after the insertions of vertices in $N(C)$, we have to argue similarly as in Chapters 4 and 5, and for this we need to assert extra properties of X.

Let us call a set of vertices $X \subseteq V(G)$ a *k-closure* of T_{pref} if $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$ and the treewidth of $\mathsf{torso}_G(X)$ is at most $2k + 1$. In particular, the set X in the refinement operation is a k-closure of T_{pref}. We say that a k-closure X is *linked* into T_{pref} if for each component C of $G \setminus X$, the set $N(C)$ is linked into $\mathsf{bags}(T_{\mathsf{pref}})$. The key property for controlling the width of $\mathcal{T}^C$ is that if X is linked into T_{pref}, then each bag of $\mathcal{T}^C$ has size at most the size of the corresponding bag in $\mathcal{T}$. In particular, let $\mathcal{T}_a = (T_a, \mathsf{bag}_a)$ be a subtree of $\mathcal{T}$ rooted at an appendix a of T_{pref}, and C a component of $G \setminus X$ contained in the bags in $\mathcal{T}_a$. We can bound the size of $\mathsf{bag}^C(t)$ for any $t \in V(T_a)$ as follows.

Lemma 6.1. *If X is linked into T_{pref}, then $|\mathsf{bag}^C(t)| \leq |\mathsf{bag}_a(t)|$.*

The proof of Lemma 6.1 is similar to the proofs in Section 5.2. It implies that if T_{pref} contains all bags of size more than $6k + 6$ and X is linked into T_{pref}, then the tree decomposition resulting from refinement will have width at most $6k + 5$. We note that the existence of a k-closure of T_{pref} that is linked into T_{pref} is non-trivial, but before going into that let us immediately generalize the notion of linkedness in order to obtain a stronger form of Lemma 6.1. For a set of vertices S, we again denote by $d(S) = \sum_{v \in S} \mathsf{depth}_{\mathcal{T}}(\mathsf{forget}_{\mathcal{T}}(v))$ the sum of the values of the vertex-depth function on vertices in S, and say that a k-closure X is *d-linked* into T_{pref} if it is linked into T_{pref}, and additionally for each neighborhood $N(C)$, there are no separators S with $|S| = |N(C)|$ and $d(S) < d(N(C))$ separating $N(C)$ from $\mathsf{bags}(T_{\mathsf{pref}})$.

Recall that our potential function is $\Phi(\mathcal{T}) = \sum_{t \in V(T)} (\gamma \cdot k)^{|\mathsf{bag}(t)|} \cdot \mathsf{hgt}(t)$. By using d-linkedness similarly as in Section 5.2 and the potential function similarly as in Section 4.3 we can prove that the actual definition of the refinement operation satisfies the following properties.

Lemma 6.2 (Informal)**.** *Let X be a d-linked k-closure of T_{pref}, a an appendix of T_{pref}, and $C_1, \ldots, C_\ell$ the connected components of $G \setminus X$ that are contained in $\mathcal{T}_a$. It holds that $\sum_{i=1}^{\ell} \Phi(\mathcal{T}^{C_i}) \leq \Phi(\mathcal{T}_a)$, and moreover, we can construct the tree decompositions $\mathcal{T}^{C_i}$ for all i in time $f(k) \cdot (\Phi(\mathcal{T}_a) - \sum_{i=1}^{\ell} \Phi(\mathcal{T}^{C_i}))$, together with their updated dynamic programming tables.*

We note that Lemma 6.2 would hold also for a potential function without the $\mathsf{hgt}(t)$ factor. This factor is included in the potential only for the purposes of the height reduction scheme that will be outlined in Subsection 6.1.3. Also, in the actual refinement operation

each C_i in Lemma 6.2 can actually be the union of multiple different components with the same neighborhood $N(C_i)$, and we can actually charge a bit extra from the potential for each of these "connected components". This extra potential will be used for constructing the binary trees for the high-degree attachment points.

After setting technical details aside, the main takeaway of Lemma 6.2 is that constructing the decompositions $\mathcal{T}^C$ is "free" in terms of the potential. The only place where we could use a lot of time or increase the potential a lot is finding the set X and constructing the tree decomposition $\mathcal{T}^X$. For bounding this, we give a lemma asserting that we can assume X to have size at most $k^{\mathcal{O}(1)}|T_{\mathsf{pref}}|$, and in addition to have an even stronger structural property that will be useful in the height reduction scheme. We prove the following statement using the Dealternation Lemma of Bojańczyk and Pilipczuk [2022]. The bound $2k+1$ in the definition of k-closure comes from this proof. Recall the definition that $\mathsf{cmp}(t) = \mathsf{bags}(\mathsf{desc}(t)) \setminus \mathsf{bag}(t)$ for a node t of $\mathcal{T}$.

Lemma 6.3. *Let G be a graph of treewidth at most k, and $\mathcal{T}$ a tree decomposition of G of width $\mathcal{O}(k)$. For any prefix T_{pref} of $\mathcal{T}$, there exists a k-closure X of T_{pref} so that for each appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$ it holds that $|X \cap \mathsf{cmp}(a)| \leq \mathcal{O}(k^4)$.*

As $\mathcal{T}$ is a binary tree, T_{pref} has at most $|T_{\mathsf{pref}}|+1$ appendices, so by Lemma 6.3, T_{pref} admits a k-closure with at most $\mathcal{O}(k^4|T_{\mathsf{pref}}|)$ vertices. By using such a k-closure, we can bound the size of $\mathcal{T}^X$ by $k^{\mathcal{O}(1)}|T_{\mathsf{pref}}|$. As each node in $\mathcal{T}^X$ has potential at most $k^{\mathcal{O}(k)}(\mathsf{hgt}(\mathcal{T})+\log n)$ in the resulting decomposition, we get that the refinement operation increases the total potential by at most $k^{\mathcal{O}(k)}(|T_{\mathsf{pref}}|\cdot(\mathsf{hgt}(\mathcal{T})+\log n))$. In particular, in the refinement operation applied directly after edge insertion, we have that $|T_{\mathsf{pref}}| \leq 2|\mathsf{hgt}(\mathcal{T})|$, so the potential function increases by at most $k^{\mathcal{O}(k)}\mathsf{hgt}(\mathcal{T})^2$ (note that $\mathsf{hgt}(\mathcal{T}) \geq \log n$).

Let us now turn to two issues that we have delayed for some time. How to guarantee that the k-closure X is d-linked, and how to actually find such an X? We say that a k-closure is *c-small* if it satisfies the condition of Lemma 6.3 for some concrete bound $c \leq \mathcal{O}(k^4)$ that can be obtained from the proof of Lemma 6.3. The following lemma that uses similar ideas to the proofs in Section 4.2 and Section 5.2 gives a way to guarantee d-linkedness.

Lemma 6.4. *Let T_{pref} be a prefix of a tree decomposition. If X is a c-small k-closure of T_{pref} that among all c-small k-closures of T_{pref} primarily minimizes $|X|$, and secondarily minimizes $d(X)$, then X is d-linked into T_{pref}.*

With Lemma 6.4, we can use dynamic programming for finding c-small k-closures that are d-linked. In particular, we adapt the dynamic programming of Bodlaender and Kloks [1996] for computing treewidth into computing c-small k-closures that optimize for

the conditions in the lemma. This is similar to the dynamic programming in Section 4.4, but with a lot of technical details related to running Bodlaender-Kloks with the graph $\mathsf{torso}_G(X)$. By maintaining these dynamic programming tables throughout the algorithm, we get that given T_{pref}, we can in time $2^{k^{\mathcal{O}(1)}}|T_{\mathsf{pref}}|$ find an c-small k-closure X of T_{pref} that is d-linked, and also the graph $\mathsf{torso}_G(X)$.

6.1.3 Height reduction

In this subsection we sketch the height reduction scheme, in particular, the following lemma.

Lemma 6.5 (Height reduction)**.** *Let $\mathcal{T}$ be the tree decomposition we are maintaining. If $\mathsf{hgt}(\mathcal{T}) > 2^{\Omega(k \log k \sqrt{\log n \log\log n})}$, then there exists a prefix T_{pref} of $\mathcal{T}$ so that the refinement operation on T_{pref} results in a tree decomposition $\mathcal{T}'$ with $\Phi(\mathcal{T}') < \Phi(\mathcal{T})$ and runs in time $2^{k^{\mathcal{O}(1)}}(\Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$.*

To sketch the proof of Lemma 6.5, let us build a certain model of accounting how the potential $\Phi(\mathcal{T})$ changes in a refinement operation with a prefix T_{pref}. First, recall that Lemma 6.2 takes care of the potential in the tree decompositions $\mathcal{T}^C$ for components C of $G \setminus X$, and the only place we need to worry about increasing the potential are the nodes of $\mathcal{T}^X$. In the previous section we bounded this increase by $k^{\mathcal{O}(k)}(|T_{\mathsf{pref}}| \cdot (\mathsf{hgt}(\mathcal{T}) + \log n))$, where in particular the factor $\mathsf{hgt}(\mathcal{T}) + \log n$ comes from the fact that after attaching the tree decomposition $\mathcal{T}^C$, the height of a node in $\mathcal{T}^X$ could be as large as $\mathsf{hgt}(\mathcal{T}) + \log n$. This upper bound was sufficient for the refinement operation performed after an edge addition to control the width, but to prove Lemma 6.5 we need a more fine-grained view.

Consider the following model. We start with the tree decomposition $\mathcal{T}^X$ of height $\mathcal{O}(\log n)$, and attach the tree decompositions of the components $\mathcal{T}^C$ to it one by one. Each time we attach a tree decomposition $\mathcal{T}^C$, we increase the height of at most $\mathcal{O}(\log n)$ nodes in $\mathcal{T}^X$ (because $\mathsf{hgt}(\mathcal{T}^X) \leq \mathcal{O}(\log n)$), and this height is increased by at most $\mathsf{hgt}(\mathcal{T}^C)$, which is at most $\mathsf{hgt}(a)$, where a is the appendix of T_{pref} whose subtree contains C. While there can be many components C contained in $\mathcal{T}_a$, observe that Lemma 6.3 implies that in fact such components can have only at most $k^{\mathcal{O}(k)}$ different neighborhoods $N(C)$, and therefore at most $k^{\mathcal{O}(k)}$ different attachment points in $\mathcal{T}^X$. In particular, after handling technical details about the binary trees used to flatten high-degree attachment points, we can assume that an appendix a of T_{pref} is responsible for increasing the height of at most $k^{\mathcal{O}(k)} \log n$ nodes in $\mathcal{T}^X$. Moreover, it increases the height of those nodes by at most $\mathsf{hgt}(a)$ each, so in total it is responsible for increasing the potential by $k^{\mathcal{O}(k)} \cdot \mathsf{hgt}(a) \cdot \log n$.

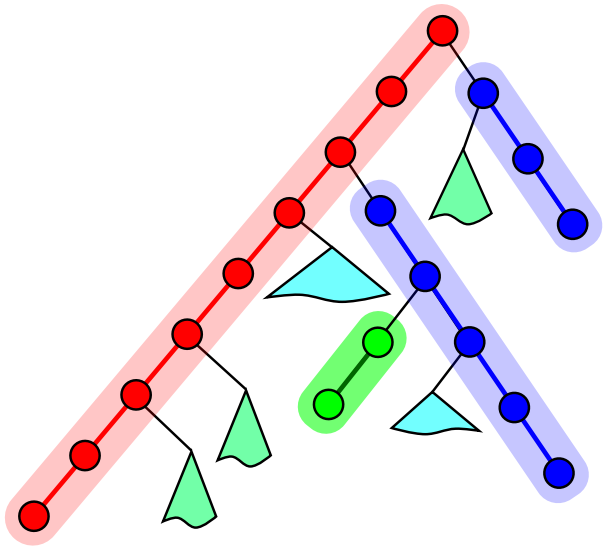

Figure 6.2: Construction of T_{pref} in height reduction. The consecutive paths extracted by the construction procedure are depicted in red, blue, and green. Their union constitutes T_{pref}. Big trees are depicted in sea-green, shallow trees are depicted in cyan.

Denote $\Phi(T_{\mathsf{pref}}) = \sum_{t \in T_{\mathsf{pref}}} (\gamma \cdot k)^{|\mathsf{bag}(t)|} \cdot \mathsf{hgt}(t)$, i.e., the value of the potential on the nodes in T_{pref}. The above discussion, combined with Lemma 6.2, leads to the following lemma.

Lemma 6.6 (Informal)**.** *Let T_{pref} be a prefix of a tree decomposition $\mathcal{T}$ and $\mathcal{T}'$ the tree decomposition resulting from refining $\mathcal{T}$ with T_{pref}. It holds that*

$$\Phi(\mathcal{T}') \leq \Phi(\mathcal{T}) - \Phi(T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log n + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{hgt}(a) \cdot \log n.$$

Now, in order to prove Lemma 6.5, it is sufficient to prove that if $\mathcal{T}$ has too large height, then there exists a prefix T_{pref} so that

$$\Phi(T_{\mathsf{pref}}) > c_k \log n \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(a) \right),$$

where c_k is some large enough number depending on k. (Here, the required c_k comes from the number of attachment points and the potential function, so some $c_k = k^{\mathcal{O}(k)}$ is sufficient.) In our proof, the left hand side $\Phi(T_{\mathsf{pref}})$ in fact will be larger than the right hand side $c_k \log n \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(a) \right)$ by an arbitrary constant factor, which gives also the required property that the refinement operation on such a prefix T_{pref} will run in time $2^{k^{\mathcal{O}(1)}}(\Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$.

We first sketch how to find such T_{pref} when $\mathsf{hgt}(T) > n^{\varepsilon}$ for some $\varepsilon > 0$ and k is small compared to n. See Figure 6.2 for an illustration. Assume for simplicity that $|V(T)| = n$.

The natural strategy is to start by setting T_{pref} to be the path from the root to the deepest leaf in T. Then we have $\Phi(T_{\mathsf{pref}}) \geq \Omega(n^{2\varepsilon})$. However, T_{pref} may have $n^{\varepsilon}/2$ appendices that each have height $n^{\varepsilon}/2$, so it is possible that $\sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(a) \cdot \log n \geq n^{2\varepsilon} \cdot \log n/4$. The key observation is that in this case, many subtrees rooted at the appendices $a \in \mathsf{app}(T_{\mathsf{pref}})$ must be even more unbalanced than T is, having height at least $n^{\varepsilon}/2$ while containing at most $4 \cdot n^{1-\varepsilon}$ nodes. In particular, let us say that a subtree rooted on an appendix a is *big* if it contains more than $c_k \cdot n^{1-\varepsilon} \cdot \log n$ nodes, and *shallow* if $\mathsf{hgt}(a) \leq n^{\varepsilon}/(c_k \cdot \log n)$. Now, there can be at most $n/(c_k \cdot n^{1-\varepsilon} \cdot \log n) = n^{\varepsilon}/(c_k \cdot \log n)$ big subtrees, so they contribute at most $k^{\mathcal{O}(k)} \cdot n^{2\varepsilon}/c_k$ to the sum $\sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{hgt}(a) \cdot \log n$. Similarly, the shallow subtrees also contribute at most $k^{\mathcal{O}(k)} \cdot n^{2\varepsilon}/c_k$ to the sum, so by making the constant c_k large enough, the sum coming from subtrees that are big or shallow is only a tiny fraction of $\Phi(T_{\mathsf{pref}})$.

Then, there are subtrees rooted at appendices that are neither big nor shallow. These subtrees are *small and deep*, hence they seem even more unbalanced than $\mathcal{T}$. We apply the same strategy to those trees recursively. For each appendix a whose subtree is small and deep, we insert to T_{pref} the path P_a from a to its deepest descendant. As the subtree is deep, we have that $\Phi(T_{\mathsf{pref}})$ increases by $\Phi(P_a) = \Omega(n^{2\varepsilon}/(c_k \cdot \log n)^2)$. Now, when analyzing the appendices of P_a, we again apply the strategy to handle subtrees that are big or shallow by charging them from $\Phi(P_a)$, and then handling subtrees that are both small and deep recursively. This time, the right definition of big will be to have at least $n^{1-2\varepsilon}(\log n \cdot c_k)^3$ nodes, and the right definition of shallow will be to have height at most $n^{\varepsilon}/(c_k \cdot \log n)^2$. More generally, on the i-th level of such recursion we can call a subtree big if it contains more than $n^{1-i\varepsilon}(\log n \cdot c_k)^{i \cdot (i+1)/2}$ nodes, and shallow if its height is at most $n^{\varepsilon}/(c_k \cdot \log n)^i$. When ε is a constant, this recursion can continue only for a constant number of levels before no subtree can be both small and deep, simply because it would require the subtree to have larger height than the number of nodes. Therefore, in the end we are able to find a prefix T_{pref} that satisfies the requirements of Lemma 6.5.

It is not surprising that selecting the height limit to be n^{ε} is not optimal. In particular, the same strategy as outlined above will work if we select the initial height to be of the form $2^{\Omega(k \log k \sqrt{\log n \log\log n})}$, resulting in showing that we can maintain height $2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$ and thus obtain amortized update time of $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$.

Organization

The rest of this chapter is organized as follows. First, in Section 6.2 we give a framework for formalizing the maintenance of dynamic programming schemes on dynamic tree decomposition. Then, in Section 6.3 we introduce k-closures and prove structural results

about them. In Section 6.4 we give our data structure for computing k-closures and auxiliary objects related to them. Section 6.5 is dedicated to the refinement operation and Section 6.6 to the height improvement scheme. Finally, after gathering all of the ingredients, we put the proof of Theorem 1.4 together in Section 6.7. We remark that the material with the most novelty is presented in Sections 6.3, 6.5, and 6.6, while Sections 6.2 and 6.4 are mostly about definitions and relatively straightforward consequences of known results.

6.2 Dynamic dynamic programming

In this section[1] we introduce a framework for dynamic maintenance of dynamic programming tables, and other information, on tree decompositions, under specific types of updates called *prefix-rebuilding updates*. To this end, we also formalize dynamic programming on tree decompositions by *tree decomposition automata*. We also give applications of this framework, although the applications presented in this section are simple corollaries of known results. A more involved application, namely our data structure for computing k-closures, will be given in Section 6.4.

We remark that maintenance of runs of automata on dynamic trees and tree decompositions has already been investigated in the literature, even for the much more general problem of dynamic enumeration. See for example the works of Hagerup [2000], Niewerth [2018], and Amarilli et al. [2019], and the bibliographic discussion within. In particular, many (although not all) results contained in this section could be in principle derived from [Amarilli et al., 2018, Lemma 7.3], but not in a black-box manner and without concrete bounds on update time. The material in Section 4.4 could also be considered to be a predecessor of the material in this section, for the special case we needed in Chapter 4.

6.2.1 Prefix-rebuilding data structures

In our algorithm, we will maintain dynamic programming schemes on tree decompositions that are updated by *prefix rebuilding-updates*. These updates, informally speaking, change some rooted tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ by editing some prefix $T_{\mathsf{pref}} \subseteq V(T)$ of the tree T and the bags of the prefix. A *prefix-rebuilding data structure* is responsible for maintaining some information on the tree decomposition under prefix-rebuilding updates, so that the time to update the data structure is proportional to the size $|T_{\mathsf{pref}}|$ of the

[1] We attribute the joke in the title of the section to Chen et al. [2021].

changed prefix, times some *overhead* per node, which typically depends only on the width. This generalizes the ideas used in Section 4.4 for re-computing the dynamic programming tables only for the editable nodes.

The tree decompositions maintained by prefix-rebuilding data structures will store in addition the edges of the dynamic graph G we maintain, and thus represent also G. Furthermore, we will require that the maintained tree decomposition is binary (i.e., the tree T is a binary tree), for the same reasons as the tree decomposition in Chapter 4 was required to be subcubic.

Formally, we define an *annotated tree decomposition* of a graph G to be a triple $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$, where

- (T, bag) is a binary tree decomposition of G, and
- the function $\mathsf{edges}\colon V(T) \to 2^{E(G)}$ maps each node $t \in V(T)$ to the set $\mathsf{edges}(t) = \{uv \in E(G) \mid \mathsf{forget}_{(T,\mathsf{bag})}(uv) = t\}$, that is, to the set of edges for which t is the unique node closest to the root containing both endpoints of the edge.

Note that edges is uniquely determined from G and (T, bag), and conversely, G is uniquely determined from $(T, \mathsf{bag}, \mathsf{edges})$. Given a set $X \subseteq V(T)$, the *restriction* of $(T, \mathsf{bag}, \mathsf{edges})$ to X, denoted $(T, \mathsf{bag}, \mathsf{edges})\restriction_X$, is the tuple $(T[X], \mathsf{bag}\restriction_X, \mathsf{edges}\restriction_X)$.

Next, consider an update changing an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ to another annotated tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$. This update can also change the underlying graph G, in particular, it changes G to be the graph uniquely determined by $(T', \mathsf{bag}', \mathsf{edges}')$. The update is a *prefix-rebuilding update* if $(T', \mathsf{bag}', \mathsf{edges}')$ is created from $(T, \mathsf{bag}, \mathsf{edges})$ by replacing a prefix T_{pref} of T with a new rooted tree $T^\star$ and then "re-attaching" the subtrees of T rooted at the appendices of T_{pref} below the nodes of T'_{pref}. Formally, a prefix-rebuilding update is described by a tuple $\overline{u} = (T_{\mathsf{pref}}, T'_{\mathsf{pref}}, T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star, \pi)$ where

- $T_{\mathsf{pref}} \subseteq V(T)$ is a prefix of T,
- $T'_{\mathsf{pref}} \subseteq V(T')$ is a prefix of T' satisfying

$$(T, \mathsf{bag}, \mathsf{edges})\restriction_{V(T)\setminus T_{\mathsf{pref}}} = (T', \mathsf{bag}', \mathsf{edges}')\restriction_{V(T')\setminus T'_{\mathsf{pref}}}$$

- $(T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star) = (T', \mathsf{bag}', \mathsf{edges}')\restriction_{T'_{\mathsf{pref}}}$ and
- $\pi\colon \mathsf{app}(T_{\mathsf{pref}}) \to T'_{\mathsf{pref}}$ is a function that maps the appendices of T_{pref} to nodes of T'_{pref} such that for each appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$, the parent of a in T' is $\pi(a)$.

It is straightforward that $(T', \mathsf{bag}', \mathsf{edges}')$ can be uniquely determined from $(T, \mathsf{bag}, \mathsf{edges})$ and the tuple $\overline{u}$ as above. The tuple $\overline{u}$ is called a *description* of the prefix-rebuilding update. The *size* of $\overline{u}$, denoted $|\overline{u}|$, is defined as $|T_{\mathsf{pref}}| + |T'_{\mathsf{pref}}|$. It is also straightforward that given $\overline{u}$, a representation of $(T, \mathsf{bag}, \mathsf{edges})$ can be turned into a representation of $(T', \mathsf{bag}', \mathsf{edges}')$ in time $\ell^{\mathcal{O}(1)} \cdot |\overline{u}|$, where ℓ is the maximum of the widths of $(T, \mathsf{bag}, \mathsf{edges})$ and $(T', \mathsf{bag}', \mathsf{edges}')$.

A dynamic data structure is an *ℓ-prefix-rebuilding data structure* with *overhead* τ, where τ is typically a function of ℓ, if it stores an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ and supports the following operations.

- $\mathsf{Initialize}(T, \mathsf{bag}, \mathsf{edges})$: Initializes $(T, \mathsf{bag}, \mathsf{edges})$ with the given annotated tree decomposition of width at most ℓ. Runs in time $\ell^{\mathcal{O}(1)} \cdot \tau \cdot |V(T)|$.
- $\mathsf{Update}(\overline{u})$: Applies the prefix-rebuilding update described by $\overline{u}$ to the stored annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$. Assumes that the result is indeed an annotated tree decomposition of width at most ℓ. Runs in time $\ell^{\mathcal{O}(1)} \cdot \tau \cdot |\overline{u}|$.

Usually, the overhead τ will correspond to the time necessary to recompute any auxiliary information associated with each node of the decomposition undergoing the update. For example, the height of a node t in an annotated tree decomposition can be inferred in $\mathcal{O}(1)$ time from the heights of the (at most two) children of t, so the overhead required to recompute the heights of the nodes after the update is $\tau = \mathcal{O}(1)$.

Prefix-rebuilding data structures will usually implement an additional operation allowing to efficiently query the current state of the data structure. For example, next we state a data structure that allows us to access various auxiliary information about the tree decomposition.

Lemma 6.7. *For every integer ℓ, there exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}(1)$ that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a dynamic graph G, and additionally implements the following operations.*

- $\mathsf{hgt}(t)$*: Given a node $t \in V(T)$, returns the height of t in T. Runs in time $\mathcal{O}(1)$.*
- $\mathsf{size}(t)$*: Given a node $t \in V(T)$, returns the number of nodes in the subtree of T rooted at t, i.e., $|\mathsf{desc}_T(t)|$. Runs in time $\mathcal{O}(1)$.*
- $\mathsf{forget}(v)$*: Given a vertex $v \in V(G)$, returns the node $\mathsf{forget}_{\mathcal{T}}(v)$, i.e., the unique bag closest to the root that contains v. Runs in time $\mathcal{O}(1)$.*

The proof of Lemma 6.7 uses standard arguments on dynamic programming on tree decompositions. It will be proved in Subsection 6.2.4, after we first introduce more definitions that make its proof streamlined. More generally, any typical bottom-up dynamic programming scheme on tree decompositions can be turned into a prefix-rebuilding data structure. In particular, in several places in this chapter we need to maintain dynamic programming schemes on tree decompositions under prefix-rebuilding updates. In every case, we state a suitable lemma about the existence of a prefix-rebuilding data structure.

Finally, we show that the assumption that the function $\mathsf{edges}^\star$ is given in the description of a prefix-rebuilding update can be lifted in prefix-rebuilding updates that do not change the underlying graph G. Consider a prefix-rebuilding update that does not change the graph G, and let us say that a *weak description* of the update is a tuple $\widehat{u} = (T_{\mathsf{pref}}, T'_{\mathsf{pref}}, T^\star, \mathsf{bag}^\star, \pi)$ that is required to satisfy the same properties as a description of a prefix-rebuilding update except for the edges function. Because the graph G is not changed, the new annotated tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$ can be determined uniquely from $(T, \mathsf{bag}, \mathsf{edges})$ and $\widehat{u}$. We again denote $|\widehat{u}| = |T_{\mathsf{pref}}| + |T'_{\mathsf{pref}}|$.

We show that a weak description $\widehat{u}$ of a prefix-rebuilding update can be turned into a description $\overline{u}$ of a prefix-rebuilding update such that $|\overline{u}| = \mathcal{O}(|\widehat{u}|)$ and the annotated tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$ resulting from applying $\overline{u}$ is the same as the one resulting from applying $\widehat{u}$. We note that this operation can make the sets T_{pref} and T'_{pref} larger, but this is bounded by $\mathcal{O}(|\widehat{u}|)$.

Lemma 6.8. *For every integer ℓ, there exists an ℓ-prefix-rebuilding data structure that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a dynamic graph G with overhead $\mathcal{O}(1)$, and additionally implements the following operation.*

- $\mathsf{Strengthen}(\widehat{u})$*: Given a weak description $\widehat{u}$ of a prefix-rebuilding update, returns a description $\overline{u}$ of a prefix-rebuilding update such that $|\overline{u}| = \mathcal{O}(\widehat{u})$ and applying $\widehat{u}$ and $\overline{u}$ to $\mathcal{T}$ result in the same annotated tree decomposition $\mathcal{T}'$. Runs in time $|\widehat{u}| \cdot \ell^{\mathcal{O}(1)}$.*

Proof. Let $\widehat{u} = (\widehat{T}_{\mathsf{pref}}, \widehat{T}'_{\mathsf{pref}}, \widehat{T}^\star, \widehat{\mathsf{bag}}^\star, \widehat{\pi})$ and $\mathcal{T}' = (T', \mathsf{bag}', \mathsf{edges}')$ be the resulting annotated tree decomposition. We observe that the forget-node of an edge $uv \in E(G)$ can change only if $u, v \in \mathsf{bags}_{\mathcal{T}'}(\widehat{T}'_{\mathsf{pref}})$. However, if $u, v \in \mathsf{bags}_{\mathcal{T}'}(\widehat{T}'_{\mathsf{pref}})$, then because of the connectedness condition of $\mathcal{T}'$ it must hold that $u, v \in \mathsf{bags}_{\mathcal{T}}(\widehat{T}_{\mathsf{pref}} \cup \mathsf{app}(\widehat{T}_{\mathsf{pref}}))$, and in particular, uv must be stored in $\mathsf{edges}(\widehat{T}_{\mathsf{pref}} \cup \mathsf{app}(\widehat{T}_{\mathsf{pref}}))$. Therefore, the changes to the edges function are limited to the subtree of T consisting of $\widehat{T}_{\mathsf{pref}} \cup \mathsf{app}(\widehat{T}_{\mathsf{pref}})$, and therefore for constructing $\overline{u} = (T_{\mathsf{pref}}, T'_{\mathsf{pref}}, T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star, \pi)$ it suffices to take $T_{\mathsf{pref}} = \widehat{T}_{\mathsf{pref}} \cup \mathsf{app}(\widehat{T}_{\mathsf{pref}})$ and analogously construct T'_{pref}, $T^\star$, $\mathsf{bag}^\star$, and π from $\widehat{T}'_{\mathsf{pref}}$, $\widehat{T}^\star$, $\widehat{\mathsf{bag}}^\star$, and $\widehat{\pi}$. Then, the

$\mathsf{edges}^\star$ function can be determined from $(T^\star, \mathsf{bag}^\star)$ and the edges function restricted to $\widehat{T}_{\mathsf{pref}} \cup \mathsf{app}(\widehat{T}_{\mathsf{pref}})$. The running time and the bound on $|\overline{u}|$ follow from the fact that the tree decompositions are binary. □

By using the data structure from Lemma 6.8, we can assume when constructing prefix-rebuilding updates that do not change G that it is sufficient to construct a weak description, but when implementing prefix-rebuilding data structures that the Update method receives a (not weak) description.

6.2.2 Tree decomposition automata

We now introduce our notion of automata processing tree decompositions. While this notion is tailored here to our specific purposes, the idea of processing tree decompositions using various kinds of automata dates back to the work of Courcelle [1990] and is a thoroughly researched topic, see for example [Downey and Fellows, 2013, Chapter 12] and [Flum and Grohe, 2006, Chapters 10 and 11]. Hence, this subsection can be considered a formalization of folklore.

Boundaried graphs

We will work with graphs with specified boundaries, as formalized next.

Definition 6.9 (Boundaried graph)**.** *A boundaried graph is a graph G together with a set of vertices $\partial G \subseteq V(G)$, called the boundary, such that G has no edge with both endpoints in ∂G. A boundaried tree decomposition of a boundaried graph G is a triple $(T, \mathsf{bag}, \mathsf{edges})$ that is an annotated tree decomposition of G where in addition we require that ∂G is contained in the root bag.*

When speaking about a boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of a boundaried graph G, we redefine the adhesion of the root of T to be ∂G, rather than the empty set. Note that to be concise, in our definition a boundaried tree decomposition is always an annotated tree decomposition (which also implies that T is binary).

Suppose $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ is a boundaried tree decomposition of a boundaried graph G and x is a node of T. Then we say that x *induces* a boundaried graph G_x and its

boundaried tree decomposition $\mathcal{T}_x = (T_x, \mathsf{bag}_x, \mathsf{edges}_x)$, defined as

$$\begin{aligned} V(G_x) =& \mathsf{bags}_{\mathcal{T}}(\mathsf{desc}_T(x)), \\ E(G_x) =& \bigcup_{y \in \mathsf{desc}_T(x)} \mathsf{edges}(y) \\ \partial G_x =& \mathsf{adh}_{\mathcal{T}}(x), \text{ and} \\ \mathcal{T}_x = (T_x, \mathsf{bag}_x, \mathsf{edges}_x) =& \mathcal{T} \restriction_{\mathsf{desc}_T(x)}. \end{aligned}$$

It is clear that $\mathcal{T}_x$ defined as above is a boundaried tree decomposition of G_x. We use the above notation only when the boundaried tree decomposition $\mathcal{T}$ is clear from the context.

Automata

We now introduce our automaton model.

Definition 6.10 (Tree decomposition automaton)**.** *A (deterministic) tree decomposition automaton of width ℓ consists of*

- *a state set Q,*
- *a set of accepting states $F \subseteq Q$,*
- *an initial mapping ι that maps every boundaried graph G on at most $\ell + 1$ vertices to a state $\iota(G) \in Q$, and*
- *a transition mapping δ that maps every 7-tuple of form (B, X, Y, Z, J, q', q''), where B is a set of size at most $\ell+1$, $X, Y, Z \subseteq B$, $J \in \binom{B}{2} \setminus \binom{X}{2}$, $q' \in Q$, and $q'' \in Q \cup \{\bot\}$ to a state $\delta(B, X, Y, Z, J, q', q'') \in Q$.*

The run of a tree decomposition automaton $\mathcal{A}$ on a boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of a boundaried graph G is the unique labeling $\rho_{\mathcal{A}} \colon V(T) \to Q$ satisfying the following properties.

- *For every leaf node x of T, we have*

$$\rho_{\mathcal{A}}(x) = \iota(G_x).$$

- *For every non-leaf node x of T with one child y, we have*

$$\rho_{\mathcal{A}}(x) = \delta(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \emptyset, \mathsf{edges}(x), \rho_{\mathcal{A}}(y), \bot).$$

- *For every non-leaf node x of T with two children y and z, we have*

$$\rho_{\mathcal{A}}(x) = \delta(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \mathsf{adh}(z), \mathsf{edges}(x), \rho_{\mathcal{A}}(y), \rho_{\mathcal{A}}(z)).$$

A tree decomposition automaton $\mathcal{A}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if $\rho_{\mathcal{A}}(r) \in F$, where r is the root of T.

Note that in the transitions described above, nodes with one child are treated by passing a "dummy state" $\perp \notin Q$ to the transition function instead of a state. Note that this allows δ also to recognize when there is only one child. Also, the automata model presented above could in principle distinguish the left child y from the right child z and treat states passed from them differently. However, this will never be the case in our applications. In all constructed automata, the transition mapping will be symmetric with respect to swapping the role of the children y and z.

We say that a tree decomposition automaton $\mathcal{A}$ has *evaluation time* τ if the functions ι and δ can be evaluated on any tuple of their arguments in time τ, and moreover for a given $q \in Q$ it can be decided whether $q \in F$ in time τ. Note that we do *not* require the state set Q to be finite. In fact, in most of our applications it will be infinite, but we will be able to efficiently represent and manipulate the states.

We will often run a tree decomposition automaton on a non-boundaried annotated tree decomposition of a non-boundaried graph G. In such cases, we simply apply all the above definitions while treating G as a boundaried graph with an empty boundary.

For technical reasons we will also use *nondeterministic tree decomposition automata*, which are defined just like in Definition 6.10, except that ι and δ are the *initial relation* and the *transition relation*, instead of mappings, and the state set Q is required to be finite. In particular, ι is a relation consisting of pairs of the form (G, q), where G is a boundaried graph on at most $\ell + 1$ vertices, and $q \in Q$. Similarly, δ is a relation consisting of pairs of the form $((B, X, Y, Z, J, q', q''), q)$, where (B, X, Y, Z, J, q', q'') is a 7-tuple like in the domain of the transition mapping, and $q \in Q$. Then a run of a nondeterministic automaton $\mathcal{A}$ on a boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ is a labeling ρ of the nodes of T with states such that $(G_x, \rho(x)) \in \iota$ for every leaf node x, $((\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \emptyset, \mathsf{edges}(x), \rho(y), \perp), \rho(x)) \in \delta$ for every node x with one child y, and $((\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \mathsf{adh}(z), \mathsf{edges}(x), \rho(y), \rho(z)), \rho(x)) \in \delta$ for every node x with two children y and z. Note that a nondeterministic tree decomposition automaton may have multiple runs on a single tree decomposition. We say that $\mathcal{A}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if there is a run of $\mathcal{A}$ on $(T, \mathsf{bag}, \mathsf{edges})$ that is *accepting*, that is, the state associated with the root node is accepting.

In the context of nondeterministic automata, by evaluation time we mean the time needed to decide whether a given pair belongs to any of the relations ι or δ, or to decide whether a given state is accepting. Note that if $\mathcal{A}$ is a nondeterministic tree decomposition automaton with a finite state set Q, then we can determinize it, i.e., find a deterministic automaton $\mathcal{A}'$ that accepts the same tree decompositions, using the standard powerset construction. Then the state set of $\mathcal{A}'$ is 2^Q. In the following, all automata are deterministic unless explicitly stated.

6.2.3 Automata constructions

We now present four automata constructions that we will use.

Tree decomposition properties automata

We first construct two very simple automata that are used in Lemma 6.7 for maintaining properties of the tree decomposition itself.

Lemma 6.11. *For every integer ℓ there exists tree decomposition automata $\mathcal{H}_\ell$ and $\mathcal{S}_\ell$ of width ℓ, with the following properties. For any graph G, annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of G of width at most ℓ, and any node x of T, $\rho_{\mathcal{H}_\ell}(x)$ is equal to $\mathsf{hgt}(T_x)$ and $\rho_{\mathcal{S}_\ell}(x)$ is equal to $|V(T_x)|$. The evaluation times of $\mathcal{H}_\ell$ and $\mathcal{S}_\ell$ are $\mathcal{O}(1)$.*

Proof. The state sets of both of the automata are $\mathbb{Z}_{\geq 1}$. Let us define for all $n \in \mathbb{Z}_{\geq 1}$ that $\max(n, \bot) = n$ and $n + \bot = n$. For the height automaton $\mathcal{H}_\ell$, the initial mapping and the transition mapping are defined as (here _ denotes any input value)

$$\iota(\_) = 1$$
$$\delta(\_, \_, \_, \_, \_, q', q'') = 1 + \max(q', q'').$$

For the size automaton $\mathcal{S}_\ell$, the initial mapping and the transition mapping are defined as

$$\iota(\_) = 1$$
$$\delta(\_, \_, \_, \_, \_, q', q'') = 1 + q' + q''.$$

It is straightforward to see that these automata satisfy the required properties. □

CMSO_2-types automaton

As discussed in Subsection 3.3.2, the classical theorem of Courcelle [1990] states that there is an algorithm that given a CMSO_2 sentence φ and an n-vertex graph G together with a tree decomposition of width at most ℓ, decides whether φ is true in G in time $f(\ell, \varphi) \cdot n$, where f is a computable function. One way of proving Courcelle's theorem is to construct a dynamic programming procedure that processes the provided tree decomposition in a bottom-up fashion. This dynamic programming procedure can be understood as a tree decomposition automaton in the sense of Definition 6.10, yielding the following result.

Lemma 6.12 (Courcelle [1990])**.** *For every integer ℓ and CMSO_2 sentence φ, there exists a tree decomposition automaton $\mathcal{A}_{\ell,\varphi}$ of width ℓ, so that for any graph G and its annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, $\mathcal{A}_{\ell,\varphi}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if and only if φ is true in G. The evaluation time is bounded by $f(\ell, \varphi)$ for some computable function f.*

Bodlaender-Kloks automaton

As discussed in Section 3.2, Bodlaender and Kloks [1996] gave an algorithm that, given a graph G, a tree decomposition $\mathcal{T}$ of G of width at most ℓ, and a number $k < \ell$, decides whether the treewidth of G is at most k in time $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)} \cdot |\mathcal{T}|$. From the overview of the proof given in Subsection 3.2.1, it is not hard to see that this dynamic programming can be understood as a nondeterministic tree decomposition automaton with $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}$ states (or as a deterministic tree decomposition automaton with more states). Thus, from the work of Bodlaender and Kloks we can immediately deduce the following statement.

Lemma 6.13. *For every pair of integers $k \leq \ell$ there is a nondeterministic tree decomposition automaton $\mathcal{BK}_{k,\ell}$ of width ℓ so that for any graph G and its annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, $\mathcal{BK}_{k,\ell}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if and only if the treewidth of G is at most k. The state set of $\mathcal{BK}_{k,\ell}$ is of size $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}$ and can be computed in time $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}$. The evaluation time of $\mathcal{BK}_{k,\ell}$ is $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}$ as well.*

The automaton $\mathcal{BK}_{k,\ell}$ will be the only nondeterministic automaton we use. We remark that since the property of having treewidth at most k can be expressed in CMSO_2[2], Lemma 6.13 with an unspecified bound on the evaluation time also follows from Lemma 6.12. The reason behind formulating Lemma 6.13 explicitly is to keep track of the evaluation time more precisely in further arguments. This is also the reason for formulating it as a nondeterministic automaton.

[2]This can be done, for instance, by stating that the given graph does not contain any of the forbidden minor obstructions for having treewidth at most k.

We also need a deterministic version of the Bodlaender-Kloks automaton.

Lemma 6.14. *For every pair of integers $k \leq \ell$ there is a (deterministic) tree decomposition automaton $\mathcal{BK}^d_{k,\ell}$ of width ℓ so that for any graph G and its annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, $\mathcal{BK}^d_{k,\ell}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if and only if the treewidth of G is at most k. The evaluation time of $\mathcal{BK}^d_{k,\ell}$ is $2^{\mathcal{O}((k+\log \ell)\cdot \ell^2)}$.*

Proof. Follows from the work of Bodlaender and Kloks [1996], or from Lemma 6.13 by keeping track of the set of reachable states. □

6.2.4 Dynamic maintenance of automata runs

Having defined the automata we are going to use, we now show that runs of deterministic tree decomposition automata can be maintained efficiently under prefix-rebuilding updates.

Lemma 6.15. *Let $\mathcal{A} = (Q, F, \iota, \delta)$ be a tree decomposition automaton of width ℓ and evaluation time τ. Then there exists an ℓ-prefix-rebuilding data structure with overhead τ that additionally implements the operation*

- $\mathsf{State}(t)$*: Given a node t of T, returns $\rho_{\mathcal{A}}(t)$. Runs in time $\mathcal{O}(1)$.*

Proof. At every point in time, the data structure stores the annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$, and for every node $t \in V(T)$ the state $\rho_{\mathcal{A}}(t)$ in the run of $\mathcal{A}$ on $(T, \mathsf{bag}, \mathsf{edges})$. This allows for answering queries in constant time, as requested.

For initialization, we just compute the run of $\mathcal{A}$ on $(T, \mathsf{bag}, \mathsf{edges})$ in a bottom-up manner. The states for leaves are computed according to the initialization mapping, while the states for internal nodes are computed according to the transition mapping bottom-up. This requires time τ per node, so $\mathcal{O}(\tau \cdot |V(T)|)$ in total.

To apply a prefix-rebuilding update with a description $\overline{u} = (T_{\mathsf{pref}}, T'_{\mathsf{pref}}, T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star, \pi)$, the representation of $(T, \mathsf{bag}, \mathsf{edges})$ can be easily rebuilt in time $\ell^{\mathcal{O}(1)} \cdot |\overline{u}|$ by building the tree $T^\star$ and reattaching the subtrees rooted at appendices of T_{pref} to $T^\star$ according to π, using a single pointer change per appendix. Observe here that the information about the run of $\mathcal{A}$ on the reattached subtrees does not need to be altered, except for the appendices of T'_{pref}, for which the run could have to be altered because their adhesions could change. Hence, it remains to compute the states associated with the nodes of $T'_{\mathsf{pref}} \cup \mathsf{app}(T'_{\mathsf{pref}})$ in the run of $\mathcal{A}$ on the new decomposition $(T', \mathsf{bag}', \mathsf{edges}')$. This can be

done by processing $T'_{\mathsf{pref}} \cup \mathsf{app}(T'_{\mathsf{pref}})$ in a bottom-up manner, and computing each state using either the initialization mapping ι (for the leaves of T') or the transition mapping δ (for the non-leaf nodes), in total time $\mathcal{O}(\tau \cdot |T'_{\mathsf{pref}} \cup \mathsf{app}(T'_{\mathsf{pref}})|) \leq \mathcal{O}(\tau \cdot |\overline{u}|)$. □

Let us now complete the proof of Lemma 6.7, which we restate here.

Lemma 6.7. *For every integer ℓ, there exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}(1)$ that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a dynamic graph G, and additionally implements the following operations.*

- $\mathsf{hgt}(t)$*: Given a node $t \in V(T)$, returns the height of t in T. Runs in time $\mathcal{O}(1)$.*
- $\mathsf{size}(t)$*: Given a node $t \in V(T)$, returns the number of nodes in the subtree of T rooted at t, i.e., $|\mathsf{desc}_T(t)|$. Runs in time $\mathcal{O}(1)$.*
- $\mathsf{forget}(v)$*: Given a vertex $v \in V(G)$, returns the node $\mathsf{forget}_{\mathcal{T}}(v)$, i.e., the unique bag closest to the root that contains v. Runs in time $\mathcal{O}(1)$.*

Proof. Lemma 6.15 combined with Lemma 6.11 immediately gives an ℓ-prefix-rebuilding data structure implementing the first two operations. Therefore, it suffices to implement an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}(1)$ that implements the operation $\mathsf{forget}(v)$.

Consider a prefix-rebuilding update changing $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ to $\mathcal{T}' = (T', \mathsf{bag}', \mathsf{edges}')$. Observe that a prefix-rebuilding update can change the highest node where v occurs only if $v \in \mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}} \cup \mathsf{app}(T_{\mathsf{pref}}))$, and in particular, in that case the highest node of $(T', \mathsf{bag}', \mathsf{edges}')$ where v occurs will be in $T'_{\mathsf{pref}} \cup \mathsf{app}(T'_{\mathsf{pref}})$. Both $|T_{\mathsf{pref}} \cup \mathsf{app}(T_{\mathsf{pref}})|$ and $|T'_{\mathsf{pref}} \cup \mathsf{app}(T'_{\mathsf{pref}})|$ are linear in $|\overline{u}|$, so we simply maintain the mapping $\mathsf{forget}(v)$ explicitly by recomputing it for all vertices $v \in \mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}} \cup \mathsf{app}(T_{\mathsf{pref}}))$. □

We also provide two prefix-rebuilding data structures that will be used in Section 6.7. The first is about maintaining whether G satisfies a CMSO_2-expressible property.

Lemma 6.16. *For every integer ℓ and CMSO_2 sentence φ, there exists an ℓ-prefix-rebuilding data structure with overhead $f(\ell, \varphi)$, for some computable function f, that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a dynamic graph G, and additionally implements the following operation.*

- $\mathsf{Evaluate}()$*: Returns whether φ is true in G, in $\mathcal{O}(1)$ time.*

Proof. Follows from combining Lemma 6.15 with the automaton of Lemma 6.12. □

The second is about maintaining the treewidth of G.

Lemma 6.17. *For every pair of integers $k \leq \ell$, there exists an ℓ-prefix-rebuilding data structure with overhead $2^{\mathcal{O}(\ell^3)}$ that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a dynamic graph G, and additionally implements the following operation.*

- $\mathsf{Treewidth}()$*: Returns whether $\mathsf{tw}(G) \leq k$, in $\mathcal{O}(1)$ time.*

Proof. Follows from combining Lemma 6.15 with the automaton of Lemma 6.14. □

6.3 Closures

In this section, we introduce the graph-theoretical notion of a *closure* and present several results about them. Closures will be closely related to torso tree decompositions of Chapter 5, and they will be the key objects in the refinement operation of our algorithm.

Recall that $\mathsf{torso}_G(X)$ is the graph obtained from $G[X]$ by making for every $C \in \mathsf{cc}(G \setminus X)$ the neighborhood $N(C)$ into a clique. A *k-closure* of a set of vertices W is defined as follows.

Definition 6.18 (k-closure)**.** *Let k be an integer, G a graph, and $W \subseteq V(G)$. Then, a set $X \subseteq V(G)$ is called a k-closure of W in G if*

$$W \subseteq X \qquad \text{and} \qquad \mathsf{tw}(\mathsf{torso}_G(X)) \leq 2k+1.$$

This is similar to the definition of a torso tree decomposition that covers a set W in Chapter 5. Having treewidth bounded by $2k+1$ despite being named k-closure is a bit unintuitive, but this is motivated by that in this section we will prove that if G has treewidth at most k, then any set $W \subseteq V(G)$ has a k-closure that satisfies certain properties.

6.3.1 Small closures

In our algorithm, the set W in the definition of k-closure will be chosen as the union of bags of some prefix of the tree decomposition (T, bag) we are maintaining, i.e., $W = \mathsf{bags}(T_{\mathsf{pref}})$ for some prefix T_{pref} of T. We will be interested in closures X that are "small" compared to the prefix T_{pref}. The size of X should not be much larger than the size of T_{pref}, and

moreover, we will also require a more intricate smallness condition. This condition asserts that for every appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$, only a bounded number of vertices from the bags of the subtree rooted at a will be selected to the closure.

Definition 6.19 (c-small)**.** *Let (T, bag) be a tree decomposition of G and T_{pref} a prefix of T. Let also c be an integer. Then we say that a set $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$ is c-small with respect to (T, bag) if for every appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$, it holds that $|X \cap \mathsf{cmp}(a)| \leq c$.*

Note that c-smallness of X implies that $|X| \leq c \cdot |\mathsf{app}(T_{\mathsf{pref}})| + |\mathsf{bags}(T_{\mathsf{pref}})|$, which in particular if T is binary implies that $|X| \leq (c + \mathsf{width}(T, \mathsf{bag}) + 1) \cdot (|T_{\mathsf{pref}}| + 1)$.

We show in this subsection that if $\mathsf{tw}(G) \leq k$ and $\mathsf{width}(T, \mathsf{bag}) \leq \ell$, then for any prefix T_{pref} there exists a $\mathcal{O}(\ell^4)$-small k-closure of T_{pref}. This will be called the *Small Closure Lemma*.

Let us start with an auxiliary lemma for constructing k-closures from tree decompositions. This lemma explains the treewidth bound $2k + 1$ in the definition of k-closure. Recall that a set of nodes $S \subseteq V(T)$ in a rooted tree T is LCA-closed if for each pair $x, y \in S$, the LCA of x and y is also in S.

Lemma 6.20. *Let (T, bag) be a rooted tree decomposition of a graph G of width at most k. Let also $S \subseteq V(T)$ be an LCA-closed set of nodes of T. Then,*

$$\mathsf{tw}(\mathsf{torso}_G(\mathsf{bags}(S))) \leq 2k + 1.$$

Proof. We construct a rooted tree T' in the following way. Let $V(T') = S$ and let t_1 be a parent of t_2 in T' if and only if t_1 is a strict ancestor of t_2 in T and the path between t_1 and t_2 in T does not contain any other nodes in S. Since S is LCA-closed, it can be verified that T' is indeed a rooted tree. We then construct a tree decomposition (T', bag') by defining bag' as follows.

$$\mathsf{bag}'(t) = \begin{cases} \mathsf{bag}(t) & \text{if } t \text{ is the root of } T', \\ \mathsf{bag}(t) \cup \mathsf{bag}(\mathsf{parent}_{T'}(t)) & \text{otherwise.} \end{cases}$$

We claim that (T', bag') is a tree decomposition of $\mathsf{torso}_G(\mathsf{bags}(S))$ of width at most $2k + 1$. The vertex condition is trivial, and the connectedness condition follows from the connectedness condition on (T, bag). For the edge condition, consider an edge uv of $\mathsf{torso}_G(\mathsf{bags}(S))$. We have that $u, v \in \mathsf{bags}(S)$ and there exists a path P between u and v in G that is internally disjoint with $\mathsf{bags}(S)$. Pick two nodes $t_u, t_v \in S$ such that $u \in \mathsf{bag}(t_u)$ and $v \in \mathsf{bag}(t_v)$. For each node t on the path between t_u and t_v in T', the bag $\mathsf{bag}(t)$ must contain either u or v, as otherwise, $\mathsf{bag}(t) \subseteq \mathsf{bags}(S)$ would be a separator

between u and v in G disjoint with $\{u, v\}$, contradicting the existence of P. Hence, one of the following must hold.

- Both u and v belong to $\mathsf{bag}(t)$ for some $t \in S$. Then $u, v \in \mathsf{bag}'(t)$, so the edge condition is satisfied for the edge uv.
- We have $u \in \mathsf{bag}(t_1)$, $v \in \mathsf{bag}(t_2)$ for some nodes $t_1, t_2 \in S$ that are adjacent in T'. Without loss of generality, assume that t_1 is the parent of t_2. Then, since $\mathsf{bag}'(t_2) = \mathsf{bag}(t_1) \cup \mathsf{bag}(t_2)$, we infer that $u, v \in \mathsf{bag}'(t_2)$.

We conclude that (T', bag') is indeed a tree decomposition of $\mathsf{torso}_G(\mathsf{bags}(S))$. Since each bag of (T', bag') has size at most $2k + 2$, the proof is finished. □

Lemma 6.20 already shows that every set $W \subseteq V(G)$ in a graph G with $\mathsf{tw}(G) \leq k$ admits a k-closure X of size at most $\mathcal{O}(k \cdot |W|)$. Indeed, consider a tree decomposition $(T_{\mathsf{opt}}, \mathsf{bag}_{\mathsf{opt}})$ of G of optimum width. For each vertex $v \in W$, select into S a node $t \in V(T_{\mathsf{opt}})$ such that $v \in \mathsf{bag}_{\mathsf{opt}}(t)$. Then, take the LCA-closure of S (which increases $|S|$ by a factor of at most 2 by Lemma 2.5) and apply Lemma 6.20.

However, this does not yield the additional condition that X contains only a bounded number of vertices from each subtree rooted at an appendix of T_{pref}. To prove this, we will use the Dealternation Lemma of Bojańczyk and Pilipczuk [2022]. Intuitively, this lemma states that for every tree decomposition (T, bag) of G, there exists an optimum-width tree decomposition $(T_{\mathsf{opt}}, \mathsf{bag}_{\mathsf{opt}})$ that is "well-structured" with respect to (T, bag). The idea will be to apply Lemma 6.20 in the fashion described in the previous paragraph with this well-structured optimum-width tree decomposition.

Dealternation Lemma

We now present a formal version of the Dealternation Lemma. The description follows the exposition in Bojańczyk and Pilipczuk [2022], with some adjustments to the notation. To present the Dealternation Lemma, we need to first define *elimination trees* and *factors*.

The output of the Dealternation Lemma is a tree decomposition of G presented as a so-called *elimination tree*.

Definition 6.21 (Elimination tree)**.** *An elimination tree of G is a rooted tree T with $V(T) = V(G)$ so that if $uv \in E(G)$, then u and v are in an ancestor-descendant relationship in T.*

The following definition shows how to turn an elimination tree of G into a tree decomposition.

Definition 6.22. *Assume that T is an elimination tree of G. A tree decomposition induced by T is the tree decomposition (T, bag), where for each $v \in V(G)$, we set $\mathsf{bag}(v)$ to contain v and each ancestor of v connected by an edge of G to any descendant of v.*

That (T, bag) constructed as in Definition 6.22 is indeed a tree decomposition of G is argued in [Bojańczyk and Pilipczuk, 2022, Section 3]. It is now natural to define the *width* of an elimination tree T as the width of the tree decomposition induced by T. Clearly, each elimination tree has width lower-bounded by $\mathsf{tw}(G)$. On the other hand, every graph G has an elimination tree of width exactly $\mathsf{tw}(G)$ [Bojańczyk and Pilipczuk, 2022, Lemma 3.6].

Let us then define *factors* of trees. Informally, a factor of a rooted tree T is a well-structured "chunk" of T. Formally, a factor of a rooted tree T is a subset of $V(T)$ that is either

- a *tree factor*: a set $\Phi = \mathsf{desc}_T(r)$ for some node $r \in V(T)$, which is called the *root* of Φ,
- a *forest factor*: a set $\Phi = \bigcup_{i=1}^{\ell} \Phi_i$, where $\Phi_1, \ldots, \Phi_\ell$ are tree factors whose roots $r_1, \ldots, r_\ell$ have the same parent, or
- a *context factor*: a set $\Phi = \Phi_1 \setminus \Phi_2$, where Φ_1 is a tree factor and $\Phi_2 \subset \Phi_1$ is a forest factor. The *root* of Φ is the root of Φ_1, while the roots of the trees factors comprising Φ_2 are called the *appendices* of Φ.

Note that every tree factor is also a forest factor.

We can now state the Dealternation Lemma.

Lemma 6.23 (Dealternation Lemma, Bojańczyk and Pilipczuk [2022])**.** *Let (T, bag) be a tree decomposition of G of width at most ℓ. Then there exists an elimination tree F of G of width $\mathsf{tw}(G)$ such that for every node $t \in V(T)$, the set $\mathsf{cmp}(t)$ is a disjoint union of at most $\mathcal{O}(\ell^3)$ factors of F.*

Small Closure Lemma

We now show how the Dealternation Lemma implies the Small Closure Lemma.

Lemma 6.24 (Small Closure Lemma)**.** *Let k and ℓ be integers with $k \leq \ell$ and G be a graph with $\mathsf{tw}(G) \leq k$. Let (T, bag) be a tree decomposition of G of width at most ℓ and $T_{\mathsf{pref}} \subseteq V(T)$ a prefix of T. Then there exists a c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$ with respect to (T, bag), for $c = \mathcal{O}(\ell^4)$.*

Proof. We begin by applying Lemma 6.23 to (T, bag) and getting an elimination tree F of G of width $\mathsf{tw}(G)$, for which $\mathsf{cmp}(t)$ for each $t \in V(T)$ can be decomposed into at most $f(\ell) = \mathcal{O}(\ell^3)$ factors of F. We remind that $V(F) = V(G)$.

Let (F, bag_F) be the tree decomposition induced by F, in particular, where $\mathsf{bag}_F(v)$ for $v \in V(G)$ is defined as the set containing v and every ancestor of v in F incident to an edge whose other endpoint is a descendant of v. As discussed before, (F, bag_F) has width $\mathsf{tw}(G)$. Then let W be the LCA-closure of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ in F. We claim that the set

$$X = \bigcup_{v \in W} \mathsf{bag}_F(v)$$

is an $((\ell+1) \cdot f(\ell))$-small k-closure of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ with respect to (T, bag). It remains to prove this.

Claim 6.25. *The set X is a k-closure of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$.*

Proof of the claim. Since $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}}) \subseteq W$ and $v \in \mathsf{bag}_F(v)$ for each $v \in W$, we have that $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}}) \subseteq W \subseteq X$, as required. It remains to show that $\mathsf{tw}(\mathsf{torso}_G(X)) \leq 2k+1$. Because W is LCA-closed in F, Lemma 6.20 applies to W and the tree decomposition (F, bag_F), finishing the proof. ◁

Claim 6.26. *Let $a \in \mathsf{app}(T_{\mathsf{pref}})$ and let Φ be a factor of F with $\Phi \subseteq \mathsf{cmp}(a)$. Then $|\Phi \cap X| \leq \ell + 1$.*

Proof of the claim. Since a is an appendix of T_{pref}, it follows from the definition of $\mathsf{cmp}(a)$ that $\mathsf{cmp}(a)$ is disjoint with $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$. Thus, Φ is also disjoint with $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$.

First, assume that Φ is a forest factor. Then Φ is downwards closed, meaning that if a vertex belongs to Φ, then all its descendants also belong to Φ. Since W consists of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ and a subset of ancestors of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ in F, it follows that Φ is disjoint with W. Now, for each $v \in W$, the set $\mathsf{bag}_F(v)$ comprises v and some ancestors of v in F. So again, Φ is disjoint with each set $\mathsf{bag}_F(v)$ for $v \in W$ and thus disjoint with X.

Now assume that Φ is a context factor. Recall that $\Phi = \Phi_1 \setminus \Phi_2$, where Φ_1 is a tree factor and $\Phi_2 \subset \Phi_1$ is a forest factor. The appendices of Φ have a common parent, which we

call s. By the disjointness of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ with Φ, we see that each vertex of $\mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ is either outside of Φ_1 or inside some subtree of Φ_2.

Consider $u, v \in \mathsf{bags}_{\mathcal{T}}(T_{\mathsf{pref}})$ such that their LCA is in Φ_1. If either u or v is outside of Φ_1, then their LCA is also outside of Φ_1. Therefore, both u and v belong to Φ_2. In this case, it can be easily seen that their LCA either belongs to Φ_2 (if both u and v are from the same rooted tree of Φ_2) or is equal to s (otherwise). Thus,

$$\Phi_1 \cap W \subseteq \Phi_2 \cup \{s\}.$$

Note that $\Phi_2 \cup \{s\}$ induces a connected subtree of F containing s and some rooted subtrees attached to s.

Again, for each $v \in W$, the set $\mathsf{bag}_F(v)$ comprises v and some ancestors of v. Hence, for $v \in W \setminus \Phi_1$, the set $\mathsf{bag}_F(v)$ is disjoint with Φ. Therefore,

$$\Phi \cap X \subseteq \bigcup_{v \in \Phi_2 \cup \{s\}} \Phi \cap \mathsf{bag}_F(v).$$

Now let $v \in \Phi_2 \cup \{s\}$ and $x \in \Phi \cap \mathsf{bag}_F(v)$. By the definition of bag_F, x is either

- equal to v, in which case $x \in \Phi_2 \cup \{s\}$, so necessarily $x = s$, or
- an ancestor of v that is connected by an edge to a descendant y of v. But since also $x \in \Phi$, we get that x is also an ancestor of s. Also, y is a descendant of s (as y is a descendant of v and v is a descendant of s). We conclude that $x \in \mathsf{bag}_F(s)$.

In both cases we have $x \in \mathsf{bag}_F(s)$ and therefore

$$\Phi \cap X \subseteq \mathsf{bag}_F(s).$$

As (F, bag_F) is a tree decomposition of width $\mathsf{tw}(G) \leq k \leq \ell$, the statement of the claim follows immediately. ◁

Claim 6.27. *Let a be an appendix of T_{pref}. Then $|\mathsf{cmp}(a) \cap X| \leq (\ell + 1) \cdot f(\ell)$.*

Proof of the claim. Because we obtained F by applying the Dealternation Lemma (Lemma 6.23), $\mathsf{cmp}(a)$ can be partitioned into at most $f(\ell)$ disjoint factors of F. By Claim 6.26, each such factor intersects X in at most $\ell + 1$ elements. ◁

The proof of the Small Closure Lemma follows immediately from Claims 6.25 and 6.27. □

6.3.2 Linked closures

Having established the existence of c-small closures for sufficiently large c, we now show a structural result about such closures, the Closure Linkedness Lemma. This corresponds to the d-linkedness of torso tree decompositions defined in Section 5.2, but has a slightly different definition[3]. The proof of the Closure Linkedness Lemma will also be similar to the proof of Lemma 5.10, the main difference being that here we need to be careful to maintain c-smallness.

For the convenience of the reader, let us first restate the Pulling Lemma from Section 5.2. (We will not need the algorithmic part of it in this chapter.)

Lemma 5.8 (Pulling Lemma)**.** *Let G be a graph and $(X, (T, \mathsf{bag}))$ a torso tree decomposition in G. Let (A, S, B) be a separation of G so that there exists a node $r \in V(T)$ so that S is linked into $\mathsf{bag}(r) \cap (S \cup B)$. There exists a torso tree decomposition $((X \cap A) \cup S, (T', \mathsf{bag}'))$ so that*

1. *$T' = T$*
2. *for all $t \in V(T)$, $|\mathsf{bag}'(t)| \leq |\mathsf{bag}(t)|$, and*
3. *$S \subseteq \mathsf{bag}'(r)$.*

Moreover, when G, $(X, (T, \mathsf{bag}))$, (A, S, B), and r are given as inputs, the torso tree decomposition $((X \cap A) \cup S, (T', \mathsf{bag}'))$ can be constructed in $k^{\mathcal{O}(1)}(|V(T)| + m)$ time, where k is the width of $(X, (T, \mathsf{bag}))$.

Then, let $d \colon V(G) \to \mathbb{Z}$ be an arbitrary weight function, and for $S \subseteq V(G)$ denote $d(S) = \sum_{v \in S} d(v)$. We define d-minimal closures.

Definition 6.28 (d-minimal)**.** *Let k and c be integers and $d \colon V(G) \to \mathbb{Z}$ a weight function. Let also G be a graph, $\mathcal{T}$ a tree decomposition of G, and T_{pref} a prefix of $\mathcal{T}$. Then, we say that a c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$ is d-minimal if for every c-small k-closure X' of $\mathsf{bags}(T_{\mathsf{pref}})$, one of the following conditions holds.*

- $|X'| > |X|$
- $|X'| = |X|$ *and* $d(X') \geq d(X)$

[3]The definition used in this chapter could have also been used in Chapter 5, but not the other way around.

Similarly to Chapters 4 and 5, the function d we use will be the vertex-depth function $d_{\mathcal{T}}\colon V(G) \to \mathbb{Z}_{\geq 0}$ of a rooted tree decomposition $\mathcal{T}$ that maps each $v \in V(G)$ to the non-negative integer $d_{\mathcal{T}}(v) = \mathsf{depth}_{\mathcal{T}}(\mathsf{forget}_{\mathcal{T}}(v))$.

We also recall the definition of d-linked from Section 5.2.

Definition 5.9 (d-linked)**.** *Let G be a graph, $A, B \subseteq V(G)$, and $d : V(G) \to \mathbb{Z}$ a weight function. The set A is d-linked into B if for every (A,B)-separator S it holds either that $|S| > |A|$, or that $|S| = |A|$ and $d(S) \geq d(A)$.*

We are now ready to prove the Closure Linkedness Lemma.

Lemma 6.29 (Closure Linkedness Lemma)**.** *Let k and c be integers, and $d\colon V(G) \to \mathbb{Z}$ a weight function. Let G be a graph, $\mathcal{T}$ a tree decomposition of G, T_{pref} a prefix of $\mathcal{T}$, and X a d-minimal c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$. Then for each $C \in \mathsf{cc}(G \setminus X)$, the set $N(C)$ is d-linked into $\mathsf{bags}(T_{\mathsf{pref}})$.*

Proof. Let $W = \mathsf{bags}(T_{\mathsf{pref}})$. For the sake of contradiction, assume we have a component $C \in \mathsf{cc}(G \setminus X)$ such that its neighborhood $N(C)$ is *not* d-linked into W. By Definition 5.9, there exists an $(N(C), W)$-separator S such that either $|S| < |N(C)|$, or $|S| = |N(C)|$ and $d(S) < d(N(C))$. Without loss of generality, assume that S is such a separator with the minimum possible size, which implies that S is linked into both $N(C)$ and W. Naturally, S gives a separation (A, S, B) such that $W \subseteq A \cup S$ and $N(C) \subseteq S \cup B$.

Now, construct a new set X' from X as follows:

$$X' = (X \cap A) \cup S.$$

We claim that X' is also a c-small k-closure of W. Note that since $N(C)$ is contained in X and disjoint from $X \cap A$, we have that $X' \subseteq (X \setminus N(C)) \cup S$. Therefore, $|X'| \leq |X| - |N(C)| + |S|$ and $d(X') \leq d(X) - d(N(C)) + d(S)$. So if $|S| < |N(C)|$, we have $|X'| < |X|$, and if $|S| = |N(C)|$ and $d(S) < d(N(C))$, then $|X'| \leq |X|$ and $d(X') < d(X)$. So provided that X' is indeed a c-small k-closure of W, X cannot be d-minimal, contradicting our assumption.

Claim 6.30. *The set X' is a k-closure of W.*

Proof of the claim. Since W is disjoint with B and X is a k-closure of W, we get that $X' \supseteq W$.

Aiming to use the Pulling Lemma (Lemma 5.8), we let (T_X, bag_X) to be a tree decomposition of $\mathsf{torso}_G(X)$ of width at most $2k+1$. As $N(C)$ is a clique in $\mathsf{torso}_G(X)$, there exists

$r \in V(T_X)$ with $N(C) \subseteq \mathsf{bag}_X(r)$. Because S is linked into $N(C)$, $N(C) \subseteq \mathsf{bag}_X(r)$, and $N(C) \subseteq (S \cup B)$, we get that S is linked into $\mathsf{bag}_X(r) \cap (S \cup B)$, so we satisfy the preconditions of applying the Pulling Lemma with the separation (A, S, B) and the node r. This produces a tree decomposition of $\mathsf{torso}_G(X')$ of width at most $2k+1$. $\lhd$

Claim 6.31. *The set X' is c-small.*

Proof of the claim. Recall that $X \supseteq W$, where $W = \mathsf{bags}(T_{\mathsf{pref}})$. Therefore, as C is a connected component of $G \setminus X$, the entire connected component C must be contained within $\mathsf{cmp}(a)$ for some appendix a of T_{pref}. Hence, $N(C) \subseteq \mathsf{cmp}(a) \cup \mathsf{adh}(a)$. Also, $\mathsf{adh}(a) \subseteq W$. As $(\mathsf{cmp}(a), \mathsf{adh}(a), V(G) \setminus (\mathsf{cmp}(a) \cup \mathsf{adh}(a)))$ is a separation of G, and S is a minimum-size $(N(C), W)$-separator, it follows that $S \subseteq \mathsf{cmp}(a) \cup \mathsf{adh}(a)$. In other words, vertices outside of $\mathsf{cmp}(a) \cup \mathsf{adh}(a)$ are not useful towards the separation of $N(C)$ from W.

Now, for an appendix $a' \neq a$ of T_{pref}, $\mathsf{cmp}(a')$ is disjoint from $\mathsf{cmp}(a) \cup \mathsf{adh}(a)$. This is because $\mathsf{cmp}(a')$ is disjoint from $W \supseteq \mathsf{adh}(a)$ and from $\mathsf{cmp}(a)$ (as a and a' are not in the ancestor-descendant relationship in T). Therefore, S is disjoint with $\mathsf{cmp}(a')$ and thus, by the definition of X',

$$|X' \cap \mathsf{cmp}(a')| = |(X \cap A) \cap \mathsf{cmp}(a')| \leq |X \cap \mathsf{cmp}(a')| \leq c.$$

Hence, the smallness condition is not violated for the appendix a'.

In order to prove that the smallness condition is not violated for the appendix a, we observe that $N(C) \cap W \subseteq S$ (as S separates $N(C)$ from W), so also $N(C) \cap \mathsf{adh}(a) \subseteq S$. Therefore, $|N(C) \cap \mathsf{adh}(a)| \leq |S \cap \mathsf{adh}(a)|$, and we get that

$$|N(C) \cap \mathsf{cmp}(a)| = |N(C)| - |N(C) \cap \mathsf{adh}(a)| \geq |S| - |S \cap \mathsf{adh}(a)| = |S \cap \mathsf{cmp}(a)|.$$

Now, as $X' \subseteq (X \setminus N(C)) \cup S$ and $N(C) \subseteq X$, we infer that

$$|X' \cap \mathsf{cmp}(a)| \leq |X \cap \mathsf{cmp}(a)| - |N(C) \cap \mathsf{cmp}(a)| + |S \cap \mathsf{cmp}(a)| \leq |X \cap \mathsf{cmp}(a)| \leq c. \lhd$$

By Claims 6.30 and 6.31 we infer that X' is a c-small k-closure of W. This is a contradiction to the d-minimality of X. $\square$

The $d_{\mathcal{T}}$-linkedness of a closure will be used in Section 6.5 to bound the width of the tree decomposition resulting from the refinement operation, similarly as it was used in Section 5.2. The utility of Lemma 6.29 is that it suggests a method for finding c-small

k-closures closures that satisfy this $d_{\mathcal{T}}$-linkedness property. In particular, it is sufficient to find a $d_{\mathcal{T}}$-minimal c-small k-closure. Note that this is analogous to finding minimum splits in Chapter 4.

6.3.3 Blockages and explored nodes

We then introduce concepts of *blockages* and *explored nodes*. These are related to the concept of editable nodes used in Chapter 4. Blockages and explored nodes will be used in the refinement operation in Section 6.5, and in Section 6.4 we give a data structure for computing them along with k-closures.

Let us then define *blockages*. We assume that $\mathcal{T} = (T, \mathsf{bag})$ is a tree decomposition of a graph G, and X is a k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$ for a prefix T_{pref} of T and some integer k.

Definition 6.32. *A node $t \in V(T) \setminus T_{\mathsf{pref}}$ is a blockage in $\mathcal{T}$ with respect to T_{pref} and X if no strict ancestor of t is a blockage, and either*

- $\mathsf{adh}(t) \subseteq N[C]$ *for some component $C \in \mathsf{cc}(G \setminus X)$ that intersects $\mathsf{adh}(t)$, in which case t is called a component blockage covered by C, or*
- $\mathsf{adh}(t) \subseteq X$ *and $\mathsf{adh}(t)$ is a clique in $\mathsf{torso}_G(X)$, in which case t is called a clique blockage.*

Note that the adhesion of a component blockage intersects exactly one component $C \in \mathsf{cc}(G \setminus X)$.

Since $\mathcal{T}$, T_{pref} and X will usually be known from the context, we will usually just say that t is a blockage. We denote the set of blockages by $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$.

By definition, no two blockages are in an ancestor-descendant-relationship, and no blockage is in T_{pref}, implying that there exists a prefix $T'_{\mathsf{pref}} \supseteq T_{\mathsf{pref}}$ so that $\mathsf{Blockages}(T_{\mathsf{pref}}, X) = \mathsf{app}(T'_{\mathsf{pref}})$. The nodes in T'_{pref} are called *explored*, and the set of the explored nodes is denoted by $T'_{\mathsf{pref}} = \mathsf{Expl}(T_{\mathsf{pref}}, X)$. The other nodes are called *unexplored*.

6.4 Computing closures

In this section we show that the objects defined in the previous section can be computed efficiently in a tree decomposition that is changing dynamically under prefix-rebuilding

updates. At this point, we fix $d_{\mathcal{T}}\colon V(G) \to \mathbb{Z}$ to be the vertex-depth function $d_{\mathcal{T}}(v) = \mathsf{depth}_{\mathcal{T}}(\mathsf{forget}_{\mathcal{T}}(v))$ of the annotated tree decomposition $\mathcal{T}$ that we are maintaining. In particular, we will give a prefix-rebuilding data structure, that supports an operation that given a prefix T_{pref}, returns a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$, for some fixed integers c and k, the graph $\mathsf{torso}_G(X)$ and the sets $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ and $\mathsf{Expl}(T_{\mathsf{pref}}, X)$.

6.4.1 Closure automaton

As the first ingredient of our data structure, we give an automaton for computing small closures within subtrees of a tree decomposition. Recall that when given a prefix T_{pref}, we are interested in closures X that for each appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$, contain at most c vertices from $\mathsf{cmp}(a)$, but all vertices from $\mathsf{adh}(a)$ because $\mathsf{adh}(a) \subseteq \mathsf{bags}(T_{\mathsf{pref}})$. In this subsection, we give a tree decomposition automaton for understanding all non-isomorphic ways how the selection of the c vertices can affect the eventual graph $\mathsf{torso}_G(X)$, and for optimizing this selection according to $d_{\mathcal{T}}$-minimality.

Recall the definitions on boundaried graphs, boundaried tree decompositions, and tree decomposition automata introduced in Subsection 6.2.2. We start with a few more definitions.

Let G be a boundaried graph. We say that two sets of non-boundary vertices $Y, Z \subseteq V(G) \setminus \partial G$ are *torso-equivalent* if there is an isomorphism between $\mathsf{torso}_G(Y \cup \partial G)$ and $\mathsf{torso}_G(Z \cup \partial G)$ that fixes every vertex of ∂G. Note that being torso-equivalent is an equivalence relation and for every integer c, let $\sim_{c,G}$ be the restriction of this equivalence relation to subsets of $V(G) \setminus \partial G$ of size at most c. Note $\sim_{c,G}$ has at most $2^{\mathcal{O}((c+|\partial G|)^2)}$ equivalence classes.

Suppose further that $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ is a boundaried tree decomposition of G. We define the vertex-depth function $d_{\mathcal{T}}\colon V(G) \to \mathbb{Z}_{\geq 0}$ for boundaried graphs similarly as for graphs: $d_{\mathcal{T}}(v) = \mathsf{depth}_{\mathcal{T}}(\mathsf{forget}_{\mathcal{T}}(v))$, i.e., $d_{\mathcal{T}}(v)$ is depth of the closest node to the root that contains v. For a set $Y \subseteq V(G)$, we have $d_{\mathcal{T}}(Y) = \sum_{v \in Y} d_{\mathcal{T}}(v)$. Recalling that there is a total order $\preceq$ on the vertices of G (inherited from $\mathbb{Z}_{\geq 1}$, as we assume that $V(G) \subset \mathbb{Z}_{\geq 1}$), subsets of $V(G)$ can be compared as follows. We define a total order $\preceq_{\mathcal{T}}$ on sets of vertices by setting $Y \preceq_{\mathcal{T}} Y'$ for $Y, Y' \subseteq V(G)$ if

- $d_{\mathcal{T}}(Y) < d_{\mathcal{T}}(Y')$, or
- $d_{\mathcal{T}}(Y) = d_{\mathcal{T}}(Y')$ and Y is lexicographically not larger than Y' with respect to $\preceq$.

For a non-empty set of subsets $\mathcal{S}$ of $V(G)$, we let $\min_{\mathcal{T}} \mathcal{S}$ be the $\preceq_{\mathcal{T}}$-smallest element of $\mathcal{S}$. This allows us to define the *c-small torso representatives* as

$$\mathsf{reps}^c(G,\mathcal{T}) = \{(d_{\mathcal{T}}(\min_{\mathcal{T}}\mathcal{K}), \mathsf{torso}_G(\min_{\mathcal{T}}\mathcal{K} \cup \partial G)) \mid \mathcal{K} \text{ is an equivalence class of } \sim_{c,G}\}.$$

Note that each member of an equivalence class of $\sim_{c,G}$ has the same size, so at this point we do not optimize for the size even though it will later be needed to find $d_{\mathcal{T}}$-minimal closures. The set $\mathsf{reps}^c(G,\mathcal{T})$ depends on both the boundaried graph G and its boundaried tree decomposition $\mathcal{T}$. Also, the size of $\mathsf{reps}^c(G,\mathcal{T})$ is equal to the number of equivalence classes of $\sim_{c,G}$, which, as noted, is at most $2^{\mathcal{O}((c+|\partial G|)^2)}$.

The closure automaton is provided in the following lemma.

Lemma 6.33. *For every pair of integers c, ℓ there is a tree decomposition automaton $\mathcal{R} = \mathcal{R}_{c,\ell}$ of width ℓ so that for any graph G and its annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, the run of $\mathcal{R}$ on $\mathcal{T}$ satisfies*

$$\rho_{\mathcal{R}}(x) = \mathsf{reps}^c(G_x, \mathcal{T}_x) \qquad \text{for all } x \in V(T).$$

The evaluation time of $\mathcal{R}$ is $2^{\mathcal{O}((c+\ell)^2)}$.

Proof. For the state set Q of $\mathcal{R}$ we take the set of all sets of pairs of the form (p, H), where $p \in \mathbb{Z}_{\geq 0}$ and H is a graph on at most $c + \ell + 1$ vertices. The final states of $\mathcal{R}$ are irrelevant for the lemma statement, hence we can set $F = \emptyset$. As for the initial mapping ι, for a boundaried graph H on at most $\ell + 1$ vertices we can set $\iota(H) = \mathsf{reps}^c(H, \mathcal{T}_0)$, where $\mathcal{T}_0$ is the trivial one-node boundaried tree decomposition of H in which all vertices and edges are put in the root bag. It is straightforward to see that $\iota(H)$ can be computed in time $2^{\mathcal{O}((c+\ell)^2)}$ directly from the definition.

It remains to define the transition mapping δ. For this, it suffices to prove the following. Suppose $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ is a boundaried tree decomposition of a boundaried graph G and x is a node of T with children y and z. Then knowing

$$R_y = \mathsf{reps}^c(G_y, \mathcal{T}_y), \qquad R_z = \mathsf{reps}^c(G_z, \mathcal{T}_z),$$

as well as $\mathsf{bag}(x)$, $\mathsf{edges}(x)$, and the adhesions of x, y, z, one can compute

$$R_x = \mathsf{reps}^c(G_x, \mathcal{T}_x)$$

in time $2^{\mathcal{O}((c+\ell)^2)}$. Formally, we would also need such an argument for the case when x has only one child y, but this follows from the argument for the case of two children

by considering a dummy second child z with an empty bag. So we focus only on the two-children case.

For any pair of sets $Y \subseteq \mathsf{cmp}(y)$ and $Z \subseteq \mathsf{cmp}(z)$, define $G_x(Y,Z)$ to be the graph with the set of vertices $Y \cup Z \cup \mathsf{bag}(x)$ and the edge set consisting of the union of the edges of the graphs $\mathsf{torso}_{G_y}(Y \cup \mathsf{adh}(y))$ and $\mathsf{torso}_{G_z}(Z \cup \mathsf{adh}(z))$, and the set $\mathsf{edges}(x)$. The following claim is straightforward.

Claim 6.34. *For any triple of sets $X \subseteq \mathsf{bag}(x) \setminus \mathsf{adh}(x)$, $Y \subseteq \mathsf{cmp}(y)$ and $Z \subseteq \mathsf{cmp}(z)$, we have*

$$\mathsf{torso}_{G_x}(X \cup Y \cup Z \cup \mathsf{adh}(x)) = \mathsf{torso}_{G_x(Y,Z)}(X \cup Y \cup Z \cup \mathsf{adh}(x)).$$

To compute R_x, we first construct a family of candidates C as follows. Consider every pair of pairs $(p_y, H_y) \in R_y$ and $(p_z, H_z) \in R_z$, and every $X \subseteq \mathsf{bag}(x) \setminus \mathsf{adh}(x)$. Let $Y = V(H_y) \setminus \mathsf{adh}(y)$ and $Z = V(H_z) \setminus \mathsf{adh}(z)$, and note that the graph $G_x(Y,Z)$ is the union of graphs H_y, H_z, and $(\mathsf{bag}(x), \mathsf{edges}(x))$. If $|X \cup Y \cup Z| \leq c$, then we add to C the pair

$$(p_y + p_z + |Y| + |Z|, \mathsf{torso}_{G_x(Y,Z)}(X \cup Y \cup Z \cup \mathsf{adh}(x))).$$

Otherwise, if $|X \cup Y \cup Z| > c$, no pair is added to C. The first coordinate of the pair added to C is equal to $d_{T_x}(X \cup Y \cup Z)$ because $X \subseteq \mathsf{bag}(x)$ and $Y, Z \subseteq V(G_x) \setminus \mathsf{bag}(x)$. By Claim 6.34, the second coordinate of the pair added to C is equal to the graph $\mathsf{torso}_{G_x}(X \cup Y \cup Z \cup \mathsf{adh}(x))$.

Further, we have

$$|C| \leq |R_y| \cdot |R_z| \cdot 2^{\ell+1} \leq 2^{\mathcal{O}((c+\ell)^2)},$$

and C can be computed in time $2^{\mathcal{O}((c+\ell)^2)}$.

Next, the candidates are filtered as follows. As long as in C there is are distinct pairs (p,H) and (p',H') such that H and H' are isomorphic by an isomorphism that fixes $\mathsf{adh}(x)$, we remove the pair that has the larger first coordinate. If both pairs have the same first coordinate, remove the one where the vertex set of the second coordinate is larger in $\preceq$. Clearly, this filtering procedure can be performed exhaustively in time $2^{\mathcal{O}((c+\ell)^2)}$.

Thus, after filtering, all second coordinates of the pairs in C are pairwise non-equivalent in $\sim_{c,G_x}$. It is now straightforward to see using a simple exchange argument that R_x is equal to C after the filtering. This constitutes the definition and the algorithm computing the transition function δ. □

6.4.2 Data structure for closures

We are now ready to state the prefix-rebuilding data structure for computing closures. Recall the definitions of blockages and explored nodes from Subsection 6.3.3.

Lemma 6.35. *For all integers c, ℓ, k with $\ell \geq k$ and $\ell, c \leq k^{\mathcal{O}(1)}$, there exists an ℓ-prefix-rebuilding data structure with overhead $2^{\mathcal{O}((c+\ell)^2)}$, that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a dynamic graph G, and additionally supports the following query.*

- $\mathsf{Closure}(T_{\mathsf{pref}})$*: Given a prefix T_{pref} of T, returns either* No closure *if there is no c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$, or*
  - *a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$,*
  - *the graph $\mathsf{torso}_G(X)$,*
  - *the sets $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ and $\mathsf{Expl}(T_{\mathsf{pref}}, X)$, and*
  - *for all $t \in \mathsf{Blockages}(T_{\mathsf{pref}}, X)$ the information whether t is a clique blockage or a component blockage.*

  This operation runs in time $2^{\mathcal{O}(k\cdot(c+\ell)^2)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$.

The rest of this section is dedicated to the proof of Lemma 6.35.

For the proof, we fix the following two tree decomposition automata.

- $\mathcal{R} = \mathcal{R}_{c,\ell}$ is the closure automaton for parameters c and ℓ, provided by Lemma 6.33.
- $\mathcal{BK} = \mathcal{BK}_{2k+1,c+\ell}$ is the Bodlaender-Kloks nondeterministic automaton for parameters $2k+1$ and $c+\ell$, provided by Lemma 6.13.

Let $\mathcal{BK} = (Q, F, \iota, \delta)$. Recall that $\mathcal{BK}$ is nondeterministic, $|Q| \leq 2^{\mathcal{O}((k+\log(c+\ell))\cdot(c+\ell)^2)} \leq 2^{\mathcal{O}(k\cdot(c+\ell)^2)}$, Q can be computed in time $2^{\mathcal{O}(k\cdot(c+\ell)^2)}$, and membership in F, ι, or δ for relevant objects can be decided in time $2^{\mathcal{O}(k\cdot(c+\ell)^2)}$.

Our data structure simply consists of the data structure provided by Lemma 6.15 for the automaton $\mathcal{R}$. Thus, the initialization time is $2^{\mathcal{O}((c+\ell)^2)} \cdot |V(T)|$ and the update time is $2^{\mathcal{O}((c+\ell)^2)} \cdot |\overline{u}|$ as requested. It remains to implement the operation $\mathsf{Closure}(T_{\mathsf{pref}})$. For this, we may assume that the stored annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ is labeled with the run $\rho_{\mathcal{R}}$ of $\mathcal{R}$ on $(T, \mathsf{bag}, \mathsf{edges})$. This means that for every $x \in V(T)$, we have access to $\mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$.

Consider a query $\mathsf{Closure}(T_{\mathsf{pref}})$. We break answering this query into two steps. In the first step, we compute a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$, together with

$\mathsf{torso}_G(X)$. This will take time $2^{\mathcal{O}(k(c+\ell)^2)} \cdot |T_{\mathsf{pref}}|$. In the second step, we find the sets $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ and $\mathsf{Expl}(T_{\mathsf{pref}}, X)$. This will take time $2^{\mathcal{O}((c+\ell)^2)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$.

Finding the closure and its torso

For brevity, let $A = \mathsf{app}(T_{\mathsf{pref}})$ be the set of appendices of T_{pref}. For $a \in A$, let

$$R(a) = \mathsf{reps}^c(T_a, \mathsf{bag}_a, \mathsf{edges}_a)$$

and recall that $R(a)$ for all $a \in A$ is stored in the data structure. Let Λ be the set of all mappings λ with domain A such that $\lambda(a) \in R(a)$ for all a. For $\lambda \in \Lambda$ and $a \in A$, let $d^\lambda(a)$ and $H^\lambda(a)$ be the first, respectively the second coordinate of $\lambda(a)$, and let $s^\lambda(a) = |V(H^\lambda(a)) \setminus \mathsf{adh}(a)|$. For $\lambda \in \Lambda$ we define H^λ to be the graph with

$$V(H^\lambda) = \mathsf{bags}(T_{\mathsf{pref}}) \cup \bigcup_{a \in A} V(H^\lambda(a)) \text{ and}$$
$$E(H^\lambda) = \bigcup_{t \in T_{\mathsf{pref}}} \mathsf{edges}(t) \cup \bigcup_{a \in A} E(H^\lambda(a)),$$

and s^λ and d^λ to be the integers

$$s^\lambda = \sum_{a \in A} s^\lambda(a) \text{ and}$$
$$d^\lambda = \sum_{a \in A} \left(d^\lambda(a) + \mathsf{depth}_T(a) \cdot s^\lambda(a) \right).$$

By the definition of reps^c, we have that

- $H^\lambda = \mathsf{torso}_G(V(H^\lambda))$,
- $s^\lambda = |V(H^\lambda)| - |\mathsf{bags}(T_{\mathsf{pref}})|$, and
- $d^\lambda = d_{\mathcal{T}}(V(H^\lambda)) - d_{\mathcal{T}}(\mathsf{bags}(T_{\mathsf{pref}}))$

for all $\lambda \in \Lambda$.

Further, for each $a \in A$, the set $R(a)$ comprises all possible non-isomorphic torsos that can be obtained by picking at most c vertices within $\mathsf{cmp}(a)$, and with each possible torso $R(a)$ stores a realization with the least possible total depth. This immediately implies the following statement.

Claim 6.36. *Let $\lambda \in \Lambda$ be such that the treewidth of H^λ is at most $2k+1$ and, among such mappings λ, s^λ is minimum, and among those d^λ is minimum. Then $V(H^\lambda)$ is a*

$d_{\mathcal{T}}$-minimal c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$*. Further, if no such* λ *exists, then there is no c-small k-closure of* $\mathsf{bags}(T_{\mathsf{pref}})$*.*

By Claim 6.36, it suffices to find $\lambda \in \Lambda$ that primarily minimizes s^λ and secondarily d^λ such that H^λ has treewidth at most $2k+1$, or conclude that no such λ exists. Indeed, then we can output $X = V(H^\lambda)$ and $\mathsf{torso}_G(X) = H^\lambda$ as the output to the query. Note here that once a mapping λ as above is found, one can easily construct H^λ in time $(\ell+c)^{\mathcal{O}(1)} \cdot |T_{\mathsf{pref}}|$ right from the definition. Informally speaking, to find a suitable λ we analyze the possible runs of the Bodlaender-Kloks automaton $\mathcal{BK}$ on the natural tree decomposition of H^λ inherited from T_{pref}, for different choices of λ. More formally, this is done as follows.

Let $S = T_{\mathsf{pref}} \cup A$. For $x \in S$ let $S_x = \mathsf{desc}_T(x) \cap S$, $T_{\mathsf{pref},x} = \mathsf{desc}_T(x) \cap T_{\mathsf{pref}}$, $A_x = \mathsf{desc}_T(x) \cap A$, and Λ_x be defined just like Λ, but for domain A_x instead of A. For $\lambda \in \Lambda_x$, define H^λ_x to be the graph with

$$V(H^\lambda_x) = \mathsf{bags}(T_{\mathsf{pref},x}) \cup \bigcup_{a \in A_x} V(H^\lambda(a)) \text{ and}$$
$$E(H^\lambda_x) = \bigcup_{t \in T_{\mathsf{pref},x}} \mathsf{edges}(t) \cup \bigcup_{a \in A_x} E(H^\lambda(a)) \setminus \binom{\mathsf{adh}(x)}{2},$$

and s^λ_x and d^λ_x to be the integers

$$s^\lambda_x = \sum_{a \in A_x} s^\lambda(a) \text{ and}$$
$$d^\lambda_x = \sum_{a \in A_x} \left(d^\lambda(a) + (\mathsf{depth}_T(a) - \mathsf{depth}_T(x)) \cdot s^\lambda(a) \right).$$

We treat H^λ_x as a boundaried graph with boundary $\mathsf{adh}(x)$. Then, H^λ_x has a boundaried tree decomposition $(T^\lambda_x, \mathsf{bag}^\lambda_x, \mathsf{edges}^\lambda_x)$ naturally inherited from $(T, \mathsf{bag}, \mathsf{edges})$ as

- $T^\lambda_x = T[S_x]$,
- $\mathsf{bag}^\lambda_x(y) = \mathsf{bag}(y)$ for all $y \in T_{\mathsf{pref},x}$, and $\mathsf{bag}^\lambda_x(a) = V(H^\lambda(a))$ for all $a \in A_x$, and
- $\mathsf{edges}^\lambda_x(y) = \mathsf{edges}(y) \cup \bigcup_{a \in A_y} E(H^\lambda(a)) \cap \left(\binom{\mathsf{bag}(y)}{2} \setminus \binom{\mathsf{adh}(y)}{2} \right)$ for all $y \in T_{\mathsf{pref},x}$, and $\mathsf{edges}^\lambda_x(a) = E(H^\lambda(a)) \setminus \binom{\mathsf{adh}(a)}{2}$ for all $a \in A_x$.

Note that the width of this tree decomposition is at most $c + \ell$.

Let $\overline{\mathbb{Z}} = \mathbb{Z}_{\geq 0} \cup \{+\infty\}$. Let us use a total order on pairs in $\overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$ where we first compare the first elements and if they are equal the second elements. In a bottom-up fashion, for

every $x \in S$ we compute the mapping $\zeta_x \colon Q \times 2^{\binom{\mathsf{adh}(x)}{2}} \to \overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$ defined as follows: for $q \in Q$ and $W \subseteq \binom{\mathsf{adh}(x)}{2}$, $\zeta_x(q, W)$ is the minimum value of $(s_x^\lambda, d_x^\lambda)$ among $\lambda \in \Lambda_x$ such that

- $\mathcal{BK}$ has a run on $(T_x^\lambda, \mathsf{bag}_x^\lambda, \mathsf{edges}_x^\lambda)$ in which x is labeled with q, and
- $\bigcup_{a \in A_x} E(H^\lambda(a)) \cap \binom{\mathsf{adh}(x)}{2} = W$.

In case there is no λ as above, we set $\zeta_x(q, W) = (+\infty, +\infty)$.

We now argue that the mappings ζ_x can be computed in a bottom-up manner. This follows from the following rules, whose correctness is straightforward.

- For every $a \in A$, $\zeta_a(q, W)$ is the minimum pair (s, d) such that there is $(d, H) \in R(a)$ with the following properties: $\left(H \setminus \binom{\mathsf{adh}(a)}{2}, q\right) \in \iota$, $E(H) \cap \binom{\mathsf{adh}(a)}{2} = W$, and $|V(H) \setminus \mathsf{adh}(a)| = s$.
- For every $x \in T_{\mathsf{pref}}$ with no children, $\zeta_x(q, W) = (0, 0)$ if $(G_x, q) \in \iota$ and $W = \emptyset$, and $\zeta_x(q, W) = (+\infty, +\infty)$ otherwise.
- For every $x \in T_{\mathsf{pref}}$ with one child y, $\zeta_x(q, W)$ is the minimum (s, d) such that the following holds. There exist $q' \in Q$ and $W' \subseteq \binom{\mathsf{adh}(y)}{2}$ with $(s', d') = \zeta_y(q', W')$ such that

$$\left(\left(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \emptyset, \mathsf{edges}(x) \cup W' \setminus \binom{\mathsf{adh}(x)}{2}, q', \bot\right), q\right) \in \delta,$$
$$W = W' \cap \binom{\mathsf{adh}(x)}{2},$$
$$s = s', \text{ and}$$
$$d = d' + s'.$$

  If there are no q', W' as above, then $\zeta_x(q, W) = (+\infty, +\infty)$.

- For every $x \in T_{\mathsf{pref}}$ with two children y and z, $\zeta_x(q, W)$ is the minimum (s, d) such that the following holds. There exist $q', q'' \in Q$, $W' \subseteq \binom{\mathsf{adh}(y)}{2}$, and $W'' \subseteq \binom{\mathsf{adh}(z)}{2}$ with $(s', d') = \zeta_y(q', W')$ and $(s'', d'') = \zeta_z(q'', W'')$ such that

$$\left(\left(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \mathsf{adh}(z), \mathsf{edges}(x) \cup W' \cup W'' \setminus \binom{\mathsf{adh}(x)}{2}, q', q''\right), q\right) \in \delta,$$
$$W = (W' \cup W'') \cap \binom{\mathsf{adh}(x)}{2},$$
$$s = s' + s'', \text{ and}$$
$$d = d' + d'' + s' + s''.$$

If there are no q', q'', W', W'' as above, then $\zeta_x(q, W) = (+\infty, +\infty)$.

Using the rules above, all mappings ζ_x for $x \in S$ can be computed in total time $2^{\mathcal{O}(k(\ell+c)^2)} \cdot |T_{\mathsf{pref}}|$, because $|Q| \leq 2^{\mathcal{O}(k(\ell+c)^2)}$ and the evaluation time of $\mathcal{BK}$ is $2^{\mathcal{O}(k(\ell+c)^2)}$.

By the properties of $\mathcal{BK}$ asserted in Lemma 6.13, the minimum (s^λ, d^λ) among those $\lambda \in \Lambda$ for which H^λ has treewidth at most k is equal to $\min_{q \in F} \zeta_r(q, \emptyset)$, where r is the root of T. The latter minimum can be computed in time $2^{\mathcal{O}(k(\ell+c)^2)}$ knowing ζ_r. Finally, to find $\lambda \in \Lambda$ witnessing the minimum, it suffices to retrace the dynamic programming in the standard way. That is, when computing mappings ζ_x, for every computed value of $\zeta_x(q, W)$ we memorize how this value was obtained. After finding $q \in F$ that minimizes $\zeta_r(q, \emptyset)$ we recursively retrace how the value of $\zeta_r(q, \emptyset)$ was obtained along S in a top-down manner, up to values computed in the nodes of A. The ways in which these values were obtained gives us the mapping λ. This concludes the construction of the closure $X = V(H^\lambda)$ and the graph $\mathsf{torso}_G(X) = H^\lambda$.

Finding blockages

Having constructed $X = V(H^\lambda)$ together with $\mathsf{torso}_G(X) = H^\lambda$, we proceed to finding the sets $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ and $\mathsf{Expl}(T_{\mathsf{pref}}, X)$. We may assume that every vertex of X has been marked as belonging to X (which can be done after computing X in time $\mathcal{O}(|X|) \leq (c+\ell)^{\mathcal{O}(1)} \cdot |T_{\mathsf{pref}}|$), hence checking whether a given vertex belongs to X can be done in constant time. We also recall that by Lemma 2.10, checking whether two vertices are adjacent in H^λ can be done in time $\mathcal{O}(k)$.

First, we observe that for every node $x \in V(T) \setminus T_{\mathsf{pref}}$, we can efficiently find out the information about the behavior of X in the subtree rooted at x. Denote $X_x = X \cap \mathsf{cmp}(x)$.

Claim 6.37. *Given a node $x \in V(T) \setminus T_{\mathsf{pref}}$, one can compute X_x and $\mathsf{torso}_{G_x}(X_x \cup \mathsf{adh}(x))$ in time $2^{\mathcal{O}((c+\ell)^2)}$.*

Proof of the claim. For every $(d, H) \in \mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$, call H a *candidate* if $V(H) \setminus \mathsf{adh}(x) \subseteq X$. Note that we can find all candidates in time $(c+\ell)^{\mathcal{O}(1)} \cdot$

$|\mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)| \leq 2^{\mathcal{O}((c+\ell)^2)}$ by inspecting all elements of $\mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$ one by one. Now, because there exists exactly one candidate with $V(H) \setminus \mathsf{adh}(x) = X_x$, we have that $\mathsf{torso}_{G_x}(X_x \cup \mathsf{adh}(x))$ is equal to the largest (in terms of the number of vertices) candidate. $\lhd$

Next, for a node $x \in V(T)$, we define $\mathsf{profile}(x) \subseteq \binom{\mathsf{bag}(x)}{2}$ to be the set comprising of all pairs $\{u, v\} \subseteq \mathsf{bag}(x)$ such that in G there is path connecting u and v that is internally disjoint with X. (Note that this definition concerns u and v both belonging and not belonging to X.) Note that if $x \in T_{\mathsf{pref}}$, we have $\mathsf{bag}(x) \subseteq X$ and $\mathsf{profile}(x)$ consists of all edges of $\mathsf{torso}_G(X)$ with both endpoints in $\mathsf{bag}(x)$. Consequently, for such x we can compute $\mathsf{profile}(x)$ in time $\ell^{\mathcal{O}(1)}$.

Next, we show that knowing the profile of a parent we can compute the profile of a child.

Claim 6.38. *Suppose x is the parent of y in T and $y \notin T_{\mathsf{pref}}$. Then given $\mathsf{profile}(x)$, the set $\mathsf{profile}(y)$ can be computed in time $2^{\mathcal{O}((c+\ell)^2)}$.*

Proof of the claim. Let J be the graph on vertex set $\mathsf{bag}(y)$ whose edge set is the union of

- $\mathsf{edges}(y)$,
- all edges present in $\mathsf{profile}(x)$ with both endpoints in $\mathsf{adh}(x)$, and
- all edges present in $\mathsf{torso}_{G_z}(X_z \cup \mathsf{adh}(z))$ that have both endpoints in $\mathsf{adh}(z)$, for every child z of y.

Note that J can be constructed in time $2^{\mathcal{O}((c+\ell)^2)}$ using Claim 6.37. Now, it is straightforward to see that $\mathsf{profile}(y)$ consists of all pairs $\{u, v\} \subseteq \mathsf{bag}(y)$ such that in J there is a path connecting u and v that is internally disjoint with $X \cap \mathsf{bag}(y)$. Using this observation, $\mathsf{profile}(y)$ can be now constructed in time $\ell^{\mathcal{O}(1)}$, because J has at most $\ell + 1$ vertices. $\lhd$

Having established Claim 6.38, we can finally describe the procedure that finds the blockages and explored nodes. The procedure inspects the appendices $a \in A$ one by one, and upon inspecting a it finds all blockages that are descendants of a. To this end, we start a depth-first search in T_a from a. At all times, together with the node $x \in V(T_a)$ which is currently processed in the search, we also store $\mathsf{profile}(x)$. Initially, $\mathsf{profile}(a)$ can be computed using Claim 6.38, where the profile of the parent of a, which belongs to T_{pref}, can be computed in time $\ell^{\mathcal{O}(1)}$, as argued. When the search enters a node y from its parent x, $\mathsf{profile}(y)$ can be computed from $\mathsf{profile}(x)$ using Claim 6.38 again. Observe that knowing $\mathsf{profile}(x)$ and that no ancestor of x is a blockage, it can be determined in time $(c + \ell)^{\mathcal{O}(1)}$ whether x is a blockage as follows.

- If $\mathsf{adh}(x) \subseteq X$ and $\mathsf{profile}(x) \supseteq \binom{\mathsf{adh}(x)}{2}$, then x is a clique blockage.
- If there exists $u \in \mathsf{adh}(x) \setminus X$ such that $\{u, v\} \in \mathsf{profile}(x)$ for all $v \in \mathsf{adh}(x) \setminus \{u\}$, then x is a component blockage.

Consequently, if x is a blockage, we output x and do not pursue the search further. Otherwise, if x is not a blockage, the search recurses to the children of x.

This procedure outputs all blockages and the information whether they are clique blockages or component blockages. By furthermore outputting all the nodes entered in the depth-first search that are not blockages, and the prefix T_{pref}, we can also output the set of explored nodes $\mathsf{Expl}(T_{\mathsf{pref}}, X)$. The total number of nodes visited by the search is $|\mathsf{Expl}(T_{\mathsf{pref}}, X) \cup \mathsf{Blockages}(T_{\mathsf{pref}}, X)| \leq \mathcal{O}(|\mathsf{Expl}(T_{\mathsf{pref}}, X)|)$, and for each node we use time $2^{\mathcal{O}((c+\ell)^2)}$, so that total running time is $2^{\mathcal{O}((c+\ell)^2)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$.

6.5 Refinement operation

In this section we introduce the *refinement operation*, which is the main ingredient of our dynamic algorithm for maintaining small-width tree decompositions. In particular, the refinement operation will be used for controlling the width of the decomposition, and in Section 6.6 we further build on it to also control the height.

Throughout this section, we denote by G the n-vertex dynamic graph we are maintaining, and assume that the treewidth of G is always at most k (this assumption will be partially lifted later). The goal is to maintain an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $\ell = 6k + 5$ (we usually write $\mathcal{T} = (T, \mathsf{bag})$ when we are not using the edges function). As a result of the edge addition operation, the width of $\mathcal{T}$ can intermittently grow to $6k + 6$.

The refinement operation takes as an input a prefix T_{pref} of $\mathcal{T}$ that contains all bags of size more than $6k + 6$. It transforms $\mathcal{T}$ into a tree decomposition $\mathcal{T}'$ that has width at most $6k+5$. The high-level idea of this transformation is to first compute a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$, then compute a logarithmic-depth tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$ of width at most $6k+5$, and then attach tree decompositions $\mathcal{T}^C$ of components $C \in \mathsf{cc}(G \setminus X)$ to $\mathcal{T}^X$ similarly as in the improvement operation of Section 5.2. However, to control the (amortized) running time of the operation and the exact properties of $\mathcal{T}'$, we will need to carefully implement and analyze the operation. Some of the properties of the refinement operation will be used in Section 6.6 for designing a height-reduction scheme to keep the height of $\mathcal{T}$ bounded by $2^{\mathcal{O}(k \log k \sqrt{\log n \log \log n})}$.

This section is organized as follows. First, in Subsection 6.5.1 we introduce our potential function for the analysis of the amortized running time of the refinement operation. Then, in Subsection 6.5.2 we describe how the decompositions $\mathcal{T}^C$ are constructed for components $C \in \mathsf{cc}(G \setminus X)$, given a closure X. In Subsection 6.5.3 we describe how the decompositions $\mathcal{T}^C$ are grouped together based on the neighborhoods $N(C)$. Finally, in Subsection 6.5.4 we give a formal statement of the refinement operation and finish the description and analysis of it.

6.5.1 Potential function

The (amortized) running time of the refinement operation is analyzed with a potential function we introduce now. This potential function generalizes some ideas of the potential function used in Section 4.3, but will be more complicated.

For a node t of a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$, we define its potential by the formula

$$\Phi_{\ell,\mathcal{T}}(t) = g_\ell(|\mathsf{bag}(t)|) \cdot \mathsf{hgt}_T(t),$$

where

$$g_\ell(x) = (84(\ell+1))^{x+1} \qquad \text{for every } x \geq 0.$$

Note that since $\ell = 6k+5$, we have that $g_\ell(x) = k^{\mathcal{O}(x)}$, and therefore if $\mathcal{T}$ has width $\mathcal{O}(k)$, then

$$\Phi_{\ell,\mathcal{T}}(t) \leq k^{\mathcal{O}(k)} \cdot \mathsf{hgt}_T(t) \qquad \text{for every } t \in V(T).$$

Intuitively, the factor $g_\ell(|\mathsf{bag}(t)|)$ in the potential function allows us to update the tree decomposition by replacing the node t with $\mathcal{O}(\ell)$ copies of t, where each copy t' has the same height as t but the bag of t' is strictly smaller than that of t. This is analogous to replacing each bag with three smaller bags in Section 4.3. We remark that the constant 84 comes from $4 \cdot 21$, where the 21 will come from Lemma 6.51.

The purpose of the second factor $\mathsf{hgt}_T(t)$ is to penalize tree decompositions with too large height. In Section 6.6 it will be used for the height-reduction scheme. We remark that the only purpose of the $\mathsf{hgt}_T(t)$ factor is the height-reduction scheme, in particular, the running time of the refinement operation could also be analyzed without it.

For a subset $W \subseteq V(T)$, we denote

$$\Phi_{\ell,\mathcal{T}}(W) = \sum_{t \in W} \Phi_{\ell,\mathcal{T}}(t),$$

and similarly, for the whole tree decomposition $\mathcal{T} = (T, \mathsf{bag})$, we set

$$\Phi_\ell(\mathcal{T}) = \Phi_{\ell,\mathcal{T}}(V(T)) = \sum_{t \in V(T)} \Phi_{\ell,\mathcal{T}}(t).$$

6.5.2 Refinement of components

Throughout this and the following subsection, let us fix that $\mathcal{T} = (T, \mathsf{bag})$ is a binary tree decomposition of G of width at most $\ell + 1 = 6k + 6$, T_{pref} is a prefix of T (given as an input for the refinement operation) that contains all bags of size $\ell + 2$, and $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$ is a $d_{\mathcal{T}}$-minimal c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$ for the vertex-depth function $d_{\mathcal{T}}(v) = \mathsf{depth}_T(\mathsf{forget}_{\mathcal{T}}(v))$, and $c \leq \mathcal{O}(k^4)$.

In the refinement operation, we would like to construct the new binary tree decomposition $\mathcal{T}'$ by combining a tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$ with tree decompositions $\mathcal{T}^C$ corresponding to components $C \in \mathsf{cc}(G \setminus X)$, each of which is a tree decomposition of the graph $G[N[C]]$ and contains $N(C)$ in its root bag. While this is the high-level intuition, this process will actually be a bit more complicated. For example, we have no control on the number of connected components of $G \setminus X$ (e.g. if G is a star and X is its center), so we actually cannot iterate through all of the components. This subsection is dedicated to resolving technicalities like this.

Let us start with a structural lemma about blockages that we will need in this subsection.

Lemma 6.39. *If $t \in \mathsf{Blockages}(T_{\mathsf{pref}}, X)$, then $\mathsf{cmp}(t) \cap X = \emptyset$.*

Proof. Assume otherwise. Then, we claim that the set $X' = X \setminus \mathsf{cmp}(t)$ is also a c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$, contradicting the $d_{\mathcal{T}}$-minimality of X. In fact, we will show that $\mathsf{torso}_G(X')$ is a subgraph of $\mathsf{torso}_G(X)$.

First, if t is a clique blockage, then this clearly holds because $\mathsf{adh}(t)$ separates $X \cap \mathsf{cmp}(t)$ from X', and is a clique in both $\mathsf{torso}_G(X)$ and $\mathsf{torso}_G(X')$.

Then, assume t is a component blockage covered by a component $C \in \mathsf{cc}(G \setminus X)$, and suppose there is a component $C' \in \mathsf{cc}(G \setminus X')$ so that $N(C')$ contains a pair of distinct vertices $u, v \in X'$ so that $uv \notin E(\mathsf{torso}_G(X'))$. The component C' must intersect $\mathsf{cmp}(t)$, as otherwise it would also be a component of $G \setminus X$. If $C' \subseteq \mathsf{cmp}(t)$, then $N(C') \subseteq \mathsf{adh}(t)$, so the lemma holds because $\mathsf{adh}(t) \cap X = \mathsf{adh}(t) \cap X' \subseteq N(C)$ is a clique in $\mathsf{torso}_G(X)$. If C' intersects $\mathsf{adh}(t)$, then $C \subseteq C'$, but $C' \setminus C \subseteq \mathsf{cmp}(t)$, so $N(C') \setminus N(C) \subseteq \mathsf{adh}(t)$, implying that $N(C') = N(C)$, which is already a clique in $\mathsf{torso}_G(X)$. □

Classification of nodes and components

We now further classify the unexplored nodes of T into two categories. Note that for every unexplored node $t \in V(T)$, there exists a unique ancestor a of t in $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ (it might be that $a = t$). If a is a component blockage covered by a component C, then we say that t is a *covered node*, covered by the component C. If a is clique blockage, then we say that t is a *clique-blocked node*. This partitions $V(T)$ into the prefix of explored nodes $\mathsf{Expl}(T_{\mathsf{pref}}, X)$, and the subtrees rooted at appendices $a \in \mathsf{app}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$, each of which consists entirely of either (1) covered nodes covered by the same component $C \in \mathsf{cc}(G \setminus X)$, or (2) clique-blocked nodes.

We classify the components $C \in \mathsf{cc}(G \setminus X)$ into three categories. We say that C is a *proper component* if C intersects $\mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$. Observe that if C is not proper, then $C \subseteq \mathsf{cmp}(t)$ for some blockage t. If the blockage t is a component blockage covered by a component C', then we say that C is a *covered component* covered by C'. Note that in this case C' must be a proper component because it intersects $\mathsf{adh}(t)$ implying that it intersects $\mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$. Otherwise, if t is a clique blockage, then we say that C is a *clique-blocked* component.

Construction of the decompositions

Then we define the tree decomposition $\mathcal{T}^C = (T^C, \mathsf{bag}^C)$ for a proper component C. First, we let

$$T^C = T[\{t \in \mathsf{Expl}(T_{\mathsf{pref}}, X) \mid \mathsf{bag}(t) \cap C \neq \emptyset\} \cup \{t \in V(T) \mid t \text{ is covered by } C\}].$$

In other words, T^C is the subtree of T induced by the nodes that are either (1) explored and whose bags intersect C, or (2) are covered by C. Because $X \supseteq \mathsf{bags}(T_{\mathsf{pref}})$, $V(T^C)$ is disjoint from T_{pref}. Note that T^C is indeed connected, because if a blockage t is covered by C, then C intersects $\mathsf{adh}(t)$ by definition, so $\mathsf{parent}(t) \in V(T^C)$. The connectedness of T^C implies that there exists a unique minimum-depth node $r \in V(T)$ that is in $V(T^C)$. The tree T^C will be treated as rooted at r.

Then, similarly to Chapters 4 and 5 (in fact, exactly the same as in the proof of Lemma 5.11), we define for $t \in V(T)$ and a proper component C that

$$\mathsf{pull}(t, C) = \{v \in N(C) \mid \mathsf{forget}_{\mathcal{T}}(v) \text{ is a strict descendant of } t \text{ in } T\}.$$

Then, we define bag^C by setting

$$\mathsf{bag}^C(t) = \begin{cases} (\mathsf{bag}(t) \cap N[C]) \cup \mathsf{pull}(t, C) & \text{if } t \in \mathsf{Expl}(T_{\mathsf{pref}}, X) \\ \mathsf{bag}(t) & \text{otherwise.} \end{cases}$$

Note the similarity to the pruned improvement operation of Section 4.3. Let us then argue that $\mathcal{T}^C$ is indeed a tree decomposition.

Lemma 6.40. *Let C be a proper component and denote by $\tilde{C}$ the union of C and the components of $G \setminus X$ that are covered by C. Then, $\mathcal{T}^C$ is a tree decomposition of $N[\tilde{C}]$, and the root bag of $\mathcal{T}^C$ contains $N(C) = N(\tilde{C})$.*

Proof. First, if a component $C' \in \mathsf{cc}(G \setminus X)$ is a subset of $\mathsf{cmp}(t)$ for a blockage t covered by C, then by Lemma 6.39, $N(C') \subseteq \mathsf{adh}(t)$, which by $\mathsf{adh}(t) \subseteq N[C]$ implies that $N(C') \subseteq N(C)$. This implies that $N(C) = N(\tilde{C})$.

Then, let us check the conditions of tree decompositions. Note that $V(T^C)$ contains all nodes of T whose bags intersect $\tilde{C}$, which immediately implies the vertex condition for vertices in $\tilde{C}$. Also because of this, for each edge $uv \in E(G)$ with $u \in \tilde{C}$ and $v \in N[\tilde{C}]$, there must be $t \in V(T^C)$ so that $\{u, v\} \subseteq \mathsf{bag}(t)$, implying that $\{u, v\} \subseteq \mathsf{bag}^C(t)$. This proves the edge condition for all edges except those with both endpoints in $N(C)$, and the vertex condition for vertices in $N(C)$. To show that no bag of $\mathcal{T}^C$ contains vertices outside of $N[\tilde{C}]$, observe that this holds by definition for $t \in \mathsf{Expl}(T_{\mathsf{pref}}, X)$, and for other nodes by the definition of component blockage and Lemma 6.39. This finishes the proof of the vertex condition.

Let $r \in V(T^C)$ be the root of T^C and $v \in N(C)$. If all occurrence of v in $\mathcal{T}$ would be outside of the subtree rooted at r, then $\mathcal{T}$ would not satisfy the edge condition, so either $v \in \mathsf{bag}(r)$, or $v \in \mathsf{pull}(r, C)$. In both cases, $v \in \mathsf{bag}^C(r)$, so therefore $N(C) \subseteq \mathsf{bag}^C(r)$, finishing also the proof of the edge condition.

For the connectedness condition, for each $v \in \tilde{C}$ this holds because the set of nodes whose bag contains v in $\mathcal{T}$ and in $\mathcal{T}^C$ are the same. For vertices $v \in N(C)$, we observe that because $\mathsf{cmp}(t) \cap X = \emptyset$ for each blockage t (Lemma 6.39), $\mathsf{forget}_{\mathcal{T}}(v)$ must be either explored or not a descendant of r, and thus the additions of v to bags preserves that the subtree of v is connected. □

We then bound the sizes of bags of $\mathcal{T}^C$. This is a similar argument as we used in Sections 4.2 and 5.2, in particular, by using the $d_{\mathcal{T}}$-minimality of X.

Lemma 6.41. *For $t \in \mathsf{Expl}(T_{\mathsf{pref}}, X)$, it holds that $|\mathsf{bag}^C(t)| < |\mathsf{bag}(t)|$.*

Proof. By the definition of $\mathsf{bag}^C(t)$, it suffices to prove that $|\mathsf{pull}(t,C)| < |\mathsf{bag}(t) \setminus N[C]|$.

First, suppose that $\mathsf{pull}(t,C) = \emptyset$, in which case we need to prove that $\mathsf{bag}(t)$ is not a subset of $N[C]$. However, if $\mathsf{bag}(t)$ would be a subset of $N[C]$, then $\mathsf{adh}(t)$ would be a subset of $N[C]$, so some ancestor of t would be a blockage, so $t \notin \mathsf{Expl}(T_{\mathsf{pref}}, X)$.

It remains to prove that if $\mathsf{pull}(t,C)$ is non-empty, then $|\mathsf{pull}(t,C)| < |\mathsf{bag}(t) \setminus N[C]|$. For this proof, recall that $d_{\mathcal{T}}(v) = \mathsf{depth}_T(\mathsf{forget}_{\mathcal{T}}(v))$ for all $v \in V(G)$. For the sake of contradiction, assume that $\mathsf{pull}(t,C)$ is non-empty and $|\mathsf{pull}(t,C)| \geq |\mathsf{bag}(t) \setminus N[C]|$. We claim that now,

$$S = (N(C) \setminus \mathsf{pull}(t,C)) \cup (\mathsf{bag}(t) \setminus N[C])$$

is an $(N(C), \mathsf{bags}(T_{\mathsf{pref}}))$-separator that contradicts the fact that $N(C)$ is $d_{\mathcal{T}}$-linked into $\mathsf{bags}(T_{\mathsf{pref}})$, which by Lemma 6.29 contradicts that X is $d_{\mathcal{T}}$-minimal.

First, note that because $\mathsf{pull}(t,C) \subseteq N(C)$, we indeed have that $|S| \leq |N(C)|$. Moreover, because for each $v \in \mathsf{pull}(t,C)$ the highest bag containing v is a strict descendant of t, we have for all $v \in \mathsf{pull}(t,C)$ and $u \in \mathsf{bag}(t)$ that $d_{\mathcal{T}}(v) > d_{\mathcal{T}}(u)$, implying that $d_{\mathcal{T}}(S) < d_{\mathcal{T}}(N(C))$. It remains to prove that S indeed separates $N(C)$ from $\mathsf{bags}(T_{\mathsf{pref}})$. For this, it suffices to prove that it separates $\mathsf{pull}(t,C)$ from $\mathsf{bags}(T_{\mathsf{pref}})$, because $N(C) \setminus S = \mathsf{pull}(t,C)$.

Suppose P is a shortest path from $\mathsf{pull}(t,C)$ to $\mathsf{bags}(T_{\mathsf{pref}})$ in $G \setminus S$. Because C is disjoint from $\mathsf{bags}(T_{\mathsf{pref}})$ and $N(C) \setminus S = \mathsf{pull}(t,C)$, we have that P intersects $N[C]$ only on its first vertex. However, observe that because the nodes of $\mathcal{T}$ whose bags contain vertices from $\mathsf{pull}(t,C)$ are strict descendants of t, and t is a descendant of an appendix of T_{pref}, it holds that $\mathsf{bag}(t)$ separates $\mathsf{pull}(t,C)$ from $\mathsf{bags}(T_{\mathsf{pref}})$, and therefore P must intersect $\mathsf{bag}(t)$. However, as $\mathsf{bag}(t)$ is disjoint from $\mathsf{pull}(t,C)$, the intersection of P and $\mathsf{bag}(t)$ must be in $\mathsf{bag}(t) \setminus N[C]$, but $\mathsf{bag}(t) \setminus N[C] \subseteq S$, so no such path P can exist, implying that S is an $(N(C), \mathsf{bags}(T_{\mathsf{pref}}))$-separator. □

Because the other bags of $\mathcal{T}^C = (T^C, \mathsf{bag}^C)$ are directly copied from $\mathcal{T}$, it follows that $|\mathsf{bag}^C(t)| \leq |\mathsf{bag}(t)|$ for all $t \in V(T^C)$, so the width of $\mathcal{T}^C$ is at most ℓ.

In addition to constructing the decompositions $\mathcal{T}^C$ for proper components C, we will copy clique-blocked subtrees of $\mathcal{T}$ in the refinement. In particular, for each clique blockage t, the tree decomposition rooted at it is copied without any changes. For a clique blockage t we define $\mathcal{T}^t = \mathcal{T}\restriction_{\mathsf{desc}_T(t)} = (T[\mathsf{desc}_T(t)], \mathsf{bag}\restriction_{\mathsf{desc}_T(t)})$, and view $\mathcal{T}^t$ as rooted at t.

Lemma 6.42. *Let t be a clique blockage. Then, $\mathcal{T}^t$ is a tree decomposition of the graph $G[\mathsf{cmp}(t) \cup \mathsf{adh}(t)]$.*

Proof. Straightforward verification of properties of tree decompositions. (This holds in fact for any rooted subtree of a tree decomposition, so we do not use properties of blockages.) □

The refinement forest

We define the *refinement forest* of $\mathcal{T}$ with respect to T_{pref} and X to be the collection $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ of binary tree decompositions that contains

- for every proper component $C \in \mathsf{cc}(G \setminus X)$ the tree decomposition $\mathcal{T}^C$, and
- for every clique blockage t the tree decomposition $\mathcal{T}^t$.

Note that Lemma 6.41 and the construction of $\mathcal{T}^t$ imply that all tree decompositions in the refinement forest have width at most ℓ.

Lemma 6.43. *For every vertex $v \in V(G) \setminus X$, there is exactly one tree decomposition in $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ that contains v, and for every edge $uv \in E(G)$ with $v \in V(G) \setminus X$, there is a tree decomposition in the refinement forest that contains a bag containing $\{u, v\}$.*

Proof. Let $v \in C'$ for $C' \in \mathsf{cc}(G \setminus X)$. If C' is a covered by a proper component C, or C' is a proper component C, then by Lemma 6.40 $\mathcal{T}^C$ contains v, and for every $u \in N(v)$ also a bag containing $\{u, v\}$. If C' is a clique-blocked component, then there exists a clique blockage $t \in V(T)$ so that $C' \subseteq \mathsf{cmp}(t)$, implying that v is in the tree decomposition $\mathcal{T}^t$. Because $N(C') \subseteq \mathsf{adh}(t)$, by Lemma 6.42 it also holds that for every $u \in N(v)$, there is a bag of $\mathcal{T}^t$ that contains $\{u, v\}$. □

Let $\tilde{\mathcal{T}} = (\tilde{T}, \tilde{\mathsf{bag}}) \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$ be a tree decomposition in the refinement forest. The *interface* of $\tilde{\mathcal{T}}$ is the set $X \cap \mathsf{bags}(\tilde{\mathcal{T}})$, i.e., the vertices of X in $\tilde{\mathcal{T}}$, and is denoted by $\mathsf{Int}(\tilde{\mathcal{T}})$. If $\tilde{\mathcal{T}} = \mathcal{T}^C$ for a proper component C, then $\mathsf{Int}(\tilde{\mathcal{T}}) = N(C)$ by Lemma 6.40, and if $\tilde{\mathcal{T}} = \mathcal{T}^t$ for a clique blockage t, then $\mathsf{Int}(\tilde{\mathcal{T}}) = \mathsf{adh}(t)$ by Lemma 6.42. In both cases, the interface $\mathsf{Int}(\tilde{\mathcal{T}})$ is a clique in $\mathsf{torso}_G(X)$ and a subset of the root bag of $\tilde{\mathcal{T}}$.

Next we upper bound the total potential of the refinement forest. We include a term $|\mathsf{Expl}(T_{\mathsf{pref}}, X) \setminus T_{\mathsf{pref}}|$ to be able to charge the running time of the refinement operation from the potential. Furthermore, we also include an additional term $42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}})$ for every tree decomposition $\tilde{\mathcal{T}}$. This potential will be later used for constructing additional bags of size $|\mathsf{Int}(\tilde{\mathcal{T}})|$ to combine the trees of the forest.

Lemma 6.44. *It holds that*

$$|\mathsf{Expl}(T_{\mathsf{pref}}, X) \setminus T_{\mathsf{pref}}| + \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) + 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) \leq \Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a).$$

Proof. First, let $Y_1 \subseteq V(T)$ be the set of clique-blocked nodes of $\mathcal{T}$, and let $\mathsf{ReFo}_1(T_{\mathsf{pref}}, X) \subseteq \mathsf{ReFo}(T_{\mathsf{pref}}, X)$ be the set of tree decompositions in the refinement forest that correspond to clique blockages. We observe that

$$\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_1(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) = \Phi_{\ell,\mathcal{T}}(Y_1), \tag{6.45}$$

because the clique-blocked subtrees are directly copied from $\mathcal{T}$ to the refinement forest.

Then, let $Y_2 \subseteq V(T)$ be the set of covered nodes of $\mathcal{T}$, and let $\mathsf{ReFo}_2(T_{\mathsf{pref}}, X) = \mathsf{ReFo}(T_{\mathsf{pref}}, X) \setminus \mathsf{ReFo}_1(T_{\mathsf{pref}}, X)$ be the set of tree decompositions in the refinement forest that correspond to proper components. Let $\mathcal{T}^C = (T^C, \mathsf{bag}^C) \in \mathsf{ReFo}_2(T_{\mathsf{pref}}, X)$, and let $t \in V(T^C)$. If $t \in Y_2$, then $\mathcal{T}^C$ is the only decomposition in $\mathsf{ReFo}_2(T_{\mathsf{pref}}, X)$ that contains t, $\mathsf{hgt}_{T^C}(t) \leq \mathsf{hgt}_T(t)$, and $\mathsf{bag}^C(t) = \mathsf{bag}(t)$. Then, let $Y_3 = \mathsf{Expl}(T_{\mathsf{pref}}, X) \setminus T_{\mathsf{pref}}$, and observe that if $t \notin Y_2$, then $t \in Y_3$. In this case, t occurs in all decompositions $\mathcal{T}^C$ so that $\mathsf{bag}(t) \cap C \neq \emptyset$, implying that t occurs in at most $\ell + 1$ such decompositions. Furthermore, we observe that $\mathsf{hgt}_{T^C}(t) \leq \mathsf{hgt}_T(t)$, and by Lemma 6.41, it holds that $|\mathsf{bag}^C(t)| < |\mathsf{bag}(t)|$. Putting these observations together, we obtain that

$$\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_2(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) \leq \Phi_{\ell,\mathcal{T}}(Y_2) + \sum_{t \in Y_3} (\ell + 1) \cdot g_\ell(|\mathsf{bag}(t)| - 1) \cdot \mathsf{hgt}_T(t)$$

Because $g_\ell(x) = (84(\ell + 1))^{x+1}$, we get that

$$\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_2(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) \leq \Phi_{\ell,\mathcal{T}}(Y_2) + \Phi_{\ell,\mathcal{T}}(Y_3)/84. \tag{6.46}$$

Putting (6.45) and (6.46) together, we obtain

$$\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) \leq \Phi_{\ell,\mathcal{T}}(Y_1 \cup Y_2) + \Phi_{\ell,\mathcal{T}}(Y_3)/84.$$

Then we bound $\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}})$. We start by bounding it for decompositions in $\mathsf{ReFo}_1(T_{\mathsf{pref}}, X)$. Let $\tilde{\mathcal{T}} \in \mathsf{ReFo}_1(T_{\mathsf{pref}}, X)$, in particular, $\tilde{\mathcal{T}} = \mathcal{T}^t$ for

some clique blockage t. Note that in this case

$$g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) = g_\ell(|\mathsf{adh}(t)|) \cdot \mathsf{hgt}_T(t).$$

Suppose first that $t \in \mathsf{app}(T_{\mathsf{pref}})$. For those clique blockages t, the sum is bounded by $k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a)$. Then, suppose $t \notin \mathsf{app}(T_{\mathsf{pref}})$, in particular, $\mathsf{parent}_T(t) \in Y_3$. In this case, observe that if $\mathsf{bag}(\mathsf{parent}_T(t)) = \mathsf{adh}(t)$ would hold, then some strict ancestor of t would be a clique blockage, implying that t would not be a clique blockage. Therefore, $|\mathsf{bag}(\mathsf{parent}_T(t))| > |\mathsf{adh}(t)|$. As T is binary, each node in Y_3 can be a parent of at most two clique blockages, so we obtain

$$\begin{aligned}
&\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_1(T_{\mathsf{pref}}, X)} 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) \\
\leq & k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a) + \sum_{t \in Y_3} 42 \cdot 2 \cdot g_\ell(|\mathsf{bag}(t)| - 1) \cdot \mathsf{hgt}_T(t) \\
\leq & k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a) + \Phi_{\ell,\mathcal{T}}(Y_3)/6.
\end{aligned}$$

(In the last step, we can divide by 6 because $\ell + 1 \geq 6$.)

Finally, we bound $\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_2(T_{\mathsf{pref}}, X)} 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}})$. Let $\tilde{\mathcal{T}} = \mathcal{T}^C = (T^C, \mathsf{bag}^C)$ for some proper component C, and let $r^C \in Y_3$ be the root of T^C. Now,

$$\begin{aligned}
g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) &\leq g_\ell(|N(C)|) \cdot \mathsf{hgt}_T(r^C) \\
&\leq g_\ell(|\mathsf{bag}(r^C)| - 1) \cdot \mathsf{hgt}_T(r^C),
\end{aligned}$$

where the last inequality follows from the facts that $N(C) \subseteq \mathsf{bag}^C(r^C)$ and by Lemma 6.41 $|\mathsf{bag}^C(r^C)| < |\mathsf{bag}(r^C)|$. Because each node in Y_3 can be the root r^C for at most $\ell + 1$ proper components C, we get that

$$\begin{aligned}
&\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_2(T_{\mathsf{pref}}, X)} 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) \\
\leq & \sum_{t \in Y_3} (\ell + 1) \cdot 42 \cdot g_\ell(|\mathsf{bag}(t)| - 1) \cdot \mathsf{hgt}_T(t) \\
\leq & \Phi_{\ell,\mathcal{T}}(Y_3)/2.
\end{aligned}$$

Putting everything together, we obtain

$$\begin{aligned}
&\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) + 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) \\
\leq &\Phi_{\ell,\mathcal{T}}(Y_1 \cup Y_2) + \Phi_{\ell,\mathcal{T}}(Y_3)/84 + \Phi_{\ell,\mathcal{T}}(Y_3)/6 + \Phi_{\ell,\mathcal{T}}(Y_3)/2 + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a) \\
\leq &\Phi_{\ell,\mathcal{T}}(Y_1 \cup Y_2) + \frac{5}{6} \cdot \Phi_{\ell,\mathcal{T}}(Y_3) + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a).
\end{aligned}$$

Because $g_\ell(x) \geq 6$ for all $x \geq 0$, we have that $\frac{5}{6} \cdot \Phi_{\ell,\mathcal{T}}(Y_3) \leq \Phi_{\ell,\mathcal{T}}(Y_3) - |Y_3|$, and therefore

$$\begin{aligned}
|Y_3| + &\sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) + 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) \\
&\leq \Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a),
\end{aligned}$$

which yields the conclusion. □

For later analysis of potential, we will also need the concept of the *source appendix* of a tree decomposition $\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$ in the refinement forest. If $\tilde{\mathcal{T}} = \mathcal{T}^C$ for some proper component C, then $C \subseteq \mathsf{cmp}(a)$ for some appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$ of T_{pref}. In that case, we define the source appendix of $\tilde{\mathcal{T}}$ to be $\mathsf{Source}(\tilde{\mathcal{T}}) = a$. If $\tilde{\mathcal{T}} = \mathcal{T}^t$ for some clique blockage t, then there is a unique appendix $a \in \mathsf{app}(T_{\mathsf{pref}})$ of T_{pref} so that $t \in \mathsf{desc}_T(a)$. In that case, we define the source appendix of $\tilde{\mathcal{T}}$ to be $\mathsf{Source}(\tilde{\mathcal{T}}) = a$. The properties we aim to capture with this definition are stated in the next two lemmas.

First, we observe that the height of $\tilde{\mathcal{T}}$ is at most the height of its source appendix.

Lemma 6.47. *Let $\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$. Then, $\mathsf{hgt}(\tilde{\mathcal{T}}) \leq \mathsf{hgt}_T(\mathsf{Source}(\tilde{\mathcal{T}}))$.*

Proof. Immediate from the construction of $\tilde{\mathcal{T}}$ and the definition of source appendix. □

Then, we observe that the number of distinct interfaces of tree decompositions with a fixed source appendix is bounded. Here we use that X is c-small, and the assumptions that $c \leq \mathcal{O}(k^4)$ and $\ell = \mathcal{O}(k)$.

Lemma 6.48. *For every $a \in \mathsf{app}(T_{\mathsf{pref}})$ it holds that*

$$|\{\mathsf{Int}(\tilde{\mathcal{T}}) \mid \tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X) \text{ and } \mathsf{Source}(\tilde{\mathcal{T}}) = a\}| \leq k^{\mathcal{O}(k)}.$$

Proof. We observe that if $\mathsf{Source}(\tilde{\mathcal{T}}) = a$, then $\mathsf{Int}(\tilde{\mathcal{T}}) \subseteq \mathsf{cmp}(a) \cup \mathsf{adh}(a)$. However, it also holds that $\mathsf{Int}(\tilde{\mathcal{T}}) \subseteq X$, so in fact $\mathsf{Int}(\tilde{\mathcal{T}}) \subseteq X \cap (\mathsf{cmp}(a) \cup \mathsf{adh}(a))$. By the definition of c-

smallness and the fact that $|\mathsf{adh}(a)| \leq \ell+1$, we have that $|X \cap (\mathsf{cmp}(a) \cup \mathsf{adh}(a))| \leq c+\ell+1$. Because $|\mathsf{Int}(\tilde{\mathcal{T}})| \leq 2k+2$ (as X is a k-closure and $\mathsf{Int}(\tilde{\mathcal{T}})$ is a clique in $\mathsf{torso}_G(X)$), it holds that the number of different interfaces is at most $(c+\ell+2)^{2k+2}$, which is bounded by $k^{\mathcal{O}(k)}$. □

As the last result of this subsection, we show that a representation of the refinement forest can be computed efficiently when given the output of Lemma 6.35. For this let us first define the *representation* of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$. The representation is a list of length $|\mathsf{ReFo}(T_{\mathsf{pref}}, X)|$ that for each $\tilde{\mathcal{T}} = (\tilde{T}, \tilde{\mathsf{bag}}) \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$ contains a tuple consisting of $\mathsf{Int}(\tilde{\mathcal{T}})$, $\mathsf{hgt}(\tilde{\mathcal{T}})$, and in addition,

- if $\tilde{\mathcal{T}} = \mathcal{T}^C$ for a proper component C, the restriction $\mathcal{T}^C \restriction_{\mathsf{Expl}(T_{\mathsf{pref}}, X)}$ and a mapping from blockages that are covered by C to the nodes of $\mathcal{T}^C \restriction_{\mathsf{Expl}(T_{\mathsf{pref}}, X)}$ below which they should be attached to construct $\mathcal{T}^C$, and
- if $\tilde{\mathcal{T}} = \mathcal{T}^t$ for a clique blockage t, the node t.

We then give the algorithm to compute the representation.

Lemma 6.49. *The prefix-rebuilding data structure of Lemma 6.35 can be extended so that the* $\mathsf{Closure}(T_{\mathsf{pref}})$ *operation also returns the representation of* $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$.

Proof. In order to compute the heights $\mathsf{hgt}(\tilde{\mathcal{T}})$ for the representation, we maintain also the prefix-rebuilding data structure of Lemma 6.7 to be able to query the heights of nodes of $\mathcal{T}$. Now, given the output of the $\mathsf{Closure}(T_{\mathsf{pref}})$ operation of Lemma 6.35, our goal is to compute the representation of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ in time $k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$. We remind the reader that the output of the $\mathsf{Closure}(T_{\mathsf{pref}})$ operation contains a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$, the graph $\mathsf{torso}_G(X)$, the sets $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ and $\mathsf{Expl}(T_{\mathsf{pref}}, X)$, and for each blockage whether it is a clique blockage or component blockage.

First, given this information, computing the tuples for the decompositions in $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ corresponding to clique blockages is straightforward to do in $k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$ time.

Then we compute for each proper component C the sets $C \cap \mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$ and $N(C)$. Let G^E be the graph on the vertex set $V(G^E) = \mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X)) \setminus X$ that contains the edges of $G[V(G^E)]$, and for every component blockage t a clique on $\mathsf{adh}(t) \setminus X$.

Claim 6.50. *For $u, v \in V(G^E)$, there exists an u-v-path in G^E if and only if there exists an u-v-path in $G \setminus X$.*

Proof of the claim. First, consider an u-v-path in G^E. The edges of it that are not edges of G are created by making adhesions of component blockages cliques. If $xy \in E(G^E)$ is an edge created like that, then by the definition of component blockage x and y are in the same connected component of $G \setminus X$.

Then, consider an u-v-path in $G \setminus X$. Let $x, z_1, \ldots, z_p, y$ be a subpath of it so that $x, y \in V(G^E)$ but $z_1, \ldots, z_p \notin V(G^E)$. We claim that then $xy \in E(G^E)$. Because $\{z_1, \ldots, z_p\}$ is a connected set in G disjoint from $\mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$, it holds that $\{z_1, \ldots, z_p\} \subseteq \mathsf{cmp}(t)$ for some blockage t. Now because xz_1 and z_py are edges of G and $x, y \in \mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$, it must be that $x, y \in \mathsf{adh}(t)$. This implies that the path can be translated into an u-v-path in G^E. $\lhd$

Now the sets $C \cap \mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$ for proper components C can be computed in $k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$ time by first computing G^E explicitly and then its connected components. Note that $G[V(G^E)]$ can be computed explicitly because all of its edges are stored in $\mathsf{edges}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$, where $(T, \mathsf{bag}, \mathsf{edges})$ is the stored annotated tree decomposition.

To compute the sets $N(C)$, first recall that by Lemma 6.39, $X \subseteq \mathsf{bags}(\mathsf{Expl}(T_{\mathsf{pref}}, X))$. The vertices $v \in N(C)$ that are in $\mathsf{adh}(t)$ for some component blockage t covered by C can be found by iterating through blockages. If $v \in N(C)$ is not in the adhesion of any component blockage t covered by C, then there exists $u \in C$ so that $uv \in E(G)$ and $\mathsf{forget}_{\mathcal{T}}(uv) \in \mathsf{Expl}(T_{\mathsf{pref}}, X)$, so such vertices v can be found by iterating through $\mathsf{Expl}(T_{\mathsf{pref}}, X)$ and inspecting the edges map. Therefore, computing the sets $N(C)$ for all proper components C can be done in $k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$ total time.

Then for each component C we compute the highest node in $\mathsf{Expl}(T_{\mathsf{pref}}, X)$ whose bag intersects C, and then by using depth-first search from it, compute the restriction $\mathcal{T}^C \restriction_{\mathsf{Expl}(T_{\mathsf{pref}}, X)}$ and the mapping from blockages that are covered by C to the nodes of $\mathcal{T}^C \restriction_{\mathsf{Expl}(T_{\mathsf{pref}}, X)}$ below which they should be attached to construct $\mathcal{T}^C$. By using the height data structure, we can also simultaneously compute $\mathsf{hgt}(\mathcal{T}^C)$. For each C, this takes $k^{\mathcal{O}(k)} \cdot |\mathcal{T}^C \restriction_{\mathsf{Expl}(T_{\mathsf{pref}}, X)}|$ time, which sums up to $k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$ total time. $\square$

6.5.3 Combining the components

Suppose we have constructed the refinement forest $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$, and we have a tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$. Because the interface of each tree decomposition in $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ is a clique in $\mathsf{torso}_G(X)$, it is clear that they can be combined to create a tree decomposition of G, in the fashion of the construction in Section 5.2. However,

the naive way of combining these tree decompositions could make the resulting tree decomposition not binary.

A natural idea is to then append auxiliary binary trees to $\mathcal{T}^X$, so that every tree decomposition in the refinement forest will have its own attachment point. However, this increases the potential of the resulting tree decomposition, so we have to be careful with the details of how this is implemented to control this increase. In this subsection we discuss the first part of constructing these binary trees, which is that for every interface $B = \mathsf{Int}(\tilde{\mathcal{T}})$ of a tree decomposition $\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$, we construct a tree decomposition $\mathcal{T}^B$ gathering all of the tree decompositions in $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ with the interface B to a single tree decomposition.

The following lemma, inspired by Huffman coding, will be crucial for bounding the potential of such $\mathcal{T}^B$.

Lemma 6.51. *Let $h_1, \ldots, h_m$ be positive integers and let $Q = h_1 + \ldots + h_m$. Then, there exists a binary tree T of height $\mathcal{O}(\log Q)$ and leaves labeled $h_1, \ldots, h_m$ in some order such that $\sum_{t \in V(T)} \mathsf{lheight}(t) \leq 21Q$, where*

$$\mathsf{lheight}(t) = \begin{cases} \mathsf{label}(t) & \text{if } t \text{ is a leaf,} \\ 1 + \max\{\mathsf{lheight}(c) \mid c \text{ is a child of } t\} & \text{otherwise.} \end{cases}$$

Moreover, given $h_1, \ldots, h_m$, such a tree can be computed in time $\mathcal{O}(m + \log Q)$.

Proof. Let $\lambda \colon \mathbb{Z}_{\geq 1} \to \mathbb{Z}_{\geq 0}$ be the function given by the formula $\lambda(a) = \lceil \log a \rceil$ (recall that log denotes the base-2 logarithm). In other words, $\lambda(a)$ is the smallest non-negative integer such that $2^{\lambda(a)} \geq a$. This implies that $2^{\lambda(a)} < 2a$.

We partition $h_1, \ldots, h_m$ into $\lambda(Q)$ groups $D_0, \ldots, D_{\lambda(Q)}$ as follows: We put h_i in D_j if $\frac{Q}{2^{j+1}} < h_i \leq \frac{Q}{2^j}$. This is a well-defined partition as these intervals for $j = 0, \ldots, \lambda(Q)$ are disjoint and their union covers the interval $[1, Q]$.

Now, for each non-empty group D_i let us create a binary tree T_i as follows. Let the elements of D_i be $d_1, \ldots, d_a$. Then, let T_i be an arbitrary binary tree with a leaves labeled $d_1, \ldots, d_a$, where all leaves are at distance at most $\lambda(a)$ from the root. As $2^{\lambda(a)} \geq a$, such a tree exists. If D_i is empty, we assume that T_i is empty as well.

As the next step create a path $P = t_0, \ldots, t_{\lambda(Q)}$ rooted at t_0. For each $i = 0, \ldots, \lambda(Q)$, if D_i is non-empty, assign the root of T_i as a child of t_i. Remove the suffix of P that has no subtrees attached to it. This completes the description of desired T. One can readily see that the construction can be computed in time $\mathcal{O}(m + \log Q)$.

What remains is to prove the required properties of T. As each T_i has height $\mathcal{O}(\log Q)$ and the path P has length $\mathcal{O}(\log Q)$, it is clear that the height of T is $\mathcal{O}(\log Q)$ as well.

Next, we will prove a bound on the sum of $\mathsf{lheight}(t)$ for a particular T_i. Let the elements of D_i be $d_1, \ldots, d_a \in (\frac{Q}{2^{i+1}}, \frac{Q}{2^i}]$. Let us group the nodes $t \in V(T_i)$ by their distance j to the farthest leaf in the subtree of T_i rooted at t. For $j = 0$, the group comprises the leaves of T_i. The leaves are labeled by D_i, so their values of $\mathsf{lheight}$ do not exceed $\frac{Q}{2^i}$. Next, the nodes at distance $j \geq 1$ from the farthest leaf have the value of $\mathsf{lheight}$ at most $\frac{Q}{2^i} + j$ and there are at most $2^{\lambda(a)-j}$ of them (which follows from the fact that each such node is at depth at most $\lambda(a) - j$ in T_i). Hence, we have the bound

$$\begin{aligned}
\sum_{t \in V(T_i)} \mathsf{lheight}(t) &\leq \sum_{j=0}^{\lambda(a)} 2^{\lambda(a)-j} \left(\frac{Q}{2^i} + j \right) \\
&\leq 2^{\lambda(a)} \left(\frac{Q}{2^i} \sum_{j=0}^{\lambda(a)} 2^{-j} + \sum_{j=0}^{\lambda(a)} j \cdot 2^{-j} \right) \\
&\leq 2^{\lambda(a)} \left(\frac{Q}{2^i} \sum_{j=0}^{\infty} 2^{-j} + \sum_{j=0}^{\infty} j \cdot 2^{-j} \right) \\
&\leq 2^{\lambda(a)} \left(2 \cdot \frac{Q}{2^i} + 2 \right).
\end{aligned}$$

As additionally $2^{\lambda(a)} \cdot 2 \leq 4a$ and $2^{\lambda(a)} \cdot 2 \cdot \frac{Q}{2^i} \leq 8a \cdot \frac{Q}{2^{i+1}} < 8(d_1 + \ldots + d_a)$, we find that

$$\sum_{t \in V(T_i)} \mathsf{lheight}(t) \leq 8(d_1 + \ldots + d_a) + 4a \leq 12(d_1 + \ldots + d_a). \tag{6.52}$$

Summing (6.52) over all $i = 0, 1, \ldots, \lambda(Q)$, we get that

$$\sum_{i=0}^{\lambda(Q)} \sum_{t \in V(T_i)} \mathsf{lheight}(t) \leq 12Q.$$

What remains is to bound the sum of $\mathsf{lheight}$ for the nodes of P. Define

$$x_i = \frac{Q}{2^i} + 2\lambda(Q) - i + 2.$$

We now prove $\mathsf{lheight}(t_i) \leq x_i$ by induction on i. Each node $t_i \in V(P)$ has at most two children, the root r_i of T_i if D_i is non-empty, and the node t_{i+1} on the path P if t_{i+1} exists. First, by the induction assumption, $\mathsf{lheight}(t_{i+1}) \leq x_{i+1} \leq x_i - 1$.

Then we bound $\mathsf{lheight}(r_i)$. Because each element of D_i is larger than $\frac{Q}{2^{i+1}}$, we have that $|D_i| \leq 2^{i+1}$, implying that the height of T_i is at most $i+1$. Therefore, as each element of D_i is at most $\frac{Q}{2^i}$, we get $\mathsf{lheight}(r_i) \leq \frac{Q}{2^i} + i + 1 \leq x_i - 1$. Thus indeed $\mathsf{lheight}(t_i) \leq x_i$.

Therefore,

$$\begin{aligned}\sum_{t \in P} \mathsf{lheight}(t) \leq & \sum_{i=0}^{\lambda(Q)} x_i \\ \leq & (\lambda(Q)+1)(2\lambda(Q)+2) + \sum_{i=0}^{\lambda(Q)} \frac{Q}{2^i} \\ \leq & (\lambda(Q)+1)(2\lambda(Q)+2) + 2Q.\end{aligned}$$

Now, since $\lambda(Q) < \log Q + 1$, we find that $(\lambda(Q)+1)(2\lambda(Q)+2) < (\log Q + 2)(2\log Q + 3)$. It can be shown by concavity and case-analysis for small values that for all $Q \in \mathbb{Z}_{\geq 1}$

$$(\log Q + 2)(2\log Q + 3) \leq 7Q.$$

Hence,

$$\sum_{t \in P} \mathsf{lheight}(t) \leq 9Q.$$

We conclude that

$$\sum_{t \in V(T)} \mathsf{lheight}(t) = \sum_{i=0}^{\lambda(Q)} \sum_{t \in V(T_i)} \mathsf{lheight}(t) + \sum_{t \in P} \mathsf{lheight}(t) \leq 12Q + 9Q = 21Q. \qquad \square$$

Having proved Lemma 6.51, we can define the tree decompositions that will be attached to $\mathcal{T}^X$. Let $B \subseteq V(G)$ be a set of vertices. We denote by $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$ the subset $\{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X) \mid \mathsf{Int}(\tilde{\mathcal{T}}) = B\}$ of the refinement forest consisting of the decompositions with interface B. Then, for each B so that $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$ is non-empty, we will define a binary tree decomposition $\mathcal{T}^B$ that contains all of the tree decompositions in $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$ and has root bag equal to B.

Let us enumerate $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X) = (T_1, \mathsf{bag}_1), \ldots, (T_m, \mathsf{bag}_m)$, and let $h_i = \mathsf{hgt}(T_i) + 1$ for $i \in [m]$. Let T^B_{pref} be the binary tree returned by the algorithm of Lemma 6.51 when given the integers $h_1, \ldots, h_m$. We construct a binary tree T^B by attaching each T_i as the child of the leaf of T^B_{pref} that is labeled with h_i. The binary tree decomposition $\mathcal{T}^B = (T^B, \mathsf{bag}^B)$ is then obtained by setting $\mathsf{bag}^B{\restriction_{V(T_i)}} = \mathsf{bag}_i$ for every $i \in [m]$, and $\mathsf{bag}^B(t) = B$ for every $t \in T^B_{\mathsf{pref}}$.

The point of this construction is that now, for every $t \in T^B_{\mathsf{pref}}$ it holds that $\mathsf{hgt}_{T^B}(t) = \mathsf{lheight}(t)$, where $\mathsf{lheight}(t)$ is from the statement of Lemma 6.51. This means in particular that $\sum_{t \in T^B_{\mathsf{pref}}} \mathsf{hgt}_{T^B}(t) \leq 21 \cdot \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} (\mathsf{hgt}(\tilde{\mathcal{T}}) + 1)$.

We then observe some basic properties of $\mathcal{T}^B$.

Lemma 6.53. *It holds that*

1. *$\mathcal{T}^B$ is a tree decomposition of $G[\bigcup_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{bags}(\tilde{\mathcal{T}})]$,*
2. *$\mathcal{T}^B$ has width at most ℓ,*
3. *the root bag of $\mathcal{T}^B$ is equal to B, and*
4. *$\mathsf{hgt}(\mathcal{T}^B) \leq \mathcal{O}(\log |\mathcal{T}| + \log k) + \max_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}_T(\mathsf{Source}(\tilde{\mathcal{T}}))$.*

Proof. For Item 1, the vertex and edge conditions follow directly from vertex and edge conditions of the decompositions in $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$, and the fact that B is a $(\mathsf{bags}(\tilde{\mathcal{T}}_1), \mathsf{bags}(\tilde{\mathcal{T}}_2)$-separator for any distinct $\tilde{\mathcal{T}}_1, \tilde{\mathcal{T}}_2 \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$. The connectedness condition follows from the fact that B is a subset of the root bag of every decomposition in $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$. Item 2 follows from the fact that every tree decomposition in $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ has width at most ℓ, and Item 3 is direct from the definition of $\mathcal{T}^B$.

Because $|\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)| \leq \mathcal{O}(k \cdot |\mathcal{T}|)$ and $\mathsf{hgt}(\tilde{\mathcal{T}}) \leq |\mathcal{T}|$ for all $\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$, we have that $\sum h_i \leq \mathcal{O}(k \cdot |\mathcal{T}|^2)$ for the integers $h_1, \ldots, h_m$ given to the algorithm of Lemma 6.51. Therefore, $\mathsf{hgt}(T^B_{\mathsf{pref}}) \leq \mathcal{O}(\log |\mathcal{T}| + \log k)$, implying that $\mathsf{hgt}(\mathcal{T}^B) \leq \mathcal{O}(\log |\mathcal{T}| + \log k) + \max_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}(\tilde{\mathcal{T}})$, which by Lemma 6.47 implies Item 4. □

Then, we define the *grouped refinement forest* of $\mathcal{T}$ with respect to T_{pref} and X to be $\mathsf{GReFo}(T_{\mathsf{pref}}, X) = \{\mathcal{T}^B \mid \mathsf{ReFo}_B(T_{\mathsf{pref}}, X) \neq \emptyset\}$, i.e., the set of binary tree decompositions containing $\mathcal{T}^B$ for every set B that is the interface of some tree decomposition in the refinement forest. We analyze the total potential of the trees in $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$.

Lemma 6.54. *It holds that*

$$|\mathsf{Expl}(T_{\mathsf{pref}}, X) \setminus T_{\mathsf{pref}}| + \sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\mathcal{T}^B)$$
$$\leq \Phi_{\ell, \mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a).$$

Proof. By the construction of $\mathcal{T}^B$, it holds that

$$\Phi_\ell(\mathcal{T}^B) = \Phi_{\ell, \mathcal{T}^B}(T^B_{\mathsf{pref}}) + \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}),$$

implying that

$$\sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\mathcal{T}^B) = \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) + \sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \Phi_{\ell, \mathcal{T}^B}(T^B_{\mathsf{pref}}) \tag{6.55}$$

The plan to first bound the latter sum and then apply Lemma 6.44. By the construction of $\mathcal{T}^B$, it holds that

$$\Phi_{\ell, \mathcal{T}^B}(T^B_{\mathsf{pref}}) = \sum_{t \in T^B_{\mathsf{pref}}} g_\ell(|B|) \cdot \mathsf{hgt}_{T^B}(t) = g_\ell(|B|) \cdot \sum_{t \in T^B_{\mathsf{pref}}} \mathsf{hgt}_{T^B}(t).$$

Then, the construction of T^B_{pref} by Lemma 6.51 guarantees that

$$\begin{aligned} \sum_{t \in T^B_{\mathsf{pref}}} \mathsf{hgt}_{T^B}(t) &\leq 21 \cdot \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}(\tilde{\mathcal{T}}) + 1 \\ &\leq 42 \cdot \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}(\tilde{\mathcal{T}}), \end{aligned}$$

which implies

$$\Phi_{\ell, \mathcal{T}^B}(T^B_{\mathsf{pref}}) \leq 42 \cdot g_\ell(|B|) \cdot \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}(\tilde{\mathcal{T}}),$$

which in turn implies

$$\sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \Phi_{\ell, \mathcal{T}^B}(T^B_{\mathsf{pref}}) \leq \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}). \tag{6.56}$$

Now, Lemma 6.44 states that

$$\begin{aligned} |\mathsf{Expl}(T_{\mathsf{pref}}, X) \setminus T_{\mathsf{pref}}| + \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)} \Phi_\ell(\tilde{\mathcal{T}}) + 42 \cdot g_\ell(|\mathsf{Int}(\tilde{\mathcal{T}})|) \cdot \mathsf{hgt}(\tilde{\mathcal{T}}) \\ \leq \Phi_{\ell, \mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a), \end{aligned}$$

which yields the conclusion by combining with (6.55) and (6.56) □

Then we analyze the sum of the heights of the grouped refinement forest, for the purpose of later analysis of the potential when the tree decompositions $\mathcal{T}^B$ are attached to a tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$. For a rather technical reason, in this analysis we subtract $\mathcal{O}(\log |\mathcal{T}|)$ from the height of each tree in $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$.

Lemma 6.57. *There exists a constant α, so that*

$$\sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \big(\mathsf{hgt}(\mathcal{T}^B) - \alpha \cdot \log|\mathcal{T}|\big) \leq k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a).$$

Proof. If $|\mathcal{T}| < k$, then the lemma clearly holds, so assume that $k \leq |\mathcal{T}|$. In that case, by Lemma 6.53, the height of $\mathcal{T}^B$ is at most

$$\mathcal{O}(\log|\mathcal{T}|) + \max_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}_T(\mathsf{Source}(\tilde{\mathcal{T}})),$$

so by letting α depend on the constant hidden by the $\mathcal{O}$-notation in the $\mathcal{O}(\log|\mathcal{T}|)$ term, our goal will be to bound the sum

$$\sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \max_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} \mathsf{hgt}_T(\mathsf{Source}(\tilde{\mathcal{T}})).$$

For $\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)$, let us define the source appendix $\mathsf{Source}(\mathcal{T}^B) \in \mathsf{app}(T_{\mathsf{pref}})$ of $\mathcal{T}^B$ to be the source appendix $\mathsf{Source}(\tilde{\mathcal{T}})$ for $\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$ that maximizes $\mathsf{hgt}_T(\mathsf{Source}(\tilde{\mathcal{T}}))$. Now, our goal is to bound

$$\sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \mathsf{hgt}_T(\mathsf{Source}(\mathcal{T}^B)).$$

By Lemma 6.48, for every $a \in \mathsf{app}(T_{\mathsf{pref}})$ it holds that

$$|\{\mathsf{Int}(\tilde{\mathcal{T}}) \mid \tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X) \text{ and } \mathsf{Source}(\tilde{\mathcal{T}}) = a\}| \leq k^{\mathcal{O}(k)},$$

which implies $|\{\mathcal{T}^B \mid \mathsf{Source}(\mathcal{T}^B) = a\}| \leq k^{\mathcal{O}(k)}$, and therefore

$$\sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \mathsf{hgt}_T(\mathsf{Source}(\mathcal{T}^B)) \leq \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{hgt}_T(a),$$

which concludes the proof. □

We then define the representation of the grouped refinement forest $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$ as a list that contains for every interface $B = \mathsf{Int}(\tilde{\mathcal{T}})$ of a tree decomposition $\tilde{\mathcal{T}} \in \mathsf{ReFo}(T_{\mathsf{pref}}, X)$ the restriction $\mathcal{T}^B{\restriction}_{V(T^B_{\mathsf{pref}})}$ and the mapping from the leaves of $\mathcal{T}^B{\restriction}_{V(T^B_{\mathsf{pref}})}$ to decompositions in $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ indicating how they should be attached to $\mathcal{T}^B{\restriction}_{V(T^B_{\mathsf{pref}})}$ to construct $\mathcal{T}^B$.

Lemma 6.58. *Given the representation of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$, the representation of $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$ can be computed in $k^{\mathcal{O}(1)} \cdot |\mathsf{ReFo}(T_{\mathsf{pref}}, X)| + k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log|\mathcal{T}|$ time.*

Proof. First, we compute the set of distinct interfaces in time $k^{\mathcal{O}(1)} \cdot |\mathsf{ReFo}(T_{\mathsf{pref}}, X)|$ by bucket sorting. Then, for each interface B, we know the heights of the decompositions in $\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)$ from the representation of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$, and apply Lemma 6.51 with them. With the output of Lemma 6.51, it is straightforward to construct $\mathcal{T}^B\restriction_{V(T^B_{\mathsf{pref}})}$ in $k^{\mathcal{O}(1)} \cdot |V(T^B_{\mathsf{pref}})| = k^{\mathcal{O}(1)} \cdot (|\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)| + \log \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} (\mathsf{hgt}(\tilde{\mathcal{T}}) + 1))$ time. This takes in total

$$\sum_B \mathcal{O}\left(k^{\mathcal{O}(1)} \cdot |\mathsf{ReFo}_B(T_{\mathsf{pref}}, X)| + \log \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} (\mathsf{hgt}(\tilde{\mathcal{T}}) + 1)\right)$$
$$\leq k^{\mathcal{O}(1)} \cdot |\mathsf{ReFo}(T_{\mathsf{pref}}, X)| + \mathcal{O}\left(\sum_B \log \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} (\mathsf{hgt}(\tilde{\mathcal{T}}) + 1)\right)$$

time, which by the facts that by Lemma 6.48 the number of distinct interfaces is $k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}|$ and the fact that $\log \sum_{\tilde{\mathcal{T}} \in \mathsf{ReFo}_B(T_{\mathsf{pref}}, X)} (\mathsf{hgt}(\tilde{\mathcal{T}}) + 1) \leq \mathcal{O}(\log |\mathcal{T}| + \log k)$ implies the conclusion. □

6.5.4 Refinement operation

We are now almost ready to fully describe the refinement operation. Before going into it, let us state the Bodlaender-Hagerup Lemma for making tree decompositions into logarithmic depth.

Lemma 6.59 ([Bodlaender and Hagerup, 1998, Lemma 2.2])**.** *There is an algorithm that given a graph G and a tree decomposition (T, bag) of G of width ℓ, in time $\mathcal{O}(\ell \cdot |V(T)|)$ returns a binary tree decomposition (T', bag') of G of height $\mathcal{O}(\log |V(T)|)$, width at most $3\ell + 2$, and with $|V(T')| = \mathcal{O}(|V(T)|)$.*

Then we give the refinement operation. It is given as a form of a prefix-rebuilding data structure, in order to give a clean interface upon which we can build on in the later sections. We remark that this prefix-rebuilding data structure assumes the promise that the treewidth of G is at most k, without checking if this promise holds. Later, additional wrappers will be placed on top of it to guarantee that this promise always holds.

Lemma 6.60. *Let k be an integer and $\ell = 6k+5$. There exists an $(\ell+1)$-prefix-rebuilding data structure with overhead $2^{\mathcal{O}(k^8)}$, that maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a graph G of treewidth at most k, and additionally supports the operation*

- $\mathsf{Refine}(T_{\mathsf{pref}})$*: Given a prefix* T_{pref} *of* T *so that* T_{pref} *contains all nodes of* $\mathcal{T}$ *with bags of size* $\ell+2$*, returns a description* $\overline{u}$ *of a prefix-rebuilding update, so that the tree decomposition* $\mathcal{T}'$ *obtained by applying* $\overline{u}$ *has the following properties.*

- $\mathcal{T}'$ *has width at most* ℓ*.*
- $\mathcal{T}'$ *has potential at most*

$$\Phi_\ell(\mathcal{T}') \leq \Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot \log|\mathcal{T}| \cdot \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a) \right).$$

Moreover, it holds that

- *the running time of* $\mathsf{Refine}(T_{\mathsf{pref}})$*, and therefore also* $|\overline{u}|$*, is upper-bounded by*

$$2^{\mathcal{O}(k^9)} \cdot \left(\log|\mathcal{T}| \cdot \left(|T_{\mathsf{pref}}| + \sum_{t \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(t) \right) + \Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') \right).$$

The rest of this section is dedicated to the proof of Lemma 6.60.

In our data structure we store

- an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width $\leq \ell+1$,
- an instance $\mathbb{D}^{\mathsf{aux}}$ of the $(\ell+1)$-prefix-rebuilding data structure from Lemma 6.7 for maintaining various auxiliary information about $\mathcal{T}$,
- an instance $\mathbb{D}^{\mathsf{closure}}$ of the $(\ell+1)$-prefix-rebuilding data structure from Lemma 6.35 to compute closures and related objects, initialized with the parameters k, $\ell+1$, and $c = \mathcal{O}(\ell^4)$ for the integer c given by Lemma 6.24, and
- an instance $\mathbb{D}^{\mathsf{strengthen}}$ of the $(\ell+1)$-prefix-rebuilding data structure from Lemma 6.8 to turn weak descriptions of prefix-rebuilding updates into descriptions of prefix-rebuilding updates.

All of these three prefix-rebuilding data structures will always hold the same annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$. Implementation of the update operations on our data structure is simple. Upon receiving a prefix-rebuilding update $\overline{u}$ with a prefix T_{pref}, we recompute the decomposition $(T, \mathsf{bag}, \mathsf{edges})$ according to $\overline{u}$, and pass $\overline{u}$ to the inner data structures $\mathbb{D}^{\mathsf{aux}}$, $\mathbb{D}^{\mathsf{closure}}$, and $\mathbb{D}^{\mathsf{strengthen}}$. The initialization of the data structure is similarly easy. The overhead $2^{\mathcal{O}(k^8)}$ follows from the overhead $2^{\mathcal{O}((c+\ell)^2)}$ of $\mathbb{D}^{\mathsf{closure}}$.

From now on, we focus on the Refine operation. Let $T_{\mathsf{pref}} \subseteq V(T)$ be the given prefix of T which includes all bags of $\mathcal{T}$ of size $\ell + 2$. By Lemma 6.24, the fact that G has treewidth at most k, and the choice of c, there exists a c-small k-closure of $\mathsf{bags}(T_{\mathsf{pref}})$. By applying the $\mathsf{Closure}(T_{\mathsf{pref}})$ operation of Lemma 6.35, we obtain in time $2^{\mathcal{O}(k^9)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)|$

- a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathsf{pref}})$,
- the graph $\mathsf{torso}_G(X)$, and
- the sets $\mathsf{Blockages}(T_{\mathsf{pref}}, X)$ and $\mathsf{Expl}(T_{\mathsf{pref}}, X)$, and for every blockage whether it is a clique blockage or a component blockage.

Furthermore, by applying Lemmas 6.49 and 6.58, we obtain the representations of the refinement forest $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ and the grouped refinement forest $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$, with an additional running time of $k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)| + k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log |\mathcal{T}|$.

Constructing $\mathcal{T}'$

We then define the resulting tree decomposition $\mathcal{T}'$, and the transformation from $\mathcal{T}$ into $\mathcal{T}'$. Let us first compute a suitable tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$.

Lemma 6.61. *We can in time $2^{\mathcal{O}(k^3)}|T_{\mathsf{pref}}|$ compute a binary tree decomposition $\mathcal{T}^X = (T^X, \mathsf{bag}^X)$ of $\mathsf{torso}_G(X)$ so that*

- *the width of $\mathcal{T}^X$ is at most $6k + 5$,*
- *the height of $\mathcal{T}^X$ is at most $\mathcal{O}(\log |T_{\mathsf{pref}}| + k)$,*
- *for every clique $B \subseteq X$ of $\mathsf{torso}_G(X)$, there exists a leaf node $t_B \in V(T^X)$ so that $\mathsf{bag}^X(t_B) = B$, and*
- *$|V(T^X)| \leq 2^{\mathcal{O}(k)}|T_{\mathsf{pref}}|$.*

Proof. As X is c-small for $c \in \mathcal{O}(k^4)$ and (T, bag) is binary, $|X| \leq \mathcal{O}(k^4 \cdot |T_{\mathsf{pref}}|)$. Because X is a k-closure, the treewidth of $\mathsf{torso}_G(X)$ is at most $2k + 1$. We use the algorithm of Bodlaender [1996] (Theorem 3.6 in this thesis) to compute a tree decomposition $\mathcal{T}^X_{\mathsf{init}}$ of $\mathsf{torso}_G(X)$ of width at most $2k + 1$ in time $2^{\mathcal{O}(k^3)}|X| = 2^{\mathcal{O}(k^3)}|T_{\mathsf{pref}}|$. By Lemma 2.13, we can assume that $|\mathcal{T}^X_{\mathsf{init}}| \leq |X|$.

Then we use the lemma of Bodlaender and Hagerup [1998] (Lemma 6.59) to turn $\mathcal{T}^X_{\mathsf{init}}$ into a binary tree decomposition $\mathcal{T}^X_{\mathsf{log}}$ of width at most $6k + 5$, height at most $\mathcal{O}(\log |X|)$,

and with $|\mathcal{T}^X_{\mathsf{log}}| \leq \mathcal{O}(|X|)$. This takes time $\mathcal{O}(k \cdot |X|)$. Without changing these stated bounds, we can also assume that the root bag of $\mathcal{T}^X_{\mathsf{log}}$ is empty.

Finally, we obtain $\mathcal{T}^X$ by editing $\mathcal{T}^X_{\mathsf{log}}$ so that for every subset of a bag of $\mathcal{T}^X_{\mathsf{log}}$, there exists a leaf bag in $\mathcal{T}^X$ that is equal to this subset. This can be done by subdividing each edge of $\mathcal{T}^X_{\mathsf{log}}$, with the bag of the introduced subdivision node identical to the bag of its only child, and creating a new subtree of depth $\mathcal{O}(k)$ under it containing as leaves all subsets of that bag. This increases the height first by a factor of 2 and then by an additive term of $\mathcal{O}(k)$, resulting in the height of $\mathcal{T}^X$ being $\mathcal{O}(\log |X| + k)$, which as $|X| \leq \mathcal{O}(k^4 \cdot |T_{\mathsf{pref}}|)$ is at most $\mathcal{O}(\log |T_{\mathsf{pref}}| + \log k^4 + k) = \mathcal{O}(\log |T_{\mathsf{pref}}| + k)$. It increases the number of nodes by a factor of $2^{\mathcal{O}(k)}$, resulting in $|\mathcal{T}^X| \leq 2^{\mathcal{O}(k)}|X| \leq 2^{\mathcal{O}(k)}|T_{\mathsf{pref}}|$. This step can be implemented in $2^{\mathcal{O}(k)}|T_{\mathsf{pref}}|$ time. □

Now the tree decomposition $\mathcal{T}' = (T', \mathsf{bag}')$ is constructed by taking $\mathcal{T}^X = (T^X, \mathsf{bag}^X)$, and for each $\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)$, attaching $\mathcal{T}^B$ from its root as a child of a leaf node $t_B \in V(T^X)$ with $\mathsf{bag}^X(t_B) = B$.

Lemma 6.62. *$\mathcal{T}'$ is a tree decomposition of G and its width is at most ℓ.*

Proof. Let us first check the vertex and edge conditions. Because $\mathcal{T}'$ contains $\mathcal{T}^X$, it satisfies them for vertices and edges in $G[X]$. For vertices in $V(G)\setminus X$ and edges incident to them, Lemma 6.43 states that each of them is in some tree decomposition of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$, implying that each of them is in some tree decomposition of $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$.

For the connectedness condition of a vertex $v \in V(G) \setminus X$, Lemma 6.43 states that v is in exactly one tree decomposition of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$, implying that it is in exactly one tree decomposition $\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)$, implying that $\mathcal{T}'$ satisfies the condition because $\mathcal{T}^B$ satisfies it. For $v \in X$, we have that if $v \in \mathsf{bags}(\mathcal{T}^B) \cap X$, then $v \in B$, implying that v is in the root bag of $\mathcal{T}^B$, which together with the facts that $\mathcal{T}^X$ and each $\mathcal{T}^B$ satisfies the connectedness condition implies that $\mathcal{T}'$ satisfies the connectedness condition.

For the width, by Lemma 6.53 each $\mathcal{T}^B$ has width at most ℓ, and by Lemma 6.61 $\mathcal{T}^X$ has width at most ℓ, so therefore $\mathcal{T}'$ has width at most ℓ. □

Given these objects, it is straightforward to combine the representations of $\mathsf{ReFo}(T_{\mathsf{pref}}, X)$ and $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$ returned by Lemmas 6.49 and 6.58 with $\mathcal{T}^X$ to construct a weak description $\widehat{u}$ of a prefix-rebuilding update that turns $\mathcal{T}$ into $\mathcal{T}'$. We then use $\mathbb{D}^{\mathsf{strengthen}}$ to turn this into a description $\bar{u}$ of a prefix-rebuilding update, apply $\bar{u}$ to $\mathcal{T}$ and all of our prefix-rebuilding data structures, and return $\bar{u}$.

We now have implemented the Refine operation with running time

$$\begin{aligned}
&2^{\mathcal{O}(k^9)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)| && \text{(Lemma 6.35)}\\
+&2^{\mathcal{O}(k^3)} \cdot |T_{\mathsf{pref}}| && \text{(Lemma 6.61)}\\
+&k^{\mathcal{O}(1)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)| + k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log|\mathcal{T}| && \text{(Lemmas 6.49 and 6.58)}\\
\leq&2^{\mathcal{O}(k^9)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)| + k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log|\mathcal{T}| && (6.63)
\end{aligned}$$

It remains to analyze the potential change, and relate this running time to it.

Analysis of the potential

Because $\Phi_{\ell,\mathcal{T}'}(V(T') \setminus V(T^X)) = \Phi_{\ell,\mathcal{T}'}\left(\bigcup_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}},X)} V(T^B)\right)$, Lemma 6.54 implies

$$\begin{aligned}
&\Phi_{\ell,\mathcal{T}'}(V(T') \setminus V(T^X)) + |\mathsf{Expl}(T_{\mathsf{pref}}, X) \setminus T_{\mathsf{pref}}|\\
\leq&\Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a). && (6.64)
\end{aligned}$$

Therefore, it remains to bound $\Phi_{\ell,\mathcal{T}'}(V(T^X))$.

Lemma 6.65. *It holds that*

$$\Phi_{\ell,\mathcal{T}'}(V(T^X)) \leq k^{\mathcal{O}(k)} \cdot \log|\mathcal{T}| \cdot \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a)\right)$$

Proof. Our goal will be to bound $\sum_{t \in V(T^X)} \mathsf{hgt}_{T'}(t)$. In particular, we will not use the function g_ℓ from the definition of potential non-trivially, we only use the upper bound $g_\ell(\ell+1) = k^{\mathcal{O}(k)}$.

Let $t \in V(T^X)$, and let us understand how $\mathsf{hgt}_{T'}(t)$ is formed. First, by Lemma 6.61, $\mathsf{hgt}_{T^X}(t) \leq \mathcal{O}(\log|T_{\mathsf{pref}}| + k) \leq \mathcal{O}(\log|\mathcal{T}| + k)$. Then, we attach the decompositions $\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)$ on the leaves of $\mathcal{T}^X$ to form $\mathcal{T}'$. This increases the height of t by the largest height of a decomposition $\mathcal{T}^B$ attached as a descendant of t. Let $\phi\colon V(T^X) \to \mathsf{GReFo}(T_{\mathsf{pref}}, X)$ be the function that maps each $t \in V(T^X)$ to the maximum height decomposition in $\mathsf{GReFo}(T_{\mathsf{pref}}, X)$ that is attached as a descendant of t. Now,

$$\mathsf{hgt}_{T'}(t) \leq \mathcal{O}(\log|\mathcal{T}| + k) + \mathsf{hgt}(\phi(t)),$$

which by $|V(T^X)| \leq 2^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}|$ implies that

$$\sum_{t \in V(T^X)} \mathsf{hgt}_{T'}(t) \leq 2^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log |\mathcal{T}| + \sum_{t \in V(T^X)} \mathsf{hgt}(\phi(t)).$$

In fact, for any fixed constant α, it holds that

$$\sum_{t \in V(T^X)} \mathsf{hgt}_{T'}(t) \leq 2^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log |\mathcal{T}| + \sum_{t \in V(T^X)} \left(\mathsf{hgt}(\phi(t)) - \alpha \cdot \log |\mathcal{T}|\right). \tag{6.66}$$

Now, our goal is to bound $\sum_{t \in V(T^X)} \left(\mathsf{hgt}(\phi(t)) - \alpha \cdot \log |\mathcal{T}|\right)$ for the constant α of Lemma 6.57. Because $\mathsf{hgt}(\mathcal{T}^X) \leq \mathcal{O}(\log |\mathcal{T}| + k)$, each $\mathcal{T}^B$ can be a descendant of at most $\mathcal{O}(\log |\mathcal{T}| + k)$ nodes in T^X. In particular, $|\phi^{-1}(\mathcal{T}^B)| \leq \mathcal{O}(\log |\mathcal{T}| + k)$ for all $\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)$. This implies that

$$\begin{aligned}
&\sum_{t \in V(T^X)} \left(\mathsf{hgt}(\phi(t)) - \alpha \cdot \log |\mathcal{T}|\right) \\
\leq &\mathcal{O}(\log |\mathcal{T}| + k) \cdot \sum_{\mathcal{T}^B \in \mathsf{GReFo}(T_{\mathsf{pref}}, X)} \left(\mathsf{hgt}(\mathcal{T}^B) - \alpha \cdot \log |\mathcal{T}|\right)
\end{aligned}$$

By plugging in Lemma 6.57, we conclude that

$$\begin{aligned}
&\sum_{t \in V(T^X)} \left(\mathsf{hgt}(\phi(t)) - \alpha \cdot \log |\mathcal{T}|\right) \\
\leq &\mathcal{O}(\log |\mathcal{T}| + k) \cdot k^{\mathcal{O}(k)} \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a) \\
\leq &k^{\mathcal{O}(k)} \cdot \log |\mathcal{T}| \cdot \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a)
\end{aligned}$$

By combining this with (6.66), we conclude the proof. □

Let us denote by $Q = k^{\mathcal{O}(k)} \cdot \log |\mathcal{T}| \cdot \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(a)\right)$ the upper bound given by Lemma 6.65. Combining Equation (6.64) and Lemma 6.65 yields

$$\Phi_\ell(\mathcal{T}') + |\mathsf{Expl}(T_{\mathsf{pref}}, X)| \leq \Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + Q, \tag{6.67}$$

which is the desired conclusion after ignoring $|\mathsf{Expl}(T_{\mathsf{pref}}, X)|$, which will be used only for the running time bound.

Analysis of the running time

Recall that by (6.63), the running time of Refine is

$$2^{\mathcal{O}(k^9)} \cdot |\mathsf{Expl}(T_{\mathsf{pref}}, X)| + k^{\mathcal{O}(k)} \cdot |T_{\mathsf{pref}}| \cdot \log |\mathcal{T}|.$$

By re-arranging (6.67) we obtain

$$\begin{aligned} |\mathsf{Expl}(T_{\mathsf{pref}}, X)| &\leq \Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) - \Phi_\ell(\mathcal{T}') + Q \\ &\leq \Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') + Q, \end{aligned}$$

which combined with (6.63) yields the claimed bound

$$2^{\mathcal{O}(k^9)} \cdot \left(\log |\mathcal{T}| \cdot \left(|T_{\mathsf{pref}}| + \sum_{t \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}_T(t) \right) + \Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') \right)$$

on the running time. This concludes the proof of Lemma 6.60.

6.6 Height improvement

In this section we leverage the refinement operation of Section 6.5 to design a data structure that allows us to maintain a tree decomposition of small height. As in Section 6.5, we assume that the dynamic n-vertex graph G we are maintaining is promised to have treewidth at most k. Let us give the statement of the height improvement data structure.

Lemma 6.68. *Let $k \in \mathbb{Z}_{\geq 0}$ and $\ell = 6k + 5$. The $(\ell + 1)$-prefix-rebuilding data structure from Lemma 6.60 maintaining an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ can be extended to additionally support the following operation.*

- ImproveHeight()*: Updates $\mathcal{T}$ through a sequence of prefix-rebuilding updates, producing an annotated tree decomposition $\mathcal{T}'$ such that*

  $$\mathsf{hgt}(\mathcal{T}') \leq 2^{\mathcal{O}(k \log k \sqrt{\log n \log \log n})} \qquad \text{and} \qquad \Phi_\ell(\mathcal{T}') \leq k^{\mathcal{O}(k)} n \log n.$$

  Also, $\Phi_\ell(\mathcal{T}') \leq \Phi_\ell(\mathcal{T})$ and if the width of $\mathcal{T}$ is at most ℓ, then the width of $\mathcal{T}'$ is also at most ℓ.

  The running time of ImproveHeight *is bounded by $2^{\mathcal{O}(k^9)}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}')) + \mathcal{O}(1)$, and it returns the sequence of descriptions of the prefix-rebuilding updates applied.*

Lemma 6.68 is used for keeping the height of the maintained tree decomposition at most $2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$, in particular, after each update to the tree decomposition, we call ImproveHeight to ensure that the height of the decomposition stays sufficiently small. Note here that all prefix-rebuilding updates performed by ImproveHeight in order to decrease the height of the decomposition are essentially "free" in terms of amortized running time, because the running time of ImproveHeight is fully amortized by the decrease in the potential value, i.e., $\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}')$.

As an additional byproduct, Lemma 6.68 also keeps the number of nodes of the maintained tree decomposition bounded by $k^{\mathcal{O}(k)} \cdot n \log n$, which is necessary because we have the factor $\log |\mathcal{T}|$ in the running time of some operations.

The rest of this section is dedicated to the proof of Lemma 6.68.

6.6.1 Unbalanced binary trees

The idea of ImproveHeight is to show that if the height of the tree decomposition $\mathcal{T}$ is too large, then there exists a prefix T_{pref} so that applying the $\mathsf{Refine}(T_{\mathsf{pref}})$ operation of Lemma 6.60 decreases the potential $\Phi_\ell(\mathcal{T})$ significantly. The main combinatorial argument for finding such prefix T_{pref} is the following lemma about binary trees. Recall that for a rooted tree T and $t \in V(T)$, we denote $\mathsf{size}(t) = |\mathsf{desc}_T(t)|$.

Lemma 6.69. *Let $c \geq 2$ and T be a binary tree with $n \geq 2$ nodes. If the height of T is at least $2^{\Omega(\sqrt{\log n \log c})}$, then there exists a prefix T_{pref} of T so that*

$$c \cdot \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(a) \right) \leq \sum_{t \in T_{\mathsf{pref}}} \mathsf{hgt}(t).$$

Moreover, if a representation of T is already stored and supports the functions $\mathsf{hgt}(t)$ and $\mathsf{size}(t)$ for $t \in V(T)$ in $\mathcal{O}(1)$ time, then such T_{pref} can be found in $\mathcal{O}(|T_{\mathsf{pref}}|)$ time.

Proof. Let us prove the lemma for the height bound $\mathsf{hgt}(T) \geq 2^{d\sqrt{\log n \log c}}$ with the constant $d = 4$. Because a tree with n nodes can have height at most n, we can assume that $d\sqrt{\log n \log c} \leq \log n$, which implies that $\log c \leq \log n / d^2$, which in turn implies that $\log c \leq \sqrt{\log n \log c}/d$.

Let P be a longest root-leaf path in T. Note that this implies $|P| = \mathsf{hgt}(T)$ and $\sum_{t \in P} \mathsf{hgt}(t) \geq |P|^2/2$. The idea is that the prefix T_{pref} will be obtained as a union of the prefix P, and prefixes obtained by recursive calls (i.e., induction) on subtrees rooted at the appendices of P that satisfy the preconditions of the lemma. For this, we say that an

appendix $a \in \mathsf{app}(P)$ is *deep* if $\mathsf{hgt}(a) \geq \mathsf{hgt}(T)/(4c)$ and *small* if $\mathsf{size}(a) \leq (n \cdot 4c)/|P|$. We show that the lemma can be recursively applied on appendices that are both deep and small.

Claim 6.70. *If an appendix $a \in \mathsf{app}(P)$ is deep and small, then the subtree rooted at it satisfies the preconditions of the lemma statement.*

Proof of the claim. Because $\mathsf{hgt}(a) \geq \mathsf{hgt}(T)/(4c)$, $\mathsf{hgt}(T) \geq 2^{d \log c}$, and $d = 4$, it must hold that $\mathsf{hgt}(a) \geq 2$ and therefore also $\mathsf{size}(a) \geq 2$.

Our goal is to prove that $\mathsf{hgt}(a) \geq 2^{d\sqrt{\log(\mathsf{size}(a)) \log c}}$, which holds if

$$\begin{aligned}
& & 2^{d\sqrt{\log n \log c} - \log(4c)} &\geq 2^{d\sqrt{\log(\mathsf{size}(a)) \log c}} \\
\Leftrightarrow & & d\sqrt{\log n \log c} - \log(4c) &\geq d\sqrt{\log(\mathsf{size}(a)) \log c} \\
\Leftrightarrow & & \sqrt{\log n \log c} - \sqrt{\log(\mathsf{size}(a)) \log c} &\geq (\log(4c))/d \\
\Leftarrow & & \sqrt{\log n} - \sqrt{\log(\mathsf{size}(a))} &\geq (4\sqrt{\log c})/d.
\end{aligned}$$

Then, we lower bound the left hand side by

$$\begin{aligned}
\sqrt{\log n} - \sqrt{\log(\mathsf{size}(a))} \geq & \sqrt{\log n} - \sqrt{\log n + \log(4c) - \log |P|} \\
\geq & \sqrt{\log n} - \sqrt{\log n + \log(4c) - d\sqrt{\log n \log c}} \\
\geq & \sqrt{\log n} - \sqrt{\log n - d\sqrt{\log n \log c}/2} \\
\geq & d\sqrt{\log c}/4.
\end{aligned}$$

Now, $d\sqrt{\log c}/4 \geq (4\sqrt{\log c})/d$ holds because $d = 4$. ◁

Let T_{pref} be the prefix obtained as the union of P and the prefixes returned by recursive calls on subtrees rooted at appendices of P that are both deep and small. By induction, we have that

$$c \cdot \left(|T_{\mathsf{pref}} \setminus P| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}}) \setminus \mathsf{app}(P)} \mathsf{hgt}(a) \right) \leq \sum_{t \in T_{\mathsf{pref}} \setminus P} \mathsf{hgt}(t). \tag{6.71}$$

Then we bound the sum of the heights of appendices of P that are not small. Let $A_{\mathsf{big}} \subseteq \mathsf{app}(P)$ be set the appendices of P that are not small. As the subtrees rooted at them have size at least $(n \cdot 4c)/|P|$ and are disjoint, we have that $|A_{\mathsf{big}}| \leq |P|/(4c)$, implying $\sum_{a \in A_{\mathsf{big}}} \mathsf{hgt}(a) \leq |P|^2/(4c)$.

Then we bound the sum of the heights of appendices of P that are not deep. Let $A_{\mathsf{shallow}} \subseteq \mathsf{app}(P)$ be the set of appendices of P that are not deep. By the definition of deep appendices and the fact that $|A_{\mathsf{shallow}}| \leq |P|$, we get $\sum_{a \in A_{\mathsf{shallow}}} \mathsf{hgt}(a) \leq |P|^2/(4c)$.

Because $\mathsf{app}(P) \setminus T_{\mathsf{pref}} = A_{\mathsf{big}} \cup A_{\mathsf{shallow}}$ and $\sum_{t \in P} \mathsf{hgt}(t) \geq |P|^2/2$, we obtain

$$c \cdot \left(|P| + \sum_{a \in \mathsf{app}(P) \setminus T_{\mathsf{pref}}} \mathsf{hgt}(a) \right) \leq |P|^2/2 \leq \sum_{t \in P} \mathsf{hgt}(t). \tag{6.72}$$

By combining (6.71) and (6.72), we get that T_{pref} satisfies the desired conclusion. We observe that the recursive procedure for finding such T_{pref} can be implemented in time $\mathcal{O}(|T_{\mathsf{pref}}|)$ when having access to the $\mathsf{hgt}(t)$ and $\mathsf{size}(t)$ functions. □

6.6.2 Reducing the height

We then apply Lemma 6.69 to prove Lemma 6.68.

Proof of Lemma 6.68. Recall that we maintain an instance $\mathbb{D}^{\mathsf{aux}}$ of the $(\ell+1)$-prefix-rebuilding data-structure from Lemma 6.7, in particular, in constant time we can access the height $\mathsf{hgt}(t)$ of each node t and the subtree size $\mathsf{size}(t)$.

First, suppose $\Phi_\ell(\mathcal{T}) \geq c_d \cdot c_\Phi \cdot n \log n$, where $c_\Phi = g_\ell(\ell+1) = k^{\mathcal{O}(k)}$ and c_d is a constant depending on the constants in the $\mathcal{O}$-notation in the Bodlaender-Hagerup Lemma (Lemma 6.59), in particular, so that the $\sum_{t \in V(T_{\mathsf{log}})} \mathsf{hgt}_{T_{\mathsf{log}}}(t) \leq \frac{c_d}{2} \cdot |\mathcal{T}| \log |\mathcal{T}|$ for the tree decomposition $(T_{\mathsf{log}}, \mathsf{bag}_{\mathsf{log}})$ returned by it when given a tree decomposition $\mathcal{T}$.

In this case, we compute a new tree decomposition $\mathcal{T}' = (T', \mathsf{bag}')$ from scratch by first constructing the graph G explicitly from the edges function, then applying the algorithm of Bodlaender (Theorem 3.6) and then the Bodlaender-Hagerup Lemma (Lemma 6.59). The width of $\mathcal{T}'$ is at most $3k+2 \leq \ell$, the height of it is at most $\mathcal{O}(\log n)$, and by the definition of c_d and c_Φ, its potential is at most

$$\Phi_\ell(\mathcal{T}') \leq \frac{c_d \cdot c_\Phi}{2} \cdot n \log n \leq \Phi_\ell(\mathcal{T})/2.$$

Therefore, $\mathcal{T}'$ satisfies all the properties required by the lemma statement, so with the help of the $\mathbb{D}^{\mathsf{strengthen}}$ data structure we apply a prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}'$ (with the prefixes of the description being $T_{\mathsf{pref}} = V(T)$ and $T'_{\mathsf{pref}} = V(T')$). The

running time of this is

$$2^{\mathcal{O}(k^8)}(|\mathcal{T}| + |\mathcal{T}'|) + 2^{\mathcal{O}(k^3)}n \leq 2^{\mathcal{O}(k^8)}\Phi_\ell(\mathcal{T}) \leq 2^{\mathcal{O}(k^8)}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}')).$$

Then we can assume $\Phi_\ell(\mathcal{T}) < c_d c_\Phi \cdot n \log n$, implying in particular that $|\mathcal{T}| < k^{\mathcal{O}(k)} n \log n$. Let us choose $c_1 = k^{\mathcal{O}(k)} \cdot \log |\mathcal{T}| = k^{\mathcal{O}(k)} \cdot \log n$ so that the inequality in the statement of Lemma 6.60 holds in the form

$$\Phi_\ell(\mathcal{T}') \leq \Phi_{\ell,\mathcal{T}}(V(T) \setminus T_{\mathsf{pref}}) + c_1 \cdot \left(|T_{\mathsf{pref}}| + \sum_{t \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(t) \right),$$

which can be re-arranged to the form

$$\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') \geq \Phi_{\ell,\mathcal{T}}(T_{\mathsf{pref}}) - c_1 \cdot \left(|T_{\mathsf{pref}}| + \sum_{t \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(t) \right). \tag{6.73}$$

Let also c_2 be the constant hidden by the Ω-notation in Lemma 6.69. If

$$\mathsf{hgt}(T) \leq 2^{c_2 \cdot \sqrt{\log n \log(2c_1)}} \leq 2^{\mathcal{O}(\sqrt{k \log k \log n \log\log n})},$$

then T already satisfies the claimed height bound and we return without applying any prefix-rebuilding updates.

Otherwise, we apply Lemma 6.69 with the tree T and the constant $c = 2c_1$. As we satisfy the preconditions, it returns a prefix T_{pref} so that

$$2c_1 \cdot \left(|T_{\mathsf{pref}}| + \sum_{a \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(a) \right) \leq \sum_{t \in T_{\mathsf{pref}}} \mathsf{hgt}(t) \leq \Phi_{\ell,\mathcal{T}}(T_{\mathsf{pref}}), \tag{6.74}$$

meaning in particular that if we apply the refine operation of Lemma 6.60 with this prefix, then by the combination of (6.73) and (6.74), it holds that

$$\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') \geq c_1 \cdot \left(|T_{\mathsf{pref}}| + \sum_{t \in \mathsf{app}(T_{\mathsf{pref}})} \mathsf{hgt}(t) \right)$$

Moreover, as $c_1 \geq \log |\mathcal{T}|$, in this case the running time of the refine operation is bounded by $2^{\mathcal{O}(k^9)}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))$, which is also an upper bound for the size of the prefix-rebuilding update given by it. Because $\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') \geq |T_{\mathsf{pref}}|$, the running time of the application of Lemma 6.69 can also be bounded by $\mathcal{O}(|T_{\mathsf{pref}}|) \leq \mathcal{O}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))$.

As $\Phi_\ell(\mathcal{T}') < \Phi_\ell(\mathcal{T})$, we managed to decrease the potential in time $2^{\mathcal{O}(k^9)}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))$. The resulting tree decomposition $\mathcal{T}'$ has width at most ℓ if $\mathcal{T}$ has width at most ℓ. This did not necessarily improve the height to the desired bound, so we repeat this process until the height is improved. The total running time is $2^{\mathcal{O}(k^9)}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}''))$, where $\mathcal{T}''$ is the final obtained tree decomposition with the desired height bound. □

6.7 Putting things together

We are now ready to complete the proof of Theorem 1.4. We will do this in three steps, first giving a version that simply maintains a tree decomposition when given a promise that treewidth is at most k, then adding the "Treewidth too large" feature, and finally adding support for CMSO_2.

6.7.1 Maintaining a tree decomposition

We now start with the basic maintenance of annotated tree decompositions. The point of the following lemma is that the data structure outputs the sequence of descriptions of prefix-rebuilding updates, so we can later build more features on top of it.

Lemma 6.75. *There is a data structure, that for an integer k and a dynamic n-vertex graph G that is promised to have treewidth at most k at all times, maintains an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k+6$ by using prefix-rebuilding updates, and supports the following operations.*

- $\mathsf{Initalize}(G, k)$*: Given an edgeless n-vertex graph G and an integer $k \geq 0$, initalize the data structure with them. Runs in amortized $2^{\mathcal{O}(k^9)}n$ time and returns the initial annotated tree decomposition $\mathcal{T}$.*
- $\mathsf{AddEdge}(u, v)$*: Given two vertices $u, v \in V(G)$ with $uv \notin E(G)$, add the edge uv to G. Runs in amortized $2^{\mathcal{O}(k^9 + k \log k \sqrt{\log n \log\log n})}$ time and returns the sequence of prefix-rebuilding updates used to update $\mathcal{T}$.*
- $\mathsf{DeleteEdge}(u, v)$*: Given two vertices $u, v \in V(G)$ with $uv \in E(G)$, delete the edge uv from G. Runs in worst-case $2^{\mathcal{O}(k^8 + k \log k \sqrt{\log n \log\log n})}$ time and returns the sequence of prefix-rebuilding updates used to update $\mathcal{T}$.*

After each operation, the resulting tree decomposition has width at most $6k+5$, but the intermittent tree decompositions in the sequences of updates may have width up to $6k+6$.

This subsection is dedicated to the proof of Lemma 6.75. We first describe the data structure and analyze the running times and potential changes of individual operations, and then present amortized analysis of the whole data structure.

Let $\ell = 6k + 5$. In our data structure we store

- the annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $\ell + 1$,
- an instance $\mathbb{D}^{\mathsf{refine}}$ of the $(\ell + 1)$-prefix-rebuilding data structure from Lemmas 6.60 and 6.68 initialized with the same value of k as our main data structure, supporting the operations $\mathsf{Refine}(T_{\mathsf{pref}})$ and $\mathsf{ImproveHeight}()$, and having overhead $2^{\mathcal{O}(k^8)}$,
- an instance $\mathbb{D}^{\mathsf{aux}}$ of the $(\ell + 1)$-prefix-rebuilding data structure from Lemma 6.7, supporting the operations $\mathsf{hgt}(t)$ for $t \in V(T)$ and $\mathsf{forget}(v)$ for $v \in V(G)$, and having overhead $\mathcal{O}(1)$,
- an instance $\mathbb{D}^{\mathsf{strengthen}}$ of the $(\ell+1)$-prefix-rebuilding data structure from Lemma 6.8, supporting the operation $\mathsf{Strengthen}(\widehat{u})$ for a weak description $\widehat{u}$ of a prefix-rebuilding update, and having overhead $\mathcal{O}(1)$.

All of the data structures store the same annotated tree decomposition $\mathcal{T}$. In particular, all prefix-rebuilding updates performed by $\mathbb{D}^{\mathsf{refine}}$ are applied also to the instance of $\mathcal{T}$ that we store, and are forwarded to $\mathbb{D}^{\mathsf{aux}}$ and $\mathbb{D}^{\mathsf{strengthen}}$. After the initialization and after each operation, we maintain the invariant that $\mathcal{T}$ is an annotated tree decomposition of G of width at most ℓ, height at most $h = 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$, and has $|\mathcal{T}| \leq k^{\mathcal{O}(k)} \cdot n \log n$.

Initialization

As for the initialization, because G has no edges, we can initialize $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ with T being a complete binary tree (of height $\mathcal{O}(\log n)$) with n nodes and each bag containing a different vertex of $V(G)$. Obviously, for each $t \in V(T)$, we have $\mathsf{edges}(t) = \emptyset$. The initializations of $\mathbb{D}^{\mathsf{refine}}$, $\mathbb{D}^{\mathsf{aux}}$, and $\mathbb{D}^{\mathsf{strengthen}}$ take $2^{\mathcal{O}(k^8)} \cdot n$ time. The initial decomposition $\mathcal{T}$ satisfies that $\Phi_\ell(\mathcal{T}) \leq \mathcal{O}(kn)$ since the average height of a node in T is $\mathcal{O}(1)$, and $g_\ell(1) = \mathcal{O}(k)$.

Edge addition

Then consider the edge addition operation, where we add an edge uv. Towards that goal, we must first ensure that the edge condition is satisfied for the new edge, i.e., both

u and v belong to the same bag of $\mathcal{T}$. Let $t_u = \mathsf{forget}_{\mathcal{T}}(u)$ and $t_v = \mathsf{forget}_{\mathcal{T}}(v)$ be the forget-nodes of u and v, respectively, in $\mathcal{T}$. Then, let P_u be the nodes of T on the path from t_u to the root, and P_v the the nodes of T on the path from t_v to the root. We update $\mathcal{T}$ by adding v to all bags of the nodes $P_u \cup P_v \setminus \{t_v\}$. Note that the sets P_u and P_v can be computed in time $\mathcal{O}(|P_u| + |P_v|)$ by walking up from t_u and t_v.

This update of $\mathcal{T}$ together with adding the edge can be implemented with prefix-rebuilding updates in two steps. First, it is simple to construct a weak description $\widehat{u}$ of a prefix-rebuilding update of size $|P_u \cup P_v|$ that adds v to all bags on $P_u \cup P_v \setminus \{t_v\}$, and by using $\mathbb{D}^{\mathsf{strengthen}}$ apply it to our decomposition $\mathcal{T}$ and to our prefix-rebuilding data structures. Now, both u and v are in $\mathsf{bag}(t_u)$, so we can add the edge uv by a prefix-rebuilding update that simply adds uv to $\mathsf{edges}(t_u)$. Note that indeed, t_u will be the forget-node of uv because t_u is still the forget-node of u. A description of size $|P_u|$ of such prefix-rebuilding update is easy to construct, and we apply it again to $\mathcal{T}$ and all prefix-rebuilding data structures.

However, now the decomposition may have width $\ell + 1$. To counteract this, we call the $\mathsf{Refine}(P_u \cup P_v)$ operation of $\mathbb{D}^{\mathsf{refine}}$. Note that $P_u \cup P_v$ covers all bags of size $\ell + 2$, so the call satisfies the precondition of Lemma 6.60 and the annotated tree decomposition produced by Refine has width at most ℓ. However, the decomposition might now have too large height or size. We resolve this issue by invoking the height improvement operation ($\mathsf{ImproveHeight}$), resulting in $\mathcal{T}$ having width at most ℓ, height at most $h = 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$, and size at most $|\mathcal{T}| \leq k^{\mathcal{O}(k)} \cdot n \log n$. The final tree decomposition satisfies all of the prescribed invariants. We then make sure that our annotated tree decomposition $\mathcal{T}$ and the other prefix-rebuilding data structures correspond to the annotated tree decomposition stored by $\mathbb{D}^{\mathsf{refine}}$ by also applying the same prefix-rebuilding updates to them, and finally output the sequence of descriptions of prefix-rebuilding updates applied.

Let us then analyze the running time and the potential change. For this, let $\mathcal{T}_0$ be the tree decomposition before the edge addition operation, $\mathcal{T}_1$ the tree decomposition after adding the edge uv but before improving its width, $\mathcal{T}_2$ the tree decomposition after applying Refine, and $\mathcal{T}_3$ the final tree decomposition after applying $\mathsf{ImproveHeight}$. As $|P_u \cup P_v| \leq h$, the first step of turning $\mathcal{T}_0$ into $\mathcal{T}_1$ runs in $2^{\mathcal{O}(k^8)}h$ time. Furthermore, $\Phi_\ell(\mathcal{T}_1) \leq \Phi_\ell(\mathcal{T}_0) + k^{\mathcal{O}(k)}h^2$, because only the bags in $P_u \cup P_v$ change. The Refine operation runs in time

$$\begin{aligned} &2^{\mathcal{O}(k^9)} \cdot \left(h^2 \cdot \log|\mathcal{T}_1| + \Phi_\ell(\mathcal{T}_1) - \Phi_\ell(\mathcal{T}_2)\right) \\ \leq &2^{\mathcal{O}(k^9)} \cdot (h^2 \cdot \log n + \Phi_\ell(\mathcal{T}_0) - \Phi_\ell(\mathcal{T}_2)), \end{aligned}$$

and we have that

$$\begin{aligned}\Phi_\ell(\mathcal{T}_2) &\leq \Phi_\ell(\mathcal{T}_1) + k^{\mathcal{O}(k)} \cdot h^2 \cdot \log|\mathcal{T}_1| \\ &\leq \Phi_\ell(\mathcal{T}_0) + k^{\mathcal{O}(k)} \cdot h^2 \cdot \log n.\end{aligned}$$

Then, the ImproveHeight operation runs in time

$$\begin{aligned}&2^{\mathcal{O}(k^9)} \cdot (\Phi_\ell(\mathcal{T}_2) - \Phi_\ell(\mathcal{T}_3)) \\ \leq &2^{\mathcal{O}(k^9)} \cdot (\Phi_\ell(\mathcal{T}_0) + h^2 \cdot \log n - \Phi_\ell(\mathcal{T}_3)),\end{aligned}$$

and we have that

$$\begin{aligned}\Phi_\ell(\mathcal{T}_3) &\leq \Phi_\ell(\mathcal{T}_2) \\ &\leq \Phi_\ell(\mathcal{T}_0) + k^{\mathcal{O}(k)} \cdot h^2 \cdot \log n.\end{aligned}$$

In summary, the total running time of the edge addition operation is bounded by

$$2^{\mathcal{O}(k^9)} \cdot \max(h^2 \cdot \log n, \Phi_\ell(\mathcal{T}_0) - \Phi_\ell(\mathcal{T}_3)),$$

and the potential increases by at most $k^{\mathcal{O}(k)} \cdot h^2 \cdot \log n$, but can decrease arbitrarily.

Edge deletion

Deleting an edge uv from G is much simpler. We do not change the tree decomposition (T, bag) at all, only remove uv from the set $\mathsf{edges}(\mathsf{forget}_{\mathcal{T}}(uv))$. Recall that $\mathsf{forget}_{\mathcal{T}}(uv)$ is either $\mathsf{forget}_{\mathcal{T}}(u)$ or $\mathsf{forget}_{\mathcal{T}}(v)$, whichever has smaller height, so we can locate it by using the data structure $\mathbb{D}^{\mathsf{aux}}$. It is easy to construct a description of a prefix-rebuilding update of size at most h that removes uv from $\mathsf{edges}(\mathsf{forget}_{\mathcal{T}}(uv))$, so we do that, and apply it to our decomposition $\mathcal{T}$ and our prefix-rebuilding data structures. This runs in worst-case time

$$2^{\mathcal{O}(k^8)}h = 2^{\mathcal{O}(k^8 + k \log k \sqrt{\log n \log\log n})},$$

and does not change the potential.

Amortized analysis

Let $\mathcal{T}_0$ be the initial annotated tree decomposition, and $\mathcal{T}_i$ the stored annotated tree decomposition after the i-th edge addition update. Now, because the edge deletion

operation does not change the potential, the total running time of the initialization operation and the first p edge addition operations is

$$2^{\mathcal{O}(k^8)}n + 2^{\mathcal{O}(k^9)} \cdot \left(p \cdot h^2 \cdot \log n + \sum_{i=1}^{p} \Phi_\ell(\mathcal{T}_{i-1}) - \Phi_\ell(\mathcal{T}_i) \right). \tag{6.76}$$

Because $\Phi_\ell(\mathcal{T}_i)$ is non-negative, $\Phi_\ell(\mathcal{T}_0) \leq \mathcal{O}(kn)$, and $\Phi_\ell(\mathcal{T}_i) \leq \Phi_\ell(\mathcal{T}_{i-1}) + k^{\mathcal{O}(k)} \cdot h^2 \cdot \log n$, we get that

$$\sum_{i=1}^{p} \Phi_\ell(\mathcal{T}_{i-1}) - \Phi_\ell(\mathcal{T}_i) \leq \mathcal{O}(kn) + p \cdot k^{\mathcal{O}(k)} \cdot h^2 \cdot \log n,$$

implying that (6.76) can be bounded by

$$2^{\mathcal{O}(k^9)} \cdot (n + p \cdot h^2 \log n) \leq 2^{\mathcal{O}(k^9)} \cdot (n + p \cdot 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}),$$

which gives the desired amortized running times of the initialization and edge addition operations. This finishes the proof of Lemma 6.75.

6.7.2 Additional features

We then complete the proof of Theorem 1.4.

As the first step, we make the data structure of Lemma 6.75 reject edge additions that would increase the treewidth above k. For this, we use the prefix-rebuilding data structure of Lemma 6.14 for maintaining the exact treewidth of G.

Now we give the slightly enhanced version of the data structure of Lemma 6.75.

Lemma 6.77. *There is a data structure similar to that of Lemma 6.75, but instead of assuming that the treewidth of G is always at most k, the* AddEdge *operation returns* "Treewidth too large" *and does not update the annotated tree decomposition or the graph G if the update would increase the treewidth of G above k.*

Proof. We maintain two versions of the data structure of Lemma 6.75, one with the original value of k, called $\mathbb{D}_k$, and one with $k+1$ instead, called $\mathbb{D}_{k+1}$. Let $\ell = 6k+5$ and $\ell' = 6(k+1)+5$. We also maintain an instance $\mathbb{D}^{\mathsf{BK}}$ of the $(\ell'+1)$-prefix-rebuilding data structure of Lemma 6.17 for maintaining whether treewidth is at most k, with the original value of k, relaying to it all prefix-rebuilding updates applied by $\mathbb{D}_{k+1}$ so that it maintains the same annotated tree decomposition as $\mathbb{D}_{k+1}$.

Now, the initialization and edge deletion operations are simply relied to both $\mathbb{D}_k$ and $\mathbb{D}_{k+1}$, but the edge addition is first relied only to $\mathbb{D}_{k+1}$. After applying the edge addition

to $\mathbb{D}_{k+1}$, we can determine with $\mathbb{D}^{\mathsf{BK}}$ whether it would increase the treewidth of G above k, and if so, we simply apply the edge deletion operation for $\mathbb{D}_{k+1}$ to reverse the addition and return "Treewidth too large". If the treewidth of G stays at most k, we relay the edge addition operation also to $\mathbb{D}_k$.

The sequences of descriptions of prefix-rebuilding updates returned are those returned by $\mathbb{D}_k$. We can observe that this extra maintenance keeps the running time of the data structure within the same bounds as claimed in Lemma 6.75. □

We then apply the technique of *delaying invariant-breaking insertions* by Eppstein et al. [1996] to make our data structure resilient for the treewidth of G growing above k. This makes the running time bounds slightly weaker in that now the running time of edge deletion is also amortized.

Lemma 6.78. *There is a data structure, that for an integer k and a dynamic n-vertex graph G, maintains either*

- *an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k + 6$, in which case G has treewidth at most k, or*
- *a marker* "Treewidth too large"*, in which case G has treewidth more than k.*

The data structure supports the following operations.

- $\mathsf{Initalize}(G, k)$*: Given an edgeless n-vertex graph G and an integer $k \geq 0$, initalize the data structure with them. Runs in amortized $2^{\mathcal{O}(k^9)}n$ time and returns the initial annotated tree decomposition $\mathcal{T}$.*
- $\mathsf{UpdateEdge}(u, v)$*: Given two vertices $u, v \in V(G)$, add the edge uv to G if $uv \notin E(G)$, and otherwise remove the edge uv from G. Runs in amortized $2^{\mathcal{O}(k^9 + k \log k \sqrt{\log n \log \log n})}$ time, and returns either* "Treewidth too large"*, indicating that the treewidth of G is more than k, or a sequence of prefix-rebuilding updates used to update $\mathcal{T}$, relative to the previous state when $\mathsf{tw}(G) \leq k$.*

After each update operation that does not return "Treewidth too large"*, the resulting tree decomposition has width at most $6k + 5$, but the intermittent tree decompositions in the sequences of updates may have width up to $6k + 6$.*

Proof. First, with standard techniques using binary search trees, we can maintain a data structure with initialization time $\mathcal{O}(1)$ and update time $\mathcal{O}(\log n)$ that can return, given

$u, v \in V(G)$, whether $uv \in E(G)$. We maintain an explicit representation of $E(G)$ at all times like this. It can be used for telling if an edge update is an addition or deletion.

We use an instantiation $\mathbb{D}$ of the data structure of Lemma 6.77, initialized with the same edgeless graph G and integer k, to maintain an annotated an annotated tree decomposition $\mathcal{T}$ of a subgraph G' of G with $V(G') = V(G)$, with the invariants that

1. $\mathsf{tw}(G') \leq k$ and
2. if $E(G) \setminus E(G')$ is non-empty, then there exists $uv \in E(G) \setminus E(G')$ so that adding uv to G' would increase its treewidth above k.

Now, whenever $G = G'$, we have that $\mathsf{tw}(G) \leq k$ and the tree decomposition $\mathcal{T}$ held by $\mathbb{D}$ is a tree decomposition of G. When $G \neq G'$, it holds that $\mathsf{tw}(G) > k$.

This can be implemented as follows. We relay all edge additions also to the data structure $\mathbb{D}$, but if they are rejected, add them to a binary search tree that represents the set $E(G) \setminus E(G')$.

For edge deletion operations, we first check if the edge is in $E(G) \setminus E(G')$ and in that case delete it from just the binary search tree. If the edge is in G' we relay the deletion operation to $\mathbb{D}$. In both cases, at the end of the edge deletion operation we iteratively pick edges from $E(G) \setminus E(G')$ and attempt to insert them to $\mathbb{D}$, stopping at the first rejected attempt, at which point we know that the case of Item 2 holds.

Whenever $E(G) \setminus E(G')$ is empty after an update operation, we output the sequence of all descriptions of prefix-rebuilding updates that $\mathbb{D}$ has returned after the previous time $E(G) \setminus E(G')$ held.

The amortized update time is the same as for $\mathbb{D}$ up to a constant factor, because for any sequence of p updates to G, we apply at most $2p$ operations on $\mathbb{D}$, and the $\mathcal{O}(\log n)$ overhead for maintaining the binary search trees is dominated by the $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$ running time of $\mathbb{D}$. □

We then finish the proof of Theorem 1.4, which we restate here.

Theorem 1.4. *There is a data structure that is initialized with an initially edgeless n-vertex dynamic graph G and a parameter k. The data structure supports updating G by edge insertions and deletions, and maintains a tree decomposition of G of width at most $6k+5$ whenever the treewidth of G is at most k. When the treewidth of G is more than k, the data structure contains a marker* "Treewidth too large"*. The amortized initialization time is $2^{k^{\mathcal{O}(1)}}n$ and the amortized update time is $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$.*

Moreover, the data structure can be provided a CMSO_2 *sentence* φ *upon initialization, in which case it maintains whether* φ *is true in* G *whenever the marker* "Treewidth too large" *is not present. In this case, the amortized initialization time is* $f(k,\varphi)\cdot n$ *and the amortized update time is* $f(k,\varphi)\cdot 2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$*, where* f *is a computable function.*

Proof. The first part of the statement of the theorem is immediate from Lemma 6.78, with the $k^{\mathcal{O}(1)}$ bounds being $\mathcal{O}(k^9)$.

For the second part, let $\mathbb{D}$ be an instantiation of the data structure of Lemma 6.78 with the same parameter k. We relay all updates to $\mathbb{D}$.

Now, let φ be the CMSO_2 sentence given upon initialization. We use Lemma 6.16 to instantiate a $(6k+6)$-prefix-rebuilding data structure $\mathbb{D}^{\mathsf{CMSO}_2}$, with overhead $f(k,\varphi)$ for a computable function f, that can be queried whether φ is true in the graph held by it. We relay all prefix-rebuilding updates returned by $\mathbb{D}$ also to $\mathbb{D}^{\mathsf{CMSO}_2}$. Now, whenever $\mathsf{tw}(G)\leq k$, the annotated tree decompositions held by $\mathbb{D}$ and $\mathbb{D}^{\mathsf{CMSO}_2}$ are the same, so we can use $\mathbb{D}^{\mathsf{CMSO}_2}$ to report if φ is true in G. This adds an overhead of $f(k,\varphi)$ to both the initialization and the updates, but does not change the running times otherwise. □

Chapter 7

Fast 2-approximation algorithms for rankwidth and branchwidth

In this chapter we show that both the graph-theoretic and the algorithmic ideas introduced in Chapter 4 for 2-approximating treewidth can be generalized for 2-approximating branchwidth of connectivity functions. We first show that branch decompositions of connectivity functions can be improved with a similar improvement operation as was used in Chapter 4, until the decomposition has 2-approximately optimal width. Then, we show that these improvement operations can be implemented efficiently for those connectivity functions whose branch decompositions support efficient dynamic programming for computing certain objects, analogous to the dynamic programming in Section 4.4.

We then show that this efficient dynamic programming can indeed be implemented for the graph width parameters rankwidth and branchwidth. This results in 2-approximation algorithms for them. The most significant result of this chapter is the following.

Theorem 1.5. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{2^{\mathcal{O}(k)}}n^2$ either outputs a rank decomposition of G of width at most $2k$ or determines that the rankwidth of G is larger than k.*

The running time is quadratic in the number of vertices n, because in the context of rankwidth, no result analogous to Bodlaender's technique (Theorem 3.8) is known. In particular, the algorithm works by n iterations, so that in each iteration we add one vertex to the graph and to the rank decomposition from the previous iteration, obtaining a rank decomposition of width at most $2k+1$, and then use the improvement technique to improve the width to $2k$. Therefore, by showing that the width can be improved from $2k+1$ to $2k$ in time $2^{2^{\mathcal{O}(k)}}n$ by the improvement technique, we obtain a $2^{2^{\mathcal{O}(k)}}n^2$ time algorithm.

For branchwidth of graphs, we can start with a 3-approximately optimal decomposition by first 2-approximating treewidth with the algorithm of Theorem 1.1, and therefore we obtain a very similar looking result as was obtained for treewidth in Chapter 4.

Theorem 1.7. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k)}n$ either outputs a branch decomposition of G of width at most $2k$ or determines that the branchwidth of G is larger than k.*

Organization

The rest of this chapter is organized as follows. We first introduce some additional notation related to branch decompositions in Section 7.1. Then, in Section 7.2 we show our main combinatorial results, in particular, the improvement operation for branch decompositions of connectivity functions and its main properties. Then, in Section 7.3 we give our algorithmic framework, stating that branch decompositions of connectivity functions can be efficiently improved until they are 2-approximately optimal if they support certain dynamic programming. We give the implementation of this dynamic programming for rankwidth in Section 7.4, resulting in Theorem 1.5, and for branchwidth of graphs in Section 7.5, resulting in Theorem 1.7.

7.1 Notation on branch decompositions

Recall the definitions of connectivity functions and branch decompositions from Section 2.3. In this section we introduce some additional notation for manipulating branch decompositions.

Let V be a set. A *bipartition* of V is a pair (C_1, C_2) of disjoint subsets $C_1, C_2 \subseteq V$ so that $C_1 \cup C_2 = V$. We allow a bipartition of V to contain an empty set (or if V is empty, both C_1 and C_2 are empty), so a bipartition is not necessarily a partition as defined earlier. Similarly, a *tripartition* of V is a triple (C_1, C_2, C_3) of disjoint subsets $C_i \subseteq V$ so that their union equals V.

Let $r = uv \in E(T)$ be an edge of a branch decomposition $\mathcal{T} = (T, \lambda)$. We introduce notation with the intuition that $\mathcal{T}$ is treated as rooted at the edge r. The r-subtree of a node $w \in V(T)$ is the subtree of T induced by nodes $x \in V(T)$ such that w is on the unique shortest path from x to $\{u, v\}$. Note that the r-subtree of w always contains w, and contains at most one of u and v. For a node $w \in V(T)$, we denote by $\mathcal{T}_r[w] \subseteq V$ the

subset of V mapped by λ to the leaves in the r-subtree of w. Note that $\mathcal{T}_r[u] = \mathcal{T}[uv]$, $\mathcal{T}_r[v] = \mathcal{T}[vu]$, and for any $w \in V(T)$ it holds that either $\mathcal{T}_r[w] \subseteq \mathcal{T}_r[u]$ or $\mathcal{T}_r[w] \subseteq \mathcal{T}_r[v]$.

A node x is an r-ancestor of a node w if w is in the r-subtree of x. The r-parent of a node $w \in V(T) \setminus \{u, v\}$ is the node next to w on the unique shortest path from w to $\{u, v\}$. If x is an r-ancestor of w, then w is an r-descendant of x, and if p is the r-parent of w, then w is an r-child of p. Note that every non-leaf node has exactly two r-children, and leaf nodes have no r-children.

7.2 Combinatorial framework

In this section we prove our combinatorial results, which essentially state that the improvement operation of Chapter 4 can be adapted to the setting of branch decompositions of connectivity functions. Throughout we use the convention that $f : 2^V \to \mathbb{Z}_{\geq 0}$ is a connectivity function.

7.2.1 Improvement operation

The central concept of our algorithm is the improvement operation, analogous to the improvement operation of Chapter 4. Before defining it, we need the definition of a *partial branch decomposition.*

Definition 7.1 (Partial branch decomposition)**.** *A partial branch decomposition on a set C is a pair (T, λ), where T is a cubic tree and λ is an injection from C to the leaves of T.*

In particular, a partial branch decomposition can have leaves to which no elements are mapped. Let (T, λ) be a branch decomposition of f and let $C_i \subseteq V$. We denote by $(T, \lambda{\restriction_{C_i}})$ the partial branch decomposition on the set C_i obtained by restricting the mapping λ to only C_i. Then we can define the improvement operation (see Figure 7.1).

We define a *tripartition* of a set V to be a triple (C_1, C_2, C_3) of disjoint subsets $C_i \subseteq V$ so that their union equals V. Note that $\{C_1, C_2, C_3\}$ is not necessarily a partition of V, because it can contain empty sets. Similarly, a *bipartition* of V is a pair (C_1, C_2) of disjoint subsets $C_1, C_2 \subseteq V$ so that $C_1 \cup C_2 = V$.

Definition 7.2 (Improvement)**.** *Let $\mathcal{T} = (T, \lambda)$ be a branch decomposition, $uv \in E(T)$ an edge of T, and (C_1, C_2, C_3) a tripartition of V. We define the improvement of $\mathcal{T}$ with (uv, C_1, C_2, C_3) as the following branch decomposition.*

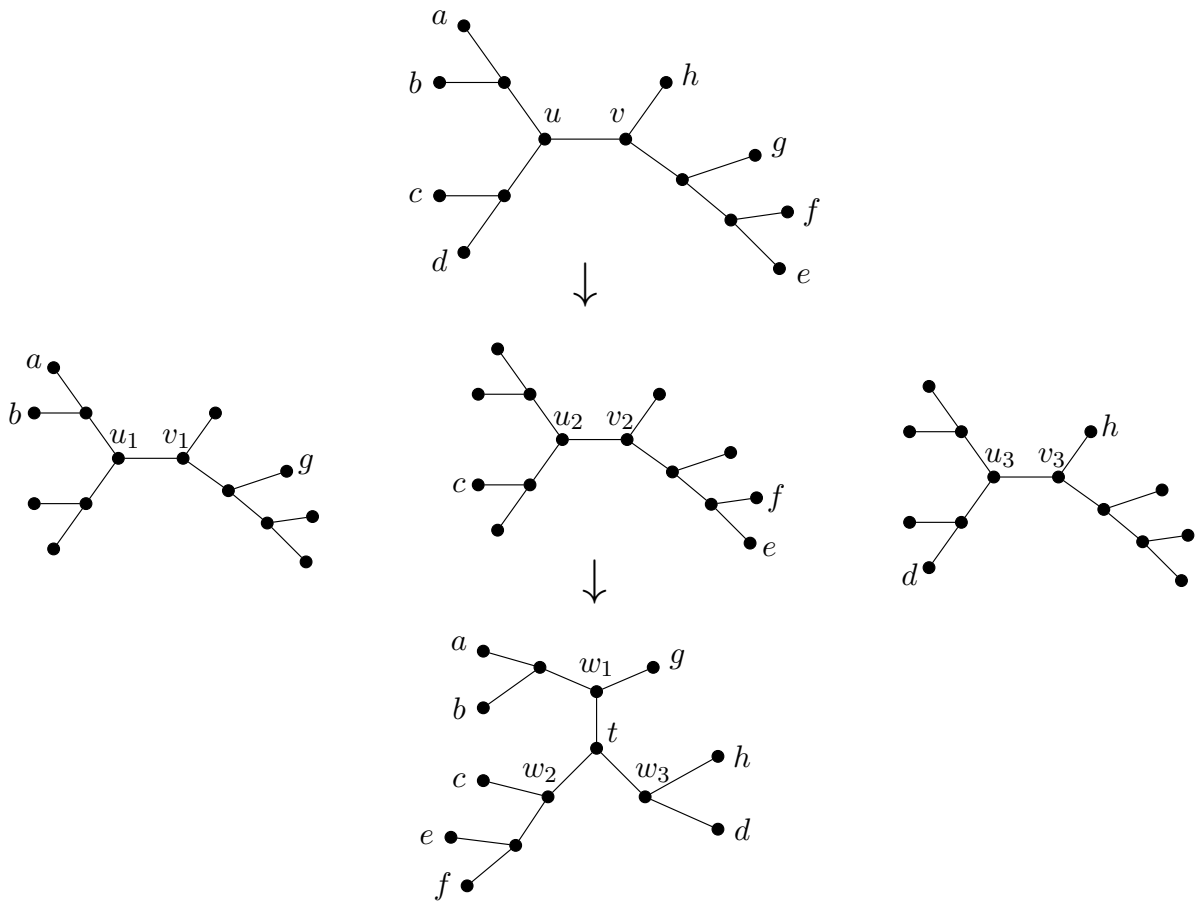

Figure 7.1: An example of the improvement operation. A branch decomposition (T, λ) of a function $f\colon 2^V \to \mathbb{Z}_{\geq 0}$ with $V = \{a, b, c, d, e, f, g, h\}$ (top). For a tripartition $(C_1 = \{a, b, g\}, C_2 = \{c, e, f\}, C_3 = \{d, h\})$, we have the partial branch decompositions $(T_1, \lambda_1) = (T, \lambda{\restriction_{\{a,b,g\}}})$, $(T_2, \lambda_2) = (T, \lambda{\restriction_{\{c,e,f\}}})$, and $(T_3, \lambda_3) = (T, \lambda{\restriction_{\{d,h\}}})$ (middle), and the improvement of (T, λ) with (uv, C_1, C_2, C_3) (bottom).

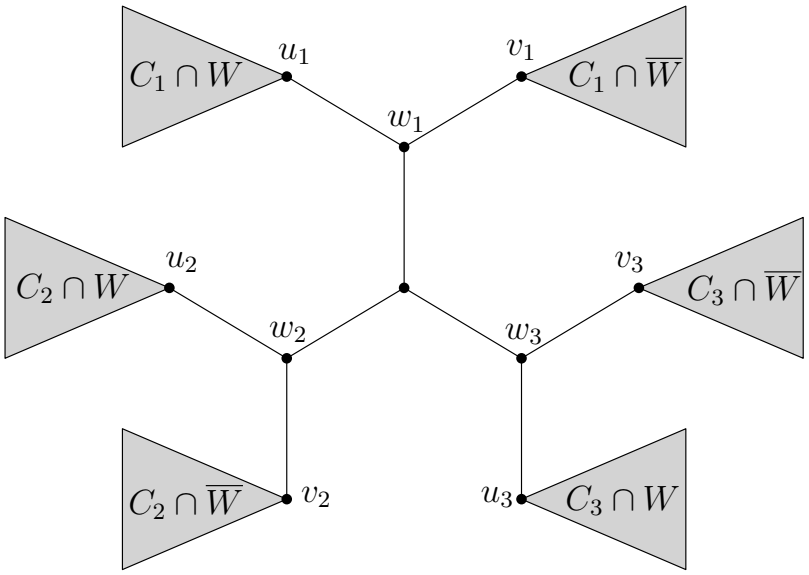

Figure 7.2: Changes of the width "around" uv.

For each $i \in [3]$, let $\mathcal{T}_i = (T_i, \lambda_i) = (T, \lambda\restriction_{C_i})$, and let u_iv_i be the copy of the edge uv in T_i. Now, let $\mathcal{T}' = (T', \lambda')$ be a partial branch decomposition on V obtained by first inserting a node w_i on the edge u_iv_i of each T_i (i.e. $V(T_i) \leftarrow V(T_i) \cup \{w_i\}$ and $E(T_i) \leftarrow E(T_i) \cup \{u_iw_i, w_iv_i\} \setminus \{u_iv_i\}$), then taking the disjoint union of $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, and then inserting a node t adjacent to w_1, w_2, w_3, connecting the disjoint union into a tree. Finally, the improvement is obtained by simplifying $\mathcal{T}'$ by iteratively removing leaves that are not labeled by λ', and suppressing degree-2 nodes.

We observe that if $\mathcal{T}'$ is an improvement of $\mathcal{T}$ with (r, C_1, C_2, C_3), then every edge of $\mathcal{T}'$ either corresponds to a bipartition $(C_i, \overline{C_i})$ of V or to a bipartition of V of form $(\mathcal{T}_r[w] \cap C_i, \overline{\mathcal{T}_r[w] \cap C_i})$, where $w \in V(\mathcal{T})$ (see also Figure 7.2). More formally as follows.

Lemma 7.3. *Let $\mathcal{T}$ be a branch decomposition, $r \in E(\mathcal{T})$, and (C_1, C_2, C_3) a tripartition of V. Let $\mathcal{T}'$ be the improvement of $\mathcal{T}$ with (r, C_1, C_2, C_3). It holds that*

$$\{\{\mathcal{T}'[u'v'], \mathcal{T}'[v'u']\} \mid u'v' \in E(\mathcal{T}')\} =$$
$$\bigcup_{i\in[3]} \Big(\{\{C_i, \overline{C_i}\}\} \cup \{\{\mathcal{T}_r[w] \cap C_i, \overline{\mathcal{T}_r[w] \cap C_i}\} \mid w \in V(\mathcal{T})\}\Big) \setminus \{\{\emptyset, V\}\}.$$

Note that as a branch decomposition is a cubic tree, no two edges correspond to the same bipartition, i.e., the set $\{\{\mathcal{T}[uv], \mathcal{T}[vu]\} \mid uv \in E(\mathcal{T})\}$ has size $|E(\mathcal{T})|$.

7.2.2 Improving with splits

Now we state the definitions and results of our combinatorial framework, postponing the proofs to Subsections 7.2.3 and 7.2.4. First we define a *split* of a set $W \subseteq V$.

Definition 7.4 (Split)**.** *Let $W \subseteq V$. A tripartition (C_1, C_2, C_3) of V is a split of W if for every $i \in [3]$, $f(C_i) < f(W)$, $f(C_i \cap W) < f(W)$, and $f(C_i \cap \overline{W}) < f(W)$.*

We note that because of the condition $f(C_i \cap W) < f(W)$, it holds that $C_i \neq V$ for every $i \in [3]$. However, it can be that $C_i = \emptyset$. We also observe that splits are symmetric in the sense that re-ordering the sets C_1, C_2, C_3 or interchanging W with $\overline{W}$ does not change whether (C_1, C_2, C_3) is a split.

The first main property of splits, analogous to Lemma 4.1, is that for every set W with large enough $f(W)$ there exists a split.

Lemma 7.5. *If $W \subseteq V$ such that $f(W) > 2\mathsf{bw}(f)$, then there exists a split of W.*

We postpone the proof of Lemma 7.5 to Subsection 7.2.3.

If uv is an edge of a branch decomposition $\mathcal{T}$ with $(\mathcal{T}[uv], \mathcal{T}[vu]) = (W, \overline{W})$, then improving with (uv, C_1, C_2, C_3) where (C_1, C_2, C_3) is a split of W "locally improves" the branch decomposition around the edge uv in the sense that the widths of the new edges corresponding to the edge uv will be of form $f(C_i) < f(W)$, $f(C_i \cap W) < f(W)$, and $f(C_i \cap \overline{W}) < f(W)$ (recall Figure 7.2). We then define minimum splits analogously to Chapter 4, in order to argue that the improvement operation globally improves the branch decomposition if we use a minimum split.

For this, we start by defining a split of a branch decomposition.

Definition 7.6 (Split of $\mathcal{T}$)**.** *Let $\mathcal{T}$ be a branch decomposition and $r = uv \in E(\mathcal{T})$ an edge of $\mathcal{T}$. A split of $\mathcal{T}$ on r is a tuple (r, C_1, C_2, C_3), where $W = \mathcal{T}[uv]$, and (C_1, C_2, C_3) is a split of W.*

Then, we say that C_i *r-cuts* a node w of $\mathcal{T}$ if both C_i and $\overline{C_i}$ intersect the set $\mathcal{T}_r[w]$. We also define that a split (r, C_1, C_2, C_3) of $\mathcal{T}$ *cuts* a node w if at least one of the sets C_1,C_2, and C_3 r-cuts w. Note that if one of the sets r-cuts w, then at least two of the sets r-cuts w. We also define that the *sum-width* of a split (r, C_1, C_2, C_3) is $f(C_1) + f(C_2) + f(C_3)$. Now we are ready to define a minimum split of $\mathcal{T}$.

Definition 7.7 (Minimum split of $\mathcal{T}$)**.** *A split (r, C_1, C_2, C_3) of $\mathcal{T}$ on $r = uv \in E(\mathcal{T})$ is a minimum split of $\mathcal{T}$ if it*

1. *has minimum sum-width among all splits of $\mathcal{T}$ on r, and*
2. *subject to (1), cuts the minimum number of nodes of $\mathcal{T}$.*

The following theorem is our main combinatorial result. Analogously to Lemma 4.4, it states that the improvement operation with a minimum split of $\mathcal{T}$ improves $\mathcal{T}$ on all edges except those that are not affected by the improvement operation.

Theorem 7.8. *Let $\mathcal{T}$ be a branch decomposition, $r = uv \in E(\mathcal{T})$ an edge of $\mathcal{T}$, and (r, C_1, C_2, C_3) a minimum split of $\mathcal{T}$. Then for every $i \in [3]$ and every node $w \in V(\mathcal{T})$, it holds that $f(\mathcal{T}_r[w] \cap C_i) \leq f(\mathcal{T}_r[w])$. Moreover, the equality holds only if $\mathcal{T}_r[w] \cap C_i = \mathcal{T}_r[w]$ or $\mathcal{T}_r[w] \cap C_i = \emptyset$.*

We postpone the proof of Theorem 7.8 to Subsection 7.2.4.

7.2.3 Existence of a split

Now we prove that if $f(W) > 2\mathsf{bw}(f)$, then there exists a split of W, that is, we prove Lemma 7.5. The idea of the proof is that we take an optimum-width branch decomposition $\mathcal{T}^\star$ of f, i.e., a branch decomposition $\mathcal{T}^\star$ with $\mathsf{width}(\mathcal{T}^\star) = \mathsf{bw}(f) < f(W)/2$, and argue that either a bipartition of V corresponding to some edge of $\mathcal{T}^\star$ or a tripartition of V corresponding to some node of $\mathcal{T}^\star$ results in a split of W.

We define an orientation of a set $C \subseteq V$ with respect to a set $W \subseteq V$. This will be used for orienting the edges of the optimal branch decomposition $\mathcal{T}^\star$ based on W.

Definition 7.9 (Orientation)**.** *Let $C, W \subseteq V$. We say that*

- *the set W directly orients C if $f(C \cap W) < f(\overline{C} \cap W)$ and $f(C \cap \overline{W}) < f(\overline{C} \cap \overline{W})$,*
- *the set W inversely orients C if it directly orients $\overline{C}$, and*
- *the set W disorients C if it neither directly nor inversely orients C.*

Note that the definition of orienting is symmetric with respect to complementing W, i.e., W (directly, inversely, dis)-orients C if and only if $\overline{W}$ (directly, inversely, dis)-orients C. Next we show that if there is an edge of an optimal branch decomposition that is disoriented according to Definition 7.9, then it corresponds to a split of W.

Lemma 7.10. *Let $C, W \subseteq V$. If W disorients C and $f(C) < f(W)/2$, then $(C, \overline{C}, \emptyset)$ is a split of W.*

Proof. By possibly interchanging C with $\overline{C}$, without loss of generality we can assume that $f(C \cap W) \leq f(\overline{C} \cap W)$ and $f(C \cap \overline{W}) \geq f(\overline{C} \cap \overline{W})$. By these inequalities and since $f(C) = f(\overline{C}) < f(W)/2$, to show that $(C, \overline{C}, \emptyset)$ is a split of W, it suffices to prove that $f(\overline{C} \cap W) < f(W)$ and $f(C \cap \overline{W}) < f(W)$.

First, we have

$$\begin{aligned} f(\overline{C} \cap W) &\leq f(\overline{C}) + f(W) - f(\overline{C} \cup W) && \text{(submodularity)} \\ &\leq f(C) + f(W) - f(C \cap \overline{W}) && \text{(symmetry).} \end{aligned} \tag{7.11}$$

Then, by submodularity

$$f(C \cap \overline{W}) + f(\overline{C} \cap \overline{W}) \geq f(\overline{W}) + f(\emptyset) > 2f(C),$$

implying $f(C \cap \overline{W}) > f(C)$, implying by (7.11) that $f(\overline{C} \cap W) < f(W)$. The proof of $f(C \cap \overline{W}) < f(W)$ is the same with the roles of C and $\overline{C}$, and of W and $\overline{W}$, interchanged. □

Now, to prove Lemma 7.5, we take an optimum-width branch decomposition $\mathcal{T}^\star$ of f and orient each edge uv with $(\mathcal{T}^\star[uv], \mathcal{T}^\star[vu]) = (C, \overline{C})$ towards v if W directly orients C and towards u if W inversely orients C. If no such orientation exists, i.e., W disorients C, then Lemma 7.10 shows that $(C, \overline{C}, \emptyset)$ is a split of W and we are done, so for the remainder of the proof assume that every edge is indeed oriented.

Lemma 7.12. *No edge of $\mathcal{T}^\star$ is oriented towards a leaf.*

Proof. Suppose there is an edge of $\mathcal{T}^\star = (T^\star, \lambda^\star)$ oriented towards a leaf $\ell \in V(\mathcal{T}^\star)$, and let $v = \lambda^{-1}(\ell)$. Thus W directly orients $V \setminus \{v\}$, implying $f(W \setminus \{v\}) < f(W \cap \{v\})$ and $f(\overline{W} \setminus \{v\}) < f(\overline{W} \cap \{v\})$. However, one of the sets $W \cap \{v\}$ or $\overline{W} \cap \{v\}$ must be empty and therefore either $f(W \cap \{v\}) = 0$ or $f(\overline{W} \cap \{v\}) = 0$, implying $f(W \setminus \{v\}) < 0$ or $f(\overline{W} \setminus \{v\}) < 0$, but f cannot take values less than 0, so we get a contradiction. □

Now, by walking in $\mathcal{T}^\star$ according to the orientation, we end up finding an internal node towards which all incident edges are oriented. The following lemma shows that this node indeed gives a split of W, and therefore completes the proof of Lemma 7.5.

Lemma 7.13. *Let $W \subseteq V$, and (C_1, C_2, C_3) a tripartition of V so that for each $i \in [3]$, $f(C_i) < f(W)/2$ and W directly orients C_i. Then (C_1, C_2, C_3) is a split of W.*

Proof. Since W directly orients C_i, we have $f(C_i \cap W) < f(\overline{C}_i \cap W)$. Therefore,

$$\begin{aligned}
2f(C_i \cap W) &< f(C_i \cap W) + f(\overline{C_i} \cap W) \\
&< 2f(C_i) + 2f(W) - f(C_i \cup W) - f(\overline{C_i} \cup W) && \text{(submodularity)} \\
&< 2f(C_i) + 2f(W) - f(V) - f(W) && \text{(submodularity)} \\
&< 2f(C_i) + f(W) < 2f(W)
\end{aligned}$$

Therefore, $f(C_i \cap W) < f(W)$. The proof that $f(C_i \cap \overline{W}) < f(W)$ is the same with the roles of W and $\overline{W}$ interchanged. □

7.2.4 Improving globally

Then, we show that by selecting the split as a minimum split of $\mathcal{T}$, the improvement operation by using it indeed improves $\mathcal{T}$ globally. In particular, we prove Theorem 7.8.

The key intermediate definition will be *linked splits*. Let $A \subseteq B \subseteq V$. The set A is *linked into* B if for all sets S with $A \subseteq S \subseteq B$ it holds that $f(A) \leq f(S)$. Then, let $\mathcal{T}$ be a branch decomposition, $r = uv \in E(\mathcal{T})$ an edge of $\mathcal{T}$, and $W = \mathcal{T}[uv]$. A split (r, C_1, C_2, C_3) of $\mathcal{T}$ is *linked* if for every $i \in [3]$ it holds that

1. $W \cap C_i$ is linked into W,
2. if C_i r-cuts an r-descendant w of u, then $f((W \cap C_i) \cup \mathcal{T}_r[w]) > f(W \cap C_i)$, and
3. the same conditions hold when W is interchanged with $\overline{W}$ and u is interchanged with v.

Then we prove the main lemma of this subsection.

Lemma 7.14. *If (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$, then (r, C_1, C_2, C_3) is linked.*

Proof. Suppose that (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$ that is not linked. Without loss of generality, assume that it violates either Item 1 or Item 2 in the definition of linkedness (note that by symmetry we can interchange the roles of W and $\overline{W}$, and of u and v). In particular, there exists $i \in [3]$ and a set S with $W \cap C_i \subseteq S \subseteq W$ so that either

1. $f(S) < f(W \cap C_i)$, or
2. $f(S) = f(W \cap C_i)$ and $S = (W \cap C_i) \cup \mathcal{T}_r[w]$ for some r-descendant w of u so that C_i r-cuts w.

We moreover fix such set S that minimizes $f(S)$. We will use the set S to construct a new split that contradicts the minimality of (r, C_1, C_2, C_3). To simplify notation, assume without loss of generality that $i = 1$.

Now, we let the new split be

$$(r, C_1', C_2', C_3') = (r, C_1 \cup S, C_2 \setminus S, C_3 \setminus S).$$

We first bound the values $f(C_j' \cap W)$ and $f(C_j' \cap \overline{W})$ for all $j \in [3]$. Because $S \subseteq W$, we have $C_j' \cap \overline{W} = C_j \cap \overline{W}$ for all $j \in [3]$, implying $f(C_j' \cap \overline{W}) < f(W)$. Also, by definition, $f(C_1' \cap W) = f(S) \leq f(C_1 \cap W) < f(W)$.

Claim 7.15. *For $j \in \{2,3\}$, it holds that $f(C'_j \cap W) \leq f(C_j \cap W)$.*

Proof of the claim. Let $D = C_j \cap W$. We observe that

$$f(S) \leq f(\overline{D} \cap S) \tag{7.16}$$

because $W \cap C_1 \subseteq \overline{D} \cap S \subseteq W$, but S minimizes $f(S)$ among such sets. Then,

$$\begin{aligned} f(C'_j \cap W) = f(D \cap \overline{S}) &= f(\overline{D} \cup S) && \text{(symmetry)} \\ &\leq f(\overline{D}) + f(S) - f(\overline{D} \cap S) && \text{(submodularity)} \\ &\leq f(\overline{D}) = f(D) && \text{((7.16) and symmetry)} \end{aligned}$$

◁

Then we bound the values $f(C'_j)$ for $j \in [3]$.

Claim 7.17. *For $j \in [3]$, it holds that $f(C'_j) \leq f(C_j)$.*

Proof of the claim. If $j = 1$, let $D = C_j$, and if $j \in \{2,3\}$, let $D = \overline{C_j}$. Now, $C_1 \subseteq D$, and our goal is to prove that $f(D \cup S) \leq f(D)$. We observe that

$$f(S) \leq f(D \cap S) \tag{7.18}$$

because $W \cap C_1 \subseteq D \cap S \subseteq W$, but S minimizes $f(S)$ among such sets. Then,

$$\begin{aligned} f(D \cup S) &\leq f(D) + f(S) - f(D \cap S) && \text{(submodularity)} \\ &\leq f(D). && \text{by (7.18)} \end{aligned}$$

◁

We have now proven that (C'_1, C'_2, C'_3) is indeed a split of W. We then wish to show that (r, C'_1, C'_2, C'_3) contradicts that (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$.

First, suppose that the case of Item 1 holds, i.e., $f(S) < f(W \cap C_1)$. Now,

$$\begin{aligned} f(C'_1) = f(C_1 \cup S) &\leq f(C_1) + f(S) - f(C_1 \cap S) && \text{(submodularity)} \\ &\leq f(C_1) + f(S) - f(C_1 \cap W) && \text{(by } C_1 \cap S = C_1 \cap W\text{)} \\ &< f(C_1). && \text{(by } f(S) < f(W \cap C_1)\text{)} \end{aligned}$$

By combining with Claim 7.17, we get that (C'_1, C'_2, C'_3) has smaller sum-width than (C_1, C_2, C_3), and therefore contradicts that (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$.

Suppose that the case of Item 2 holds, i.e., $f(S) = f(W \cap C_1)$ and $S = (W \cap C_1) \cup \mathcal{T}_r[w]$ for some r-descendant w of u so that C_1 r-cuts w. By Claim 7.17 (C'_1, C'_2, C'_3) has the same sum-width as (C_1, C_2, C_3). Let us analyze which nodes of $\mathcal{T}$ are cut by (r, C'_1, C'_2, C'_3).

Claim 7.19. *If (r, C'_1, C'_2, C'_3) cuts a node $x \in V(\mathcal{T})$, then also (r, C_1, C_2, C_3) cuts x.*

Proof of the claim. If x is an r-descendant of w, then $\mathcal{T}_r[x] \subseteq S$, so (r, C'_1, C'_2, C'_3) does not cut x. If $\mathcal{T}_r[x]$ is disjoint from $\mathcal{T}_r[w]$, then $C'_j \cap \mathcal{T}_r[x] = C_j \cap \mathcal{T}_r[x]$ for all $j \in [3]$, so (r, C'_1, C'_2, C'_3) cuts x if and only if (r, C_1, C_2, C_3) cuts x. If x is an r-ancestor of w, then (r, C_1, C_2, C_3) cuts x because it cuts w. ◁

Because (r, C_1, C_2, C_3) cuts w but (r, C'_1, C'_2, C'_3) does not cut w, (r, C'_1, C'_2, C'_3) cuts less nodes of $\mathcal{T}$ than (r, C_1, C_2, C_3), and therefore contradicts that (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$. □

Then we prove Theorem 7.8 by using Lemma 7.14.

Theorem 7.8. *Let $\mathcal{T}$ be a branch decomposition, $r = uv \in E(\mathcal{T})$ an edge of $\mathcal{T}$, and (r, C_1, C_2, C_3) a minimum split of $\mathcal{T}$. Then for every $i \in [3]$ and every node $w \in V(\mathcal{T})$, it holds that $f(\mathcal{T}_r[w] \cap C_i) \leq f(\mathcal{T}_r[w])$. Moreover, the equality holds only if $\mathcal{T}_r[w] \cap C_i = \mathcal{T}_r[w]$ or $\mathcal{T}_r[w] \cap C_i = \emptyset$.*

Proof. If C_i does not r-cut w, then $\mathcal{T}_r[w] \cap C_i \in \{\mathcal{T}_r[w], \emptyset\}$. In both of these cases, it is straightforward to verify that the conclusion holds.

Then, assume that C_i r-cuts w, and suppose without loss of generality that w is an r-descendant of u. Let $D_i = W \cap C_i$. By Lemma 7.14 (r, C_1, C_2, C_3) is linked, and therefore by Item 2 of the definition it holds that $f(D_i \cup \mathcal{T}_r[w]) > f(D_i)$. By submodularity, $f(D_i \cap \mathcal{T}_r[w]) < f(\mathcal{T}_r[w])$, which concludes the proof as $D_i \cap \mathcal{T}_r[w] = C_i \cap \mathcal{T}_r[w]$. □

7.3 Algorithmic framework

In this section we present our algorithmic framework for designing FPT 2-approximation algorithms for computing branch decompositions. In particular, we show that a sequence of improvement operations decreasing the width of a branch decomposition from k to $k-1$ or concluding $k \leq 2\mathsf{bw}(f)$ can be implemented in time $t(k) \cdot 2^{\mathcal{O}(k)} \cdot n$ for connectivity functions f whose branch decompositions support certain type of dynamic programming with running time $t(k)$ per node. The concrete implementation of this framework for

rankwidth, with $t(k) = 2^{2^{\mathcal{O}(k)}}$, is provided in Section 7.4 and for graph branchwidth, with $t(k) = 2^{\mathcal{O}(k)}$, in Section 7.5.

7.3.1 Amortized analysis

A naive implementation of the improvement operation (as described in Definition 7.2) would use $\Omega(n)$ time on each improvement, which would result in the running time of $\Omega(n^2)$ over the course of n improvements. In this section we show that the improvement operations can be implemented so that over any sequence of improvement operations using minimum splits of a branch decomposition of width at most k, the total work done in improving the branch decomposition amortizes to $2^{\mathcal{O}(k)}n$.

The efficient implementation of improvements is based on the notion of the edit set of a minimum split of $\mathcal{T}$. Informally, the edit set of a minimum split (r, C_1, C_2, C_3) of $\mathcal{T}$ consists of the nodes of the subtree of $\mathcal{T}$ obtained after pruning all r-subtrees whose leaves are entirely from one of the sets C_i. Formally, as follows.

Definition 7.20 (Edit set)**.** *Let $\mathcal{T}$ be a branch decomposition, $r \in E(\mathcal{T})$, and (r, C_1, C_2, C_3) a minimum split of $\mathcal{T}$. The edit set of (r, C_1, C_2, C_3) is the set $R \subseteq V(\mathcal{T})$ of nodes of $\mathcal{T}$ that are cut by (r, C_1, C_2, C_3), i.e.,*

$$R = \{w \in V(\mathcal{T}) \mid \mathcal{T}_r[w] \text{ intersects at least two sets from } \{C_1, C_2, C_3\}\}.$$

Note that for a minimum split (uv, C_1, C_2, C_3) of $\mathcal{T}$, both u and v are necessarily in the edit set. We formalize the intuition about edit sets in the following lemma. It will be implicitly used in many of our arguments.

Lemma 7.21. *Let $\mathcal{T} = (T, \lambda)$ be a branch decomposition, $r = uv \in E(\mathcal{T})$, (r, C_1, C_2, C_3) a minimum split of $\mathcal{T}$, R the edit set of (r, C_1, C_2, C_3), and $\mathcal{T}'$ the improvement of $\mathcal{T}$ with (r, C_1, C_2, C_3). It holds that*

(1) every node in R is non-leaf,

(2) the nodes of R induce a connected subtree $T[R]$ of T, and

(3) there exists an edge $r' \in E(\mathcal{T}')$ so that for every $w \in V(T) \setminus R$ there is a node $w' \in V(\mathcal{T}')$ with $\mathcal{T}_r[w] = \mathcal{T}'_{r'}[w']$.

Proof. For (1), the set $\mathcal{T}_r[w]$ of a leaf w consists of one element. Thus it is not cut by (r, C_1, C_2, C_3). For (2), first note that $\{u, v\} \subseteq R$. Then, consider a node $w \in R \setminus \{u, v\}$, and let p be the r-parent of w. It holds that $\mathcal{T}_r[w] \subseteq \mathcal{T}_r[p]$, so p must also be in R.

For (3), observe that by the definition of edit set for every node $w \in V(T) \setminus R$ it holds that $\mathcal{T}_r[w] \subseteq C_i$ for some i. This implies that the r-subtree of w appears identically in $\mathcal{T}'$, and therefore $\mathcal{T}'$ consists of the r-subtrees of all $w \in N_T(R)$ and a connected subtree inserted in the place of R and connected to $N_T(R)$. As $|R| \geq 2$, this inserted subtree contains at least one edge, which we can designate as r'. □

Next we define the *neighborhood partition* of an edit set R.

Definition 7.22 (Neighborhood partition)**.** *Let (r, C_1, C_2, C_3) be a minimum split and R its edit set. The neighborhood partition of R is the partition (N_1, N_2, N_3) of the neighbors $N_T(R)$ of R, where $N_i = \{w \in N_T(R) \mid \mathcal{T}_r[w] \subseteq C_i\}$.*

Note that the neighborhood partition is indeed a partition of $N_T(R)$ by the definition of edit set.

Next we give an algorithm for performing the improvement operation in $\mathcal{O}(|R|)$ time, given the edit set R and its neighborhood partition.

Lemma 7.23. *Let $\mathcal{T} = (T, \lambda)$ be a branch decomposition, $r = uv \in E(\mathcal{T})$, and (r, C_1, C_2, C_3) a minimum split of $\mathcal{T}$. Given the edit set R of (r, C_1, C_2, C_3) and the neighborhood partition (N_1, N_2, N_3) of R, $\mathcal{T}$ can be turned into the improvement of $\mathcal{T}$ with (r, C_1, C_2, C_3) in $\mathcal{O}(|R|)$ time. Moreover, all information stored in nodes $V(\mathcal{T}) \setminus R$ is preserved and the other nodes are marked as new.*

Proof. We create three copies T_1, T_2, T_3 of the induced subtree $T[R]$. We denote the copy of a node $x \in R$ in T_i by x_i, and denote $R_i = \{x_i \mid x \in R\}$. To each T_i we also insert a new node w_i on the edge $u_i v_i$, i.e., let $V(T_i) \leftarrow V(T_i) \cup \{w_i\}$ and $E(T_i) \leftarrow E(T_i) \setminus \{u_i, v_i\} \cup \{u_i w_i, w_i v_i\}$. We then insert a new center node t and connect each w_i to it.

For each node $y \in N_i$, let $p \in R$ be the r-parent of y. We remove the edge yp and insert the edge yp_i. It remains to remove all nodes of R, and then iteratively remove degree-1 nodes and suppress degree-2 nodes in $R_1 \cup R_2 \cup R_3 \cup \{t, w_1, w_2, w_3\}$.

For the running time, these operations can be done in time linear in $|R| + |R_1| + |R_2| + |R_3| + |N_1| + |N_2| + |N_3| = \mathcal{O}(|R|)$ because T is represented as an adjacency list and the maximum degree of T is 3. □

The outline of the improvement operation in our framework is that the dynamic programming outputs the edit set R and its neighborhood partition in $\mathcal{O}(t(k) \cdot |R|)$ time, then the algorithm of Lemma 7.23 computes the improvement in $\mathcal{O}(|R|)$ time, and then the

dynamic programming tables of the $|R|$ new nodes are computed in $\mathcal{O}(t(k) \cdot |R|)$ time. To bound the sum of the sizes of the edit sets R over the course of the algorithm, we introduce the following potential function.

Definition 7.24 (k-potential)**.** *Let $\mathcal{T}$ be a branch decomposition of a connectivity function f. The k-potential of $\mathcal{T}$ is*

$$\Phi_k(\mathcal{T}) = \sum_{\substack{e \in E(\mathcal{T}) \\ f(e) < k}} f(e) \cdot 3^{f(e)} + \sum_{\substack{e \in E(\mathcal{T}) \\ f(e) \geq k}} 4 \cdot f(e) \cdot 3^{f(e)}.$$

When working with k-potentials, we will use the following notation. For $x \geq 0$, let

$$\Phi_k(x) = \begin{cases} x \cdot 3^x, & \text{if } x < k \\ 4 \cdot x \cdot 3^x, & \text{otherwise,} \end{cases}$$

For $W \subseteq V$, we will use $\Phi_k(W)$ to denote $\Phi_k(f(W))$. With this notation, the k-potential of $\mathcal{T}$ is

$$\Phi_k(\mathcal{T}) = \sum_{uv \in E(\mathcal{T})} \Phi_k(\mathcal{T}[uv]).$$

The k-potential of a branch decomposition $\mathcal{T}$ is at most $\mathcal{O}(3^{\mathsf{width}(\mathcal{T})} \cdot \mathsf{width}(\mathcal{T}) \cdot |\mathcal{T}|)$, which is $2^{\mathcal{O}(k)} \cdot |\mathcal{T}|$ when $\mathsf{width}(\mathcal{T}) = \mathcal{O}(k)$.

Next we show that performing a improvement operation with an edit set R decreases the k-potential by at least $|R|$.

Lemma 7.25. *Let $\mathcal{T}$ be a branch decomposition with an edge $r = uv \in E(\mathcal{T})$ so that $f(uv) = \mathsf{width}(\mathcal{T}) = k$. Let (r, C_1, C_2, C_3) be a minimum split of $\mathcal{T}$, R the edit set of (r, C_1, C_2, C_3), and $\mathcal{T}'$ be the improvement of $\mathcal{T}$ with (r, C_1, C_2, C_3). Then it holds that $\Phi_k(\mathcal{T}') \leq \Phi_k(\mathcal{T}) - |R|$.*

Proof. We use the notation that $W = \mathcal{T}[uv]$. Note that

$$\Phi_k(\mathcal{T}) = \Phi_k(k) + \sum_{w \in (V(\mathcal{T}) \setminus \{u,v\})} \Phi_k(\mathcal{T}_r[w]) \tag{7.26}$$

and that by Lemma 7.3

$$\Phi_k(\mathcal{T}') = \sum_{i \in [3]} \left(\Phi_k(C_i) + \sum_{w \in V(\mathcal{T})} \Phi_k(C_i \cap \mathcal{T}_r[w]) \right). \tag{7.27}$$

Then

$$\begin{aligned}\Phi_k(\mathcal{T}) - \Phi_k(\mathcal{T}') &= \Phi_k(\mathcal{T}) - \sum_{i\in[3]} \left(\Phi_k(C_i) \qquad + \sum_{w\in V(\mathcal{T})} \Phi_k(C_i \cap \mathcal{T}_r[w]) \right) \\ &\geq \Phi_k(\mathcal{T}) - \sum_{i\in[3]} \Bigg(\Phi_k(C_i) \qquad + \Phi_k(C_i \cap W) + \Phi_k(C_i \cap \overline{W}) \\ &\qquad\qquad + \sum_{w\in(V(\mathcal{T})\setminus\{u,v\})} \Phi_k(C_i \cap \mathcal{T}_r[w]) \Bigg),\end{aligned}$$

where the last inequality is obtained by taking $C_i \cap \mathcal{T}_r[u] = C_i \cap W$ and $C_i \cap \mathcal{T}_r[v] = C_i \cap \overline{W}$ out of the sum.

By the definition of a split of W, we have that $\Phi_k(C_i) \leq \Phi_k(k-1)$ and $\Phi_k(C_i \cap W) \leq \Phi_k(k-1)$. Then by interleaving the sums (7.26) and (7.27), we have that $\Phi_k(\mathcal{T}) - \Phi_k(\mathcal{T}')$ is at least

$$\Phi_k(k) - 9\Phi_k(k-1) + \sum_{w\in(V(\mathcal{T})\setminus\{u,v\})} \left(\Phi_k(\mathcal{T}_r[w]) - \sum_{i\in[3]} \Phi_k(C_i \cap \mathcal{T}_r[w]) \right).$$

By observing $\Phi_k(k) - 9 \cdot \Phi_k(k-1) \geq 2$, we lower bound $\Phi_k(\mathcal{T}) - \Phi_k(\mathcal{T}')$ by

$$2 + \sum_{w\in(V(\mathcal{T})\setminus\{u,v\})} \left(\Phi_k(\mathcal{T}_r[w]) - \sum_{i\in[3]} \Phi_k(C_i \cap \mathcal{T}_r[w]) \right).$$

Let us note that for $w \notin R$, $\Phi_k(\mathcal{T}_r[w]) = \sum_{i\in[3]} \Phi_k(C_i \cap \mathcal{T}_r[w])$ because $\mathcal{T}_r[w]$ is a subset of some C_i and $\Phi_k(\emptyset) = 0$. Also by Theorem 7.8, for any node w in the edit set and every i it holds that $f(C_i \cap \mathcal{T}_r[w]) < f(\mathcal{T}_r[w])$, implying $\Phi_k(\mathcal{T}_r[w]) - \sum_{i\in[3]} \Phi_k(C_i \cap \mathcal{T}_r[w]) \geq 1$. By making use of these observations, we conclude that

$$\Phi_k(\mathcal{T}) - \Phi_k(\mathcal{T}') \geq \qquad 2 + \sum_{w\in(R\setminus\{u,v\})} \left(\Phi_k(\mathcal{T}_r[w]) - \sum_{i\in[3]} \Phi_k(C_i \cap \mathcal{T}_r[w]) \right) \geq |R|.$$

□

In particular, when performing a sequence of improvement operations with minimum splits of $\mathcal{T}$ on edges r of width $f(r) = \mathsf{width}(\mathcal{T}) = k$, the sum of the sizes of edit sets across all of the operations is at most $\mathcal{O}(3^k \cdot k \cdot |\mathcal{T}|)$.

7.3.2 Improvement data structure

We define the *improvement data structure* to formally capture what is required from the underlying dynamic programming in our framework.

Definition 7.28 (Improvement data structure)**.** *Let f be a connectivity function and k an integer. An improvement data structure of f with running time $t(k)$ maintains a branch decomposition $\mathcal{T}$ of f with $\mathsf{width}(\mathcal{T}) \leq k$ rooted on an edge $r = uv \in E(\mathcal{T})$ and supports the following operations.*

- $\mathsf{Init}(\mathcal{T}, uv)$*: Given $k \in \mathbb{Z}_{\geq 0}$, a branch decomposition $\mathcal{T}$ of f with $\mathsf{width}(\mathcal{T}) \leq k$, and an edge $uv \in E(\mathcal{T})$, initialize the data structure in $\mathcal{O}(t(k) \cdot |\mathcal{T}|)$ time.*
- $\mathsf{Move}(vw)$*: Move the root edge $r = uv$ to an incident edge vw, i.e., set $r \leftarrow vw$. Runs in $\mathcal{O}(t(k))$ time.*
- $\mathsf{Width}()$*: Return $f(uv)$ in $\mathcal{O}(t(k))$ time.*
- $\mathsf{CanImprove}()$*: Returns true if there exists a split of $W = \mathcal{T}[uv]$ and false otherwise. Runs in time $\mathcal{O}(t(k))$. Once* $\mathsf{CanImprove}()$ *has returned true, the following can be invoked:*
  - $\mathsf{EditSet}()$*: Let (r, C_1, C_2, C_3) be a minimum split of $\mathcal{T}$, R the edit set of (r, C_1, C_2, C_3), and (N_1, N_2, N_3) the neighborhood partition of R. Returns R and (N_1, N_2, N_3). Runs in $\mathcal{O}(t(k) \cdot |R|)$ time.*
  - $\mathsf{Improve}(R, (N_1, N_2, N_3))$*: Implements the improvement operation described in Lemma 7.23. That is, computes the improvement of $\mathcal{T}$ by removing the edit set R and inserting a connected subtree of $|R|$ nodes in its place. Sets r to an arbitrary edge between two newly inserted nodes (such an edge exists because $|R| \geq 2$). Runs in $\mathcal{O}(t(k) \cdot |R|)$ time.*
- $\mathsf{Output}()$*: Outputs $\mathcal{T}$ in $\mathcal{O}(t(k) \cdot |\mathcal{T}|)$ time.*

We explain how our algorithm uses the improvement data structure in Subsection 7.3.3. Let us here informally explain how the improvement data structure is typically implemented using dynamic programming. The formal descriptions for rankwidth and branchwidth of graphs constitute Sections 7.4 and 7.5.

For each node w of $\mathcal{T}$, the improvement data structure stores a dynamic programming table of size $\mathcal{O}(t(k))$ that represents information of the r-subtree of w in such a way that the dynamic programming tables of the nodes u and v combined together can be used to

detect the existence of a split of $W = \mathcal{T}[uv]$. Now, the $\mathsf{Init}(\mathcal{T}, uv)$ operation is to compute these dynamic programming tables in a bottom-up fashion for all nodes from the leaves towards the root uv, using $\mathcal{O}(t(k))$ time per node. The $\mathsf{Move}(vw)$ operation changes the root edge uv to an incident edge vw. For implementing $\mathsf{Move}(vw)$, we observe the following useful property.

Lemma 7.29. *Let $\mathcal{T}$ be a branch decomposition, $r = uv \in E(\mathcal{T})$ an edge of $\mathcal{T}$, and $r' = vw \in E(\mathcal{T})$ another edge of $\mathcal{T}$ incident to r. For all nodes $x \in V(\mathcal{T}) \setminus \{v\}$, the r-subtree of x is the same as the r'-subtree of x.*

In particular, as the dynamic programming table of a node depends only on its r-subtree, it suffices to re-compute only the dynamic programming table of the node v in $\mathcal{O}(t(k))$ time when using $\mathsf{Move}(vw)$. The $\mathsf{Width}()$ operation is typically implemented without dynamic programming, using some other auxiliary data structure. The $\mathsf{CanImprove}()$ operation is implemented by combining the information of the dynamic programming tables of u and v in an appropriate way. Then, the $\mathsf{EditSet}()$ operation is implemented by tracing the dynamic programming backwards, working with a representation of the minimum split (r, C_1, C_2, C_3) that allows for efficiently determining whether C_i and $\mathcal{T}_r[w]$ intersect. The $\mathsf{Improve}(R, (N_1, N_2, N_3))$ operation is a direct application of Lemma 7.23 followed by computing the dynamic programming tables of the $|R|$ new nodes inserted by Lemma 7.23 in $\mathcal{O}(t(k) \cdot |R|)$ time, and possibly also updating other auxiliary data structures. The implementation of $\mathsf{Output}()$ is typically straightforward, as it just amounts to outputting the branch decomposition that the data structure has been maintaining.

7.3.3 General algorithm

We present a general algorithm that uses the improvement data structure to either improve the width of a given branch decomposition from k to $k-1$ or to conclude that it is already a 2-approximation, with running time $t(k) \cdot 2^{\mathcal{O}(k)}n$.

Our algorithm is described as a pseudocode Algorithm 3. The algorithm is a depth-first-search on the given branch decomposition $\mathcal{T}$, where whenever we return from a subtree via an edge uv of width $f(uv) = k$, we check if there exists a minimum split (uv, C_1, C_2, C_3) of $\mathcal{T}$. If there is no such minimum split, then we conclude that $\mathcal{T}$ is already a 2-approximation. If there is such minimum split, then we improve $\mathcal{T}$ with the improvement operation. We need to be careful to proceed so that the improvement does not break invariants of depth-first-search, and the extra work caused by improving with an edit set R can be bounded by $\mathcal{O}(t(k) \cdot |R|)$.

Algorithm 3 Iterative improvement.

Input: Branch decomposition $\mathcal{T} = (T, \lambda)$ of a connectivity function f.
Output: A branch decomposition of f of width at most $\mathsf{width}(\mathcal{T}) - 1$ or the conclusion that $\mathsf{width}(\mathcal{T}) \leq 2\mathsf{bw}(f)$.

1: Let $k \leftarrow \mathsf{width}(\mathcal{T})$
2: Let `state` be an array initialized with the value `unseen` for all nodes of $\mathcal{T}$, including new nodes that will be created by the improvement operation.
3: Let s be an arbitrary leaf node of T
4: $v \leftarrow s$
5: $u \leftarrow$ the neighbor of v
6: `state`[v] $\leftarrow$ `open`
7: `state`[u] $\leftarrow$ `open`
8: **while** `state`[u] $=$ `open` **do**
9: **if** exists $w \in N_T(u)$ with `state`[w] $=$ `unseen` **then**
10: $v \leftarrow u$
11: $u \leftarrow w$
12: `state`[u] $\leftarrow$ `open`
13: **else if** $f(uv) < k$ **then**
14: **if** $v = s$ **then return** $\mathcal{T}$
15: **else**
16: `state`[u] $\leftarrow$ `closed`
17: $u \leftarrow v$
18: $v \leftarrow$ the node $v \in N_T(u)$ with `state`$[v]$ $=$ `open`
19: ▷ Such a node v is unique.
20: **else**
21: **if** exists a minimum split (uv, C_1, C_2, C_3) of $\mathcal{T}$ **then**
22: $\mathcal{T} \leftarrow \mathsf{Improve}(\mathcal{T}, (uv, C_1, C_2, C_3))$
23: ▷ Where the improvement operation works as in Lemma 7.23, i.e., by removing the edit set R and inserting a connected subtree of $|R|$ nodes in its place.
24: $v \leftarrow$ the node $v \in N_T(R)$ with `state`$[v]$ $=$ `open`
25: $u \leftarrow$ the node $u \in N_T(v)$ that was inserted by the improvement
26: ▷ Such nodes v and u are unique.
27: `state`[u] $\leftarrow$ `open`
28: **else**
29: **conclude** $\mathsf{width}(\mathcal{T}) \leq 2\mathsf{bw}(f)$

Let us explain how Algorithm 3 is implemented with the improvement data structure. We always maintain that the root edge uv in the improvement data structure corresponds to the edge uv in Algorithm 3. We start by calling $\mathsf{Init}(\mathcal{T}, uv)$ after Line 7. In the cases of Line 9 and Line 15 the edge uv is changed to an incident edge vw, which is done by the $\mathsf{Move}(vw)$ operation. The edge uv is changed also after the improvement operation. There we can move to the appropriate edge with $|R|$ $\mathsf{Move}(vw)$ operations. Now that the edge uv of Algorithm 3 corresponds to the edge uv of the improvement data structure, all non-elementary operations of Algorithm 3 can be performed with the improvement data structure. In particular, checking $f(uv)$ on Line 13 is done by $\mathsf{Width}()$, Line 21

corresponds to CanImprove(), and Line 22 corresponds to EditSet() and Improve(). The returned edit set is also used to determine the node v on Line 24.

The rest of this section is devoted to proving the correctness and the running time of Algorithm 3. The next lemma shows that adding the improvement operation does not significantly change the properties of depth-first-search and provides the key argument for proving the correctness.

Lemma 7.30. *Algorithm 3 maintains the invariant that the nodes with state* `open` *form a path $w_1, \ldots, w_\ell$ in T, where $\ell \geq 2$, $w_1 = s$, $w_{\ell-1} = v$, and $w_\ell = u$.*

Proof. This invariant is satisfied at the beginning of the algorithm. There are three cases in the if-else structure that do not terminate the algorithm and alter u, v, or the states, i.e., the cases of Line 9, Line 15, and Line 21. The case of Line 9 maintains the invariant by extending the path by one node. The case of Line 15 maintains the invariant by removing the last node of the path. In the case of Line 21, recall that both u and v are in the edit set R and the edit set is a connected subtree of T, so the improvement removes some suffix $w_j, \ldots, w_\ell$ of the path. Together with the fact that w_1 is a leaf and thus $w_1 \notin R$ (see Lemma 7.21), this implies that the node v determined in Line 24 must be the node w_{j-1} of the path. Finally, the path is extended by one node in Lines 25 and 27. □

The next lemma will be used to prove the correctness of Algorithm 3 in the case when it returns an improved branch decomposition.

Lemma 7.31. *If Algorithm 3 reaches Line 14, i.e., terminates by returning a branch decomposition, then all nodes except v and u have state* `closed`.

Proof. We show that Algorithm 3 maintains the invariant that if a node w is closed, then all nodes w' in the s-subtree of w are also closed. This is trivially maintained by the case of Line 9. In other cases all of the neighbors of u except v are closed due to Lemma 7.30. This implies that the case of Line 15 also maintains the invariant. The case of Line 21, i.e., the improvement operation, maintains this because by Line 24, the union of the edit set R and the path $w_1, \ldots, w_\ell$ defined in Lemma 7.30 is a connected subtree that contains u, v, and s.

This invariant implies the conclusion of the lemma because at Line 14 all neighbors of u except v are closed. □

We are ready to prove the correctness and the running time of Algorithm 3. In particular, next we complete the proof of the main theorem of this section.

Theorem 7.32. *Let f be a connectivity function for which there exists a improvement data structure with running time $t(k)$. There is an algorithm that, given a branch decomposition $\mathcal{T} = (T, \lambda)$ of f of width k, in time $t(k) \cdot 2^{\mathcal{O}(k)} \cdot |\mathcal{T}|$ either outputs a branch decomposition of f of width at most $k-1$, or correctly concludes that $k \leq 2\mathsf{bw}(f)$.*

Proof. It suffices to prove that Algorithm 3 is correct and runs in time $t(k) \cdot 2^{\mathcal{O}(k)}n$ provided an improvement data structure with running time $t(k)$.

We first show the correctness. The algorithm terminates with the conclusion $\mathsf{width}(\mathcal{T}) \leq 2\mathsf{bw}(f)$ if and only if there is no split of $W = \mathcal{T}[uv]$, where $f(W) = \mathsf{width}(\mathcal{T})$. Therefore, by Lemma 7.5, $\mathsf{width}(\mathcal{T}) = f(W) \leq 2\mathsf{bw}(f)$. For the other case, let $w \neq s$ be a node of T and p be the s-parent of w. We note that the state of w can be closed only if $f(wp) < k$. Therefore, by Lemma 7.31, when Algorithm 3 reaches Line 14, we have that $f(wp) < k$ for all $w \in V(\mathcal{T}) \setminus \{u, v\}$, and by Line 13 we also have $f(uv) < k$. Therefore we have that $f(e) < k$ for all edges e of T, implying that Algorithm 3 is correct when it returns a branch decomposition.

Then we prove the running time. By the definition of a split and Theorem 7.8, the width of $\mathcal{T}$ never increases. By Lemma 7.25, with every improvement, the potential function drops by at least $|R|$, the size of the edit set. While we cannot control the size of the edit set for each new improvement, the total sum of the sizes of the edit sets over all the sequence of improvements does not exceed $\Phi_k(\mathcal{T}) = 2^{\mathcal{O}(k)}n$. Thus the total running time of the improvement operations is $t(k) \cdot 2^{\mathcal{O}(k)}n$ and the total number of new nodes created over the course of the algorithm in improvement operations is $2^{\mathcal{O}(k)}n$. All cases of the algorithm advance the state of some node either from unseen to open or from open to closed, and therefore the total number of operations is $2^{\mathcal{O}(k)}n$ and their total time $t(k) \cdot 2^{\mathcal{O}(k)}n$. □

7.4 Approximating rankwidth

In this section we prove Theorem 1.5, that is, there is an algorithm that for an n-vertex graph G and an integer k in time $2^{2^{\mathcal{O}(k)}}n^2$ either computes a rank decomposition of width at most $2k$, or correctly concludes that the rankwidth of G is more than k. To this end, we define "augmented rank decompositions", show how to implement the improvement data structure of Theorem 7.32 for augmented rank decompositions with running time $t(k) = 2^{2^{\mathcal{O}(k)}}$, and then apply the algorithm of Theorem 7.32 together with iterative compression.

In this section we assume that the input graph G is stored in the adjacency matrix format. In particular, we assume that given two vertices $u, v \in G$, we can check if $uv \in E(G)$ in $\mathcal{O}(1)$ time.

7.4.1 Definitions on rank decompositions

Let us introduce further definitions about rank decompositions.

Let G be a graph and $A \subseteq V(G)$ a set of vertices. Recall from Section 2.3 that a set $R \subseteq A$ is a representative of A if for every vertex $v \in A$ there exists a vertex $u \in R$ with $N(v) \setminus A = N(u) \setminus A$. The representative is minimal if for each $v \in A$ there exists exactly one such $u \in R$. The size of a minimal representative of A is at most $2^{\mathsf{cutrk}(A)}$.

To do computations on minimal representatives, we usually work with representatives of cuts $(A, \overline{A})$. In particular, we define that a pair (R, Q) with $R \subseteq A$ and $Q \subseteq \overline{A}$ is a (minimal) *representative* of $(A, \overline{A})$ if R is a (minimal) representative of A and Q is a (minimal) representative of $\overline{A}$. We say that the *size* of (R, Q) is $|R| + |Q|$.

Given some representative of $(A, \overline{A})$, a minimal representative can be found in polynomial time by the following lemma.

Lemma 7.33. *Let $A \subseteq V(G)$. Given a representative (R, Q) of $(A, \overline{A})$, a minimal representative of $(A, \overline{A})$ can be computed in time $(|R| + |Q|)^{\mathcal{O}(1)}$.*

Proof. Because G is stored as an adjacency matrix, we can explicitly construct the graph $G[R, Q]$ in time $(|R| + |Q|)^{\mathcal{O}(1)}$. Now, any minimal representative of the cut (R, Q) of $G[R, Q]$ is a minimal representative of the cut $(A, \overline{A})$ of G, and can easily be computed in $(|R| + |Q|)^{\mathcal{O}(1)}$ time. □

We will also need the following lemma to work with representatives.

Lemma 7.34. *Let R_A be a representative of A and R_B a representative of B. Then $R_A \cup R_B$ is a representative of $A \cup B$.*

Proof. If $N(v) \setminus A = N(u) \setminus A$, then $N(v) \setminus (A \cup B) = N(u) \setminus (A \cup B)$. □

Next define the A_R-representative of a vertex.

Definition 7.35 (A_R-representative of a vertex)**.** *Let $A \subseteq V(G)$, R a minimal representative of A, and v a vertex $v \in A$. The A_R-representative of v, denoted by $\mathsf{rep}_{A_R}(v)$, is the vertex $u \in R$ with $N(u) \setminus A = N(v) \setminus A$.*

By the definition of a minimal representative, there exists exactly one A_R-representative of a vertex, so the function $\mathsf{rep}_{A_R}(v)$ is well-defined. Using a minimal representative of $(A, \overline{A})$ we can compute $\mathsf{rep}_{A_R}(v)$ efficiently.

Lemma 7.36. *Let $A \subseteq V(G)$. Given a vertex $v \in A$ and a minimal representative (R, Q) of $(A, \overline{A})$, $\mathsf{rep}_{A_R}(v)$ can be computed in $(|R| + |Q|)^{\mathcal{O}(1)}$ time.*

Proof. Test for each $u \in R$ if $N(u) \cap Q = N(v) \cap Q$. □

We also define the A_R-representative of a set.

Definition 7.37 (A_R-representative of a set)**.** *Let $A \subseteq V(G)$, R a minimal A-representative, and $X \subseteq A$. The A_R-representative of X, denoted by $\mathsf{rep}_{A_R}(X)$ is the set $\mathsf{rep}_{A_R}(X) = \bigcup_{v \in X} \{\mathsf{rep}_{A_R}(v)\}$.*

Because $\mathsf{rep}_{A_R}(v)$ is well-defined, $\mathsf{rep}_{A_R}(X)$ is also well-defined. By applying Lemma 7.36, the A_R-representative of a set X can be computed in $|X| \cdot (|R| + |Q|)^{\mathcal{O}(1)}$ time. As any A_R-representative of a set is a subset of R, there are at most $2^{|R|}$ different A_R-representatives of sets. Therefore if $\mathsf{cutrk}(A) \leq k$, there are at most 2^{2^k} different A_R-representatives of sets.

Many computations on A_R-representatives of sets rely on the following observation.

Lemma 7.38. *Let $A \subseteq V(G)$ be a set of vertices, $X \subseteq A$, $Y \subseteq A$, and let R be a minimal representative of A, P be a minimal representative of X, and Q be a minimal representative of Y. Let also $X' \subseteq X$ and $Y' \subseteq Y$. Then it holds that $\mathsf{rep}_{A_R}(X' \cup Y') = \mathsf{rep}_{A_R}(\mathsf{rep}_{X_P}(X') \cup \mathsf{rep}_{Y_Q}(Y'))$.*

7.4.2 Augmented rank decompositions

In order to do dynamic programming efficiently on a rank decomposition, we define the notion of an *augmented rank decomposition*.

Definition 7.39 (Augmented rank decomposition)**.** *An augmented rank decomposition is a pair $(\mathcal{T}, \mathcal{R})$, where $\mathcal{T}$ is a rank decomposition and $\mathcal{R}$ is an auxiliary array that stores for each edge $uv \in E(\mathcal{T})$ a minimal representative $(\mathcal{R}[uv], \mathcal{R}[vu])$ of the cut $(\mathcal{T}[uv], \mathcal{T}[vu])$.*

For an augmented rank decomposition $(\mathcal{T}, \mathcal{R})$, root edge $r \in E(\mathcal{T})$, and a node $w \in V(\mathcal{T})$ we will also use the notation $\mathcal{R}_r[w]$ to denote the minimal representative of $\mathcal{T}_r[w]$ stored

in $\mathcal{R}$. Note that because $\mathcal{R}[uv] \leq 2^{\mathsf{cutrk}(\mathcal{T}[uv])}$, an augmented rank decomposition of width k can be represented in space $\mathcal{O}(2^k n)$.

Next we show that we can efficiently insert a vertex to an augmented rank decomposition.

Lemma 7.40. *Let $v \in V(G)$. Given an augmented rank decomposition $(\mathcal{T}, \mathcal{R})$ of $G \setminus \{v\}$ of width k, an augmented rank decomposition of G of width at most $k+1$ can be computed in $2^{\mathcal{O}(k)}n$ time.*

Proof. We obtain a rank decomposition $\mathcal{T}'$ of G by subdividing an arbitrary edge of $\mathcal{T}$ and inserting v as a leaf connected to the node created by subdividing. The width of $\mathcal{T}'$ is at most $k+1$ because adding one vertex to A increases $\mathsf{cutrk}(A)$ by at most one.

For the new edge incident with v, minimal representatives are easy to compute in $\mathcal{O}(n)$ time. In particular, $\{v\}$ is a minimal representative of $\{v\}$ and a minimal representative of $V(G) \setminus \{v\}$ has one vertex from $N(v)$ and one from $V(G) \setminus (N(v) \cup \{v\})$.

All other edges of $\mathcal{T}'$ correspond to edges $uv \in E(\mathcal{T})$ in the sense that they correspond to some cut $(A', \overline{A'}) = (\mathcal{T}[uv] \cup \{v\}, \mathcal{T}[vu])$ of G. We start by setting for each such A' the representative as $\mathcal{R}[uv] \cup \{v\}$. This is not necessarily a minimal representative of A', but we turn it into minimal later. Now we have representatives of size at most $2^k + 1$ for the sides of cuts containing v.

For the sides of cuts not containing v, we compute minimal representatives by dynamic programming. We root the decomposition at v and proceed from the leaves to the root. For leaves, the minimal representatives have exactly one vertex, the leaf. For non-leaves, we want to compute a minimal representative of a set B corresponding to a subtree with $v \notin B$ such that $B = B_1 \cup B_2$, where we have already computed minimal representatives R_1 and R_2 of B_1 and B_2, and a representative R_A of $\overline{B}$ of size $|R_A| \leq 2^k + 1$. By Lemma 7.34, $R_1 \cup R_2$ is a representative of B, so $(R_1 \cup R_2, R_A)$ is a representative of $(B, \overline{B})$ of size $2^{\mathcal{O}(k)}$, so we use Lemma 7.33 to compute a minimal representative of $(B, \overline{B})$ in time $2^{\mathcal{O}(k)}$. □

7.4.3 Improvement data structure for rankwidth

This rest of this section devoted to proving the following lemma.

Lemma 7.41. *There is a improvement data structure for rank decompositions with running time $t(k) = 2^{2^{\mathcal{O}(k)}}$, where the Init operation requires an augmented rank decomposition and the Output operation outputs an augmented rank decomposition.*

The combination of Theorem 7.32 and lemma 7.41 implies the following corollary.

Corollary 7.42. *There is an algorithm, that given an augmented rank decomposition $(\mathcal{T}, \mathcal{R})$ of G of width k, either outputs an augmented rank decomposition of G of width at most $k-1$ or correctly concludes that $k \leq 2 \cdot \mathsf{rw}(G)$ in time $2^{2^{\mathcal{O}(k)}} n$.*

By inserting vertices one by one, Corollary 7.42 implies the main result of this chapter as follows.

Theorem 1.5. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{2^{\mathcal{O}(k)}} n^2$ either outputs a rank decomposition of G of width at most $2k$ or determines that the rankwidth of G is larger than k.*

Proof. Let us order the vertices of G as $v_1, \ldots, v_n$, and for $i \in [n]$ denote $G_i = G[\{v_1, \ldots, v_i\}]$. Note that $\mathsf{rw}(G_i) \leq \mathsf{rw}(G)$ for all i. Now, the algorithm works by iteratively inserting vertices. In particular, after computing an augmented rank decomposition $(\mathcal{T}_i, \mathcal{R}_i)$ of G_i of width at most $2k$, an augmented rank decomposition $(\mathcal{T}_{i+1}, \mathcal{R}_{i+1})$ of G_{i+1} of width at most $2k$ can be computed by using Lemma 7.40 to insert the vertex v_{i+1}, and then using Corollary 7.42 to improve the width to at most $2k$. If the width of $\mathcal{T}_{i+1}$ is $2k+1$ but the algorithm of Corollary 7.42 returns that $\mathsf{width}(\mathcal{T}_{i+1}) \leq 2 \cdot \mathsf{rw}(G_{i+1})$, then we can return that the rankwidth of G is more than k. □

Our improvement data structure is based on characterizing minimum splits of $\mathcal{T}$ by dynamic programming on the augmented rank decomposition $(\mathcal{T}, \mathcal{R})$ directed towards the root edge $r \in E(\mathcal{T})$. In Subsection 7.4.4 we introduce the objects manipulated in this dynamic programming and prove properties of them and in Subsection 7.4.5 we apply this dynamic programming to provide the improvement data structure.

7.4.4 Dynamic programming

Bui-Xuan et al. [2010] characterized the rank $\mathsf{cutrk}(A)$ of a cut $(A, \overline{A})$ by the existence of an "embedding" of the bipartite graph $G[A, \overline{A}]$ into a certain bipartite graph R_k. Next we define this notion of embedding. In our definition the function describing the embedding is in some sense inversed. This inversion will make manipulating embeddings in dynamic programming easier.

Definition 7.43 (Embedding)**.** *Let G and H be bipartite graphs and let (A_G, B_G) and (A_H, B_H) be bipartitioning cuts of them. A function $f : V(H) \to 2^{V(G)}$ is an embedding of G into H if*

- $f(u) \cap f(v) = \emptyset$ *for* $u \neq v$,
- $A_G = \bigcup_{v \in A_H} f(v)$, $B_G = \bigcup_{v \in B_H} f(v)$, *and*
- *for every pair* $(a_H, b_H) \in A_H \times B_H$ *and every* $(a, b) \in f(a_H) \times f(b_H)$, *it holds that* $ab \in E(G)$ *if and only if* $a_H b_H \in E(H)$.

When talking about embeddings of G into H, we assume that the bipartitioning cuts (A_G, B_G) and (A_H, B_H) are clear from the context. For the graph G, it will be that G is constructed with the notation $G[X, Y]$, in which case the bipartitioning cut (A_G, B_G) of $G[X, Y]$ is (X, Y). For the graph H this will also be made clear soon.

Note that the embedding completely defines the edges of $G[X, Y]$ in terms of the edges of H, and in particular gives a representative of (X, Y) of size $|V(H)|$, as we formalize as follows.

Lemma 7.44. *Let* G *be a graph and* $(A, \overline{A})$ *a cut of* G. *Let* H *be a bipartite graph with a bipartition cut* (A_H, B_H). *Let* $f : V(H) \to 2^{V(G)}$ *be an embedding of* $G[A, \overline{A}]$ *into* H. *Let* g *be a function mapping each* $v \in V(H)$ *to a subset of* $f(v)$ *as*

$$g(v) = \begin{cases} \{u\} \text{ where } u \in f(v) & \text{if } f(v) \text{ is non-empty,} \\ \emptyset & \text{otherwise.} \end{cases}$$

Then $(\bigcup_{v \in A_H} g(v), \bigcup_{v \in B_H} g(v))$ *is a representative of* $(A, \overline{A})$ *of size* $|V(H)|$.

Next we define the graph R_k that will be used to characterize cutrk.

Definition 7.45 (Graph R_k [Bui-Xuan et al., 2010])**.** *For each* $k \geq 0$, *we denote by* R_k *the bipartite graph with a bipartitioning cut* (A, B), *having for each subset* $X \subseteq [k]$ *a vertex* $a_X \in A$ *and a vertex* $b_X \in B$, *(in particular, having* $|A| = 2^k$ *and* $|B| = 2^k$*), and having an edge between* a_X *and* b_Y *if and only if* $|X \cap Y|$ *is odd.*

In the embeddings defined earlier, the graph H will always be the graph R_k for some $k \geq 0$, in which case the bipartitioning cut is (A, B).

Lemma 7.46 (Bui-Xuan et al. [2010])**.** *Let* $A \subseteq V(G)$. *It holds that* $\mathsf{cutrk}(A) \leq k$ *if and only if there is an embedding of* $G[A, \overline{A}]$ *into* R_k.

We will find minimum splits of $\mathcal{T}$ by computing embeddings into R_k by dynamic programming. In order to manipulate embeddings and objects related to embeddings we introduce some notation that naturally extends the definitions of intersections and unions of sets.

Definition 7.47 (Intersection $f \cap X$)**.** *Let $f : V(H) \to 2^A$ be a function and $X \subseteq A$ be a set. We denote by $f \cap X$ the function $f \cap X : V(H) \to 2^X$ with $(f \cap X)(v) = f(v) \cap X$.*

We note that an intersection of an embedding and a set is again an embedding.

Lemma 7.48. *Let A be a set, $C \subseteq A$, and $X \subseteq A$. If $f : V(H) \to 2^A$ is an embedding of $G[A \cap C, A \setminus C]$ into H, then $f \cap X$ is an embedding of $G[X \cap C, X \setminus C]$ into H.*

Finally, we define the union of functions.

Definition 7.49 (Union $f \cup g$)**.** *Let $f : V(H) \to 2^X$ and $g : V(H) \to 2^Y$ be functions. We define function $f \cup g : V(H) \to 2^{X \cup Y}$ with $(f \cup g)(v) = f(v) \cup g(v)$.*

Representatives of embeddings

Next we define the A_R-representative of an embedding, extending the definition of the A_R-representative of a set.

Definition 7.50 (A_R-representative of an embedding)**.** *Let $A \subseteq V(G)$, and R be a minimal representative of A. Let also $f : V(H) \to 2^A$ be an embedding. The A_R-representative of f is the function $g : V(H) \to 2^R$, where $g(v) = \mathsf{rep}_{A_R}(f(v))$.*

The A_R-representative of an embedding is well-defined because the A_R-representative of a set is well-defined. If $\mathsf{cutrk}(A) \leq k$, then the number of A_R-representatives of embeddings into H is at most $(2^{2^k})^{|V(H)|}$. In particular, the number of A_R-representatives of embeddings into R_k is at most $(2^{2^k})^{2 \cdot 2^k} = 2^{2^{\mathcal{O}(k)}}$.

We will define the compatibility and the composition of two representatives of embeddings. The intuition is that embeddings f_X and f_Y of disjoint subgraphs can be merged into an embedding $f_X \cup f_Y$ if and only if their representatives are compatible. Moreover, the representative of $f_X \cup f_Y$ will be the composition of the representatives of f_X and f_Y.

Let X and Y be disjoint subsets of $V(G)$ and $A = X \cup Y$. Let also R be a minimal representative of A, R_X a minimal representative of X, and R_Y a minimal representative of Y. Let also H be a bipartite graph, $g_X : V(H) \to 2^{R_X}$ an X_{R_X}-representative of an embedding, and $g_Y : V(H) \to 2^{R_Y}$ an Y_{R_Y}-representative of an embedding.

Definition 7.51 (Compatibility)**.** *Let (A_H, B_H) be the bipartitioning cut of H. The representatives g_X and g_Y are compatible if for every pair $(a_H, b_H) \in A_H \times B_H$ it holds that*

1. *for every* $(a,b) \in g_X(a_H) \times g_Y(b_H)$ *it holds that* $ab \in E(G) \Leftrightarrow a_H b_H \in E(H)$ *and*
2. *for every* $(a,b) \in g_Y(a_H) \times g_X(b_H)$ *it holds that* $ab \in E(G) \Leftrightarrow a_H b_H \in E(H)$.

Note that compatibility can be tested in $(|V(H)| + |R_X| + |R_Y|)^{\mathcal{O}(1)}$ time. Next we show that two embeddings can be merged into an embedding only if their representatives are compatible.

Lemma 7.52. *Let* $C \subseteq A$ *be a set and let* f_A *be an embedding of* $G[A \cap C, A \setminus C]$ *into* H. *If* g_X *is the* X_{R_X}*-representative of* $f_A \cap X$ *and* g_Y *is the* Y_{R_Y}*-representative of* $f_A \cap Y$, *then* g_X *and* g_Y *are compatible.*

Proof. Let $f_X = f_A \cap X$, $f_Y = f_A \cap Y$, and let (A_H, B_H) be the bipartitioning cut of H. For the case of Item 1 of compatibility it suffices to prove for every pair $(a_H, b_H) \in A_H \times B_H$ and every $(a,b) \in g_X(a_H) \times g_Y(b_H)$ that $ab \in E(G)$ if and only if $a_H b_H \in E(H)$.

As $g_X(a_H)$ is an X_{R_X}-representative of $f_X(a_H)$, there is $a' \in f_X(a_H)$ with $N(a') \setminus X = N(a) \setminus X$. Similarly there is $b' \in f_Y(b_H)$ with $N(b') \setminus Y = N(b) \setminus Y$. Therefore $ab \in E(G)$ if and only if $a'b' \in E(G)$. Since $f_X(a_H) = f_A(a_H) \cap X$ and $f_Y(b_H) = f_A(b_H) \cap Y$, we have that $a' \in f_A(a_H)$ and that $b' \in f_A(b_H)$. Because f_A is an embedding it holds that $a'b' \in E(G)$ if and only if $a_H b_H \in E(H)$. This concludes the proof that the case of Item 1 of compatibility holds.

For the case of Item 2, it suffices to prove the same but for all $(a,b) \in g_Y(a_H) \times g_X(b_H)$. This is similar to the proof for the case of Item 1. □

The composition is defined as the representative of the union.

Definition 7.53 (Composition)**.** *The composition of* g_X *and* g_Y *is the function* $g_A : V(H) \to 2^A$ *defined by* $g_A(v) = \mathsf{rep}_{A_R}(g_X(v) \cup g_Y(v))$ *for all* $v \in V(H)$.

By using a minimal $(A, \overline{A})$-representative (R, Q) the composition can be computed in time $(|V(H)| + |R| + |Q| + |R_X| + |R_Y|)^{\mathcal{O}(1)}$.

Next we prove that two embeddings can be merged into an embedding if their representatives are compatible, and in this case the composition gives the resulting representative.

Lemma 7.54. *Let* $C \subseteq A$ *be a set. Let* g_X *be the* X_{R_X}*-representative of an embedding* f_X *of* $G[X \cap C, X \setminus C]$ *into* H *and* g_Y *the* Y_{R_Y}*-representative of an embedding* f_Y *of* $G[Y \cap C, Y \setminus C]$ *into* H. *If* g_X *and* g_Y *are compatible, then* $f_X \cup f_Y$ *is an embedding of* $G[A \cap C, A \setminus C]$ *into* H *so that the composition of* g_X *and* g_Y *is the* A_R*-representative of* $f_X \cup f_Y$.

Proof. Let (A_H, B_H) be the bipartitioning cut of H. We let $f_A = f_X \cup f_Y$ and observe that f_A is a function $f_A : V(H) \to 2^A$ that satisfies $f_A(u) \cap f_A(v) = \emptyset$ for $u \neq v$, $(X \cap C) \cup (Y \cap C) = (A \cap C) = \bigcup_{v \in A_H} f_A(v)$, and $(X \setminus C) \cup (Y \setminus C) = (A \setminus C) = \bigcup_{v \in B_H} f_A(v)$. Therefore f_A is an embedding of $G[A \cap C, A \setminus C]$ into H if for every pair $(a_H, b_H) \in A_H \times B_H$ and every $(a, b) \in f_A(a_H) \times f_A(b_H)$ it holds that $ab \in E(G)$ if and only if $a_H b_H \in E(H)$. We say that f_A is *good* for a pair $(a, b) \in (A \cap C) \times (A \setminus C)$ if this holds for this pair. Now f_A is an embedding of $G[A \cap C, A \setminus C]$ into H if it is good for every pair $(a, b) \in (A \cap C) \times (A \setminus C)$.

Because f_X is an embedding of $G[X \cap C, X \setminus C]$ into H we have that f_X is good for all pairs in $(X \cap C) \times (X \setminus C)$ and therefore f_A is good for all pairs in $(X \cap C) \times (X \setminus C)$. Analogously because f_Y is an embedding of $G[Y \cap C, Y \setminus C]$ into H we have that f_A is good for all pairs in $(Y \cap C) \times (Y \setminus C)$. It remains to prove that f_A is good for all pairs in $(X \cap C) \times (Y \setminus C)$ and in $(Y \cap C) \times (X \setminus C)$.

Let $(a, b) \in (X \cap C) \times (Y \setminus C)$. The set $g_X(a_H)$ is a X_{R_X}-representative of $f_X(a_H) \supseteq \{a\}$, so there exists $a' \in g_X(a_H)$ so that $N(a') \setminus X = N(a) \setminus X$. Similarly there exists $b' \in g_Y(b_H)$ so that $N(b') \setminus Y = N(b) \setminus Y$. Therefore $ab \in E(G)$ if and only if $a'b' \in E(G)$. Now, by compatibility (Item 1) it holds that $a'b' \in E(G)$ if and only if $a_H b_H \in E(G)$. Therefore f_A is good for (a, b).

The proof for $(a, b) \in (Y \cap C) \times (X \setminus C)$ is symmetric, using compatibility (Item 2) instead. Therefore f_A is an embedding of $G[A \cap C, A \setminus C]$ into H. Finally, by Lemma 7.38 the composition of g_X and g_Y is the A_R-representative of f_A. □

Improvement embeddings

In order to construct a minimum split (C_1, C_2, C_3) of $W = \mathcal{T}[uv]$, we build six embeddings simultaneously in the dynamic programming, three to bound $\mathsf{cutrk}(C_i)$ and three to bound $\mathsf{cutrk}(C_i \cap W)$. We of course also need to bound $\mathsf{cutrk}(C_i \cap \overline{W})$, but note that $G[(C_i \cap \overline{W}) \cap W, (\overline{C_i \cap \overline{W}}) \cap W] = G[\emptyset, W]$, so dynamic programming is not required for building the side of W of the embedding $G[C_i \cap \overline{W}, \overline{C_i \cap \overline{W}}]$.

Definition 7.55 (A-restricted improvement embedding)**.** *Let $A \subseteq V(G)$. A 10-tuple*

$$E = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

is an A-restricted improvement embedding if there exists a tripartition (C_1, C_2, C_3) of A so that

1. *for each $i \in [3]$, f_i^C is an embedding of $G[A \cap C_i, A \cap \overline{C_i}]$ into R_{k_i} and*

2. for each $i \in [3]$, f_i^W is an embedding of $G[A \cap C_i, A \cap \overline{C_i}]$ into R_ℓ.

Note that an A-restricted improvement embedding E uniquely defines such a tripartition (C_1, C_2, C_3) of A. We call such a tripartition *the tripartition of* E. We say that E *cuts a set* if at least two sets in its tripartition (C_1, C_2, C_3) intersect it, and that E *r-cuts* a node w if it cuts $\mathcal{T}_r[w]$. We call the quadruple (k_1, k_2, k_3, ℓ) the *shape of* E. An A-restricted improvement embedding is k-bounded if $k_1, k_2, k_3, \ell \leq k$.

In the following lemma we introduce the notation $E \cap X$, where E is an A-restricted improvement embedding and $X \subseteq A$.

Lemma 7.56. *Let $A \subseteq V(G)$, $X \subseteq A$, and*

$$E = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

be an A-restricted improvement embedding with tripartition (C_1, C_2, C_3). The tuple

$$E \cap X = (f_1^C \cap X, f_2^C \cap X, f_3^C \cap X, f_1^W \cap X, f_2^W \cap X, f_3^W \cap X, k_1, k_2, k_3, \ell)$$

is an X-restricted improvement embedding with tripartition $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$.

Proof. By Lemma 7.48, for each i $f_i^C \cap X$ is an embedding of $G[X \cap C_i, X \cap \overline{C_i}]$ to R_{k_i} and f_i^W is an embedding of $G[X \cap C_i, X \cap \overline{C_i}]$ to R_ℓ. □

We also define the union of improvement embeddings, extending the definition of the union of embeddings.

Definition 7.57. *Let X and Y be disjoint subsets,*

$$E_1 = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

an X-restricted improvement embedding, and

$$E_2 = (g_1^C, g_2^C, g_3^C, g_1^W, g_2^W, g_3^W, k_1, k_2, k_3, \ell)$$

an Y-restricted improvement embedding with the same shape. We denote by $E_1 \cup E_2$ the 10-tuple

$$E_1 \cup E_2 = (f_1^C \cup g_1^C, f_2^C \cup g_2^C, f_3^C \cup g_3^C, f_1^W \cup g_1^W, f_2^W \cup g_2^W, f_3^W \cup g_3^W, k_1, k_2, k_3, \ell).$$

Note that if the tripartition of X is $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$, the tripartition of Y is $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$, and $E_1 \cup E_2$ is indeed an $X \cup Y$-restricted improvement embedding, then the tripartition of $E_1 \cup E_2$ is (C_1, C_2, C_3).

We will use dynamic programming on the augmented rank decomposition $(\mathcal{T}, \mathcal{R})$ to compute improvement embeddings, minimizing the number of nodes of $\mathcal{T}$ cut by the embedding.

Representatives of improvement embeddings

The definition of the A_R-representative of an improvement embedding extends the definition of the A_R-representative of an embedding.

Definition 7.58 (A_R-representative of improvement embedding)**.** *Let $A \subseteq V(G)$ and R a minimal representative of A. Let also*

$$E = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

be an A-restricted improvement embedding. The A_R-representative of E is the tuple

$$(g_1^C, g_2^C, g_3^C, g_1^W, g_2^W, g_3^W, k_1, k_2, k_3, \ell),$$

where each such g is the A_R-representative of the corresponding embedding f.

The shape of the A_R-representative of E is the same as the shape of E. When $\mathsf{cutrk}(A) \leq k$, the number of A_R-representatives of k-bounded improvement embeddings is $(2^{2^{\mathcal{O}(k)}})^6 k^4 = 2^{2^{\mathcal{O}(k)}}$. We naturally extend the definitions of composition and compatibility to representatives of improvement embeddings.

Let G be a graph, X and Y disjoint subsets of $V(G)$, and $A = X \cup Y$. Let also R be a minimal representative of A, R_X a minimal representative of X, and R_Y a minimal representative of Y. Let $E_X = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$ be the X_{R_X}-representative of an X-restricted improvement embedding and $E_Y = (g_1^C, g_2^C, g_3^C, g_1^W, g_2^W, g_3^W, k'_1, k'_2, k'_3, \ell')$ the Y_{R_Y}-representative of an Y-restricted improvement embedding.

Definition 7.59 (C-Compatibility)**.** *E_X and E_Y are C-compatible if they have the same shape and for each $i \in [3]$, f_i^C and g_i^C are compatible.*

Definition 7.60 (Compatibility)**.** *E_X and E_Y are compatible if they are C-compatible and for each $i \in [3]$, f_i^W and g_i^W are compatible.*

We derive the following lemma directly from Lemma 7.52.

Lemma 7.61. *Let E be an A-restricted improvement embedding. If E_X is the X_{R_X}-representative of $E \cap X$ and E_Y is the Y_{R_Y}-representative of $E \cap Y$, then E_X and E_Y are compatible.*

Proof. Apply Lemma 7.52 to each embedding in E. □

Next we define the composition of two representatives of improvement embeddings, extending the definition of the composition of representatives of embeddings.

Definition 7.62 (Composition)**.** *If E_X and E_Y are compatible, then the composition of E_X and E_Y is the pointwise composition of E_X and E_Y, i.e., the composition is the 10-tuple*

$$E_A = (h_1^C, h_2^C, h_3^C, h_1^W, h_2^W, h_3^W, k_1, k_2, k_3, \ell),$$

where for each $i \in [3]$ h_i^C is the composition of f_i^C and g_i^C, and h_i^W is the composition of f_i^W and g_i^W.

Next we prove the main lemma for computing improvement embeddings by dynamic programming, which is an extension of Lemma 7.54.

Lemma 7.63. *Let X and Y be disjoint sets, $A = X \cup Y$, R a minimal representative of A, R_X a minimal representative of X, and R_Y a minimal representative of Y. Let E_1 be an X-restricted improvement embedding and E_2 a Y-restricted improvement embedding. Let E_X be the X_{R_X}-representative of E_1 and E_Y the Y_{R_Y}-representative of E_2. If E_X and E_Y are compatible, then $E_1 \cup E_2$ is an A-restricted improvement embedding and the composition of E_X and E_Y is the A_R-representative of $E_1 \cup E_2$.*

Proof. Denote

$$E_1 = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

and

$$E_2 = (g_1^C, g_2^C, g_3^C, g_1^W, g_2^W, g_3^W, k_1, k_2, k_3, \ell).$$

Let (C_1^X, C_2^X, C_3^X) be the tripartition of E_1 and (C_1^Y, C_2^Y, C_3^Y) the tripartition of E_2. By Lemma 7.54, $f_i^C \cup g_i^C$ is an embedding of $G[C_i^X \cup C_i^Y, A \setminus (C_i^X \cup C_i^Y)]$ to R_{k_i} and $f_i^W \cup g_i^W$ is an embedding of $G[C_i^X \cup C_i^Y, A \setminus (C_i^X \cup C_i^Y)]$ to R_ℓ for every i. Therefore

$$E = (f_1^C \cup g_1^C, f_2^C \cup g_2^C, f_3^C \cup g_3^C, f_1^W \cup g_1^W, f_2^W \cup g_2^W, f_3^W \cup g_3^W, k_1, k_2, k_3, \ell)$$

is an improvement embedding with tripartition $(C_1^X \cup C_1^Y, C_2^X \cup C_2^Y, C_3^X \cup C_3^Y)$. By Lemma 7.54 the composition of E_X and E_Y is the A_R-representative of E. □

Finding the split

In the dynamic programming we will determine if there exists a split of W based on the representatives of W-restricted improvement embeddings and the representatives of $\overline{W}$-restricted improvement embeddings. For this purpose we will define the root-compatibility of two representatives of improvement embeddings, characterizing whether they can be merged to yield a split of W.

Let $W \subseteq V(G)$, R_W a minimal representative of W, and $R_{\overline{W}}$ a minimal representative of $\overline{W}$. Let also

$$E_W = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

be a W_{R_W}-representative of a W-restricted improvement embedding and

$$E_{\overline{W}} = (g_1^C, g_2^C, g_3^C, g_1^W, g_2^W, g_3^W, k_1, k_2, k_3, \ell)$$

be a $\overline{W}_{R_{\overline{W}}}$-representative of an $\overline{W}$-restricted improvement embedding so that E_W and $E_{\overline{W}}$ have the same shape.

Definition 7.64 (Root-compatibility)**.** *E_W and $E_{\overline{W}}$ are root-compatible if they are C-compatible and*

1. *for each i, there exists an embedding $f_i^{\overline{W}}$ of $G[\emptyset, \overline{W}]$ into R_ℓ so that f_i^W is compatible with the $\overline{W}_{R_{\overline{W}}}$-representative of $f_i^{\overline{W}}$ and*
2. *for each i, there exists an embedding $g_i^{\overline{W}}$ of $G[\emptyset, W]$ into R_ℓ so that g_i^W is compatible with the W_{R_W}-representative of $g_i^{\overline{W}}$.*

The definition does not directly provide an efficient algorithm for checking root-compatibility, but a couple of observations and brute force yields a $2^{2^{\mathcal{O}(k)}}$ time algorithm as follows.

Lemma 7.65. *Let $\mathsf{cutrk}(W) \leq k$ and suppose that the improvement embeddings is k-bounded. Given E_W, $E_{\overline{W}}$, R_W, and $R_{\overline{W}}$, the root-compatibility of E_W and $E_{\overline{W}}$ can be checked in time $2^{2^{\mathcal{O}(k)}}$.*

Proof. We prove the case of Item 1. The case of Item 2 is similar.

Suppose there exists such an embedding $f_i^{\overline{W}}$. Let (A_H, B_H) be the bipartitioning cut of R_ℓ and let $u_H, v_H \in B_H$ be a pair of distinct vertices in B_H. Suppose that there exists $u, v \in \overline{W}$ such that $N(u) \cap W = N(v) \cap W$, $u \in f_i^{\overline{W}}(u_H)$, and $v \in f_i^{\overline{W}}(v_H)$. Note that now, if we remove u from $f_i^{\overline{W}}(u_H)$ and insert it into $f_i^{\overline{W}}(v_H)$, we obtain an embedding

whose $\overline{W}_{R_{\overline{W}}}$-representative is compatible with the same W_{R_W}-representatives as $f_i^{\overline{W}}$. Therefore, we can assume for any $u, v \in \overline{W}$ that if $N(u) \cap W = N(v) \cap W$, then there exists $v_H \in B_H$ so that $\{u, v\} \subseteq f_i^{\overline{W}}(v_H)$.

As for any $v \in \overline{W}$ there is a vertex $v_R \in R_{\overline{W}}$ with $N(v) \cap W = N(v_R) \cap W$, it is sufficient to enumerate all intersections $f_i^{\overline{W}} \cap R_{\overline{W}}$ and assign each v to the same vertex of B_H as v_R. Note that in this case, the $\overline{W}_{R_{\overline{W}}}$-representative of $f_i^{\overline{W}}$ depends only on the intersection $f_i^{\overline{W}} \cap R_{\overline{W}}$. The number of the intersections is at most $\left(2^{|R_{\overline{W}}|}\right)^{|B_H|} \leq (2^{2^k})^{2^k} = 2^{2^{\mathcal{O}(k)}}$ and each can be checked in time $2^{\mathcal{O}(k)}$. □

We also introduce the definition of C_i-emptiness to denote whether none of the vertices have been assigned to C_i in the tripartition.

Definition 7.66 (C_i-empty)**.** *Let $E_R = (f_1^C, f_2^C, f_3^C, \ldots)$ be the A_R-representative of an A-restricted improvement embedding. Let $i \in [3]$, and let (A_H, B_H) be the bipartitioning cut of R_{k_i}. E_R is C_i-empty if for every $v \in A_H$ it holds that $f_i^C(v) = \emptyset$.*

The definition of C_i-emptiness is for determining which of the parts of a corresponding tripartition intersect A.

Lemma 7.67. *Let E_R be the A_R-representative of an A-restricted improvement embedding E and (C_1, C_2, C_3) the tripartition of E. It holds that $C_i = \emptyset$ if and only if E_R is C_i-empty.*

Finally, we describe how to find splits based on representatives of improvement embeddings.

Lemma 7.68. *Let $\mathcal{T}$ be a rank decomposition, $uv = r \in E(\mathcal{T})$, and $W = \mathcal{T}[uv]$. Denote $X = W = \mathcal{T}_r[u]$, $Y = \overline{W} = \mathcal{T}_r[v]$, and let R_X be a minimal representative of X and R_Y a minimal representative of Y. There exists a split (C_1, C_2, C_3) of W with $\mathsf{cutrk}(C_i) \leq k_i$ for each $i \in [3]$ if and only if there exists E_u, E_v, and ℓ such that*

1. *E_u is the X_{R_X}-representative of an X-restricted improvement embedding with shape (k_1, k_2, k_3, ℓ) whose tripartition $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$ is,*
2. *E_v is the Y_{R_Y}-representative of an Y-restricted improvement embedding with shape (k_1, k_2, k_3, ℓ) whose tripartition $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$ is,*
3. *E_u and E_v are root-compatible, and*
4. *$k_1, k_2, k_3, \ell < \mathsf{cutrk}(W)$.*

Proof. Let us first prove the if direction. Let

$$E_X = (f_1^C, f_2^C, f_3^C, f_1^W, f_2^W, f_3^W, k_1, k_2, k_3, \ell)$$

be a X-restricted improvement embedding whose X_{R_X}-representative E_u is and whose tripartition $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$ is. Let also

$$E_Y = (g_1^C, g_2^C, g_3^C, g_1^{\overline{W}}, g_2^{\overline{W}}, g_3^{\overline{W}}, k_1, k_2, k_3, \ell)$$

be a Y-restricted improvement embedding whose Y_{R_Y}-representative E_u is and whose tripartition $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$ is.

As E_u and E_v are C-compatible, Lemma 7.54 implies that for each $i \in [3]$, $f_i^C \cup g_i^C$ is an embedding of $G[C_i, \overline{C_i}]$ to k_i. Therefore by Lemma 7.46 it holds that $\mathsf{cutrk}(C_i) \leq k_i$.

As E_u and E_v are root-compatible, by the definition of root-compatibility and Lemma 7.54 for each i there exists an embedding $f_i^{\overline{W}}$ of $G[\emptyset, \overline{W}]$ to R_ℓ so that $f_i^W \cup f_i^{\overline{W}}$ is an embedding of $G[C_i \cap W, (\overline{C_i} \cap W) \cup \overline{W}] = G[C_i \cap W, \overline{C_i \cap W}]$ to R_ℓ. Therefore by Lemma 7.46 it holds that $\mathsf{cutrk}(C_i \cap W) \leq \ell$.

Symmetrically, by root-compatibility there exists an embedding g_i^W of $G[\emptyset, W]$ to R_ℓ so that $g_i^{\overline{W}} \cup g_i^W$ is an embedding of $G[C_i \cap \overline{W}, (\overline{C_i} \cap \overline{W}) \cup W] = G[C_i \cap \overline{W}, \overline{C_i \cap \overline{W}}]$ to R_ℓ. Therefore by Lemma 7.46 it holds that $\mathsf{cutrk}(C_i \cap \overline{W}) \leq \ell$.

Then we prove the only if direction. Let (C_1, C_2, C_3) be a split of W, where $\mathsf{cutrk}(C_i) = k_i$. Let also $\ell = \mathsf{cutrk}(W) - 1$. By Lemma 7.46, for each i there exists an embedding f_i^C of $G[C_i, \overline{C_i}]$ to R_{k_i}, an embedding f_i^W of $G[C_i \cap W, \overline{C_i \cap W}]$ to R_ℓ, and an embedding $f_i^{\overline{W}}$ of $G[C_i \cap \overline{W}, \overline{C_i \cap \overline{W}}]$ to R_ℓ.

Let

$$E_X = (f_1^C \cap X, f_2^C \cap X, f_3^C \cap X, f_1^W \cap X, f_2^W \cap X, f_3^W \cap X, k_1, k_2, k_3, \ell)$$

and note that the tripartition of E_X is $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$. Let also

$$E_Y = (f_1^C \cap Y, f_2^C \cap Y, f_3^C \cap Y, f_1^{\overline{W}} \cap Y, f_2^{\overline{W}} \cap Y, f_3^{\overline{W}} \cap Y, k_1, k_2, k_3, \ell)$$

and note that the tripartition of E_Y is $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$. Let E_u be the X_{R_X}-representative of E_X and E_v the Y_{R_Y}-representative of E_Y. Note that now, E_u and E_v satisfy Items 1 and 2.

By Lemma 7.52, E_u and E_v are C-compatible. Note that also by Lemma 7.52, for each i, $f_i^W \cap Y$ is an embedding of $G[\emptyset, \overline{W}]$ to R_ℓ whose Y_{R_Y}-representative is compatible with the X_{R_X}-representative of $f_i^W \cap X$, and $f_i^{\overline{W}} \cap X$ is an embedding of $G[\emptyset, W]$ whose

X_{R_X}-representative is compatible with Y_{R_Y}-representative of $f_i^{\overline{W}} \cap Y$. Therefore E_u and E_v are root-compatible by the definition of root-compatibility. □

7.4.5 The data structure

Finally, we provide the improvement data structure for augmented rank decompositions using dynamic programming building on the previous subsections. Throughout this subsection, we assume that we are maintaining an augmented rank decomposition $(\mathcal{T}, \mathcal{R})$ of width $\mathsf{width}(\mathcal{T}) \leq k$ and therefore we are only interested in k-bounded improvement embeddings.

Concrete representatives

In order to maintain an augmented rank decomposition in the improvement operation, we need to construct a minimal representative of each set C_i of the split (C_1, C_2, C_3). To do this, we maintain "concrete representatives" in the dynamic programming.

Definition 7.69 (Concrete representative)**.** *Let H be bipartite graph, $A \subseteq V(G)$, and $f : V(H) \to 2^A$ a function. A concrete representative of f is a function g defined in Lemma 7.44, i.e., a function g mapping each $v \in V(H)$ to a subset of $f(v)$ as follows:*

$$g(v) = \begin{cases} \{u\} \text{ where } u \in f(v) & \text{if } f(v) \text{ is non-empty} \\ \emptyset & \text{otherwise.} \end{cases}$$

Let $E = (f_1^C, f_2^C, f_3^C, \ldots)$ be an A-restricted improvement embedding. A concrete representative of E is a triple (g_1^C, g_2^C, g_3^C), where each g_i^C is a concrete representative of f_i^C.

Note that a concrete representative can be represented in $\mathcal{O}(|V(H)|)$ space. By Lemma 7.44, a concrete representative of an embedding of $G[C_i, \overline{C_i}]$ into H can be turned into a representative of $(C_i, \overline{C_i})$ of size $|V(H)|$. In particular, using a concrete representative of a $V(G)$-restricted improvement embedding with tripartition (C_1, C_2, C_3) we can compute a minimal representatives of each C_i in time $2^{\mathcal{O}(k)}$.

We define the union of two concrete representatives naturally.

Definition 7.70 (Union of concrete representatives)**.** *Let f_R be a concrete representative of a function $f : V(H) \to 2^X$ and g_R a concrete representative of a function $g : V(H) \to 2^Y$. We denote by $f_R \cup g_R$ the function mapping each $v \in V(H)$ to a subset of $f(v) \cup g(v)$*

as follows:

$$(f_R \cup g_R)(v) = \begin{cases} f_R(v), & \text{if } f_R(v) \neq \emptyset \\ g_R(v), & \text{otherwise.} \end{cases}$$

The union of concrete representatives of improvement embeddings is the pointwise union of such triples of concrete representatives.

Observe that such $f_R \cup g_R$ is a concrete representative of $f \cup g$.

Dynamic programming tables

In the improvement data structure we maintain a dynamic programming table for each node $w \in V(\mathcal{T})$. We call this table the *r-table* of the node w to signify that this table contains information about the r-subtree of w. Next we formally define an r-table.

Definition 7.71 (r-table)**.** *For an augmented rank decomposition $(\mathcal{T}, \mathcal{R})$, root edge $r \in E(\mathcal{T})$, node $w \in V(\mathcal{T})$, and $A = \mathcal{T}_r[w]$, $R = \mathcal{R}_r[w]$, an r-table of w is a triple $(\mathcal{E}, \mathcal{I}, \mathcal{C})$, where*

1. *$\mathcal{E}$ is the set of all A_R-representatives of k-bounded A-restricted improvement embeddings,*
2. *$\mathcal{I}$ is a function mapping each $E_R \in \mathcal{E}$ to the least integer i such that there exists an A-restricted improvement embedding E whose A_R-representative E_R is and which r-cuts i nodes of the r-subtree of w, and*
3. *$\mathcal{C}$ is a function mapping each $E_R \in \mathcal{E}$ to a concrete representative of an A-restricted improvement embedding E such that E_R is the A_R-representative of E and E r-cuts $\mathcal{I}(E_R)$ nodes of the r-subtree of w.*

Note that an r-table can be represented by making use of $2^{2^{\mathcal{O}(k)}}$ space.

To correctly construct the improvement that matches the concrete representation obtained, we need to spell out some additional properties of r-tables, which will be naturally satisfied by the way the r-tables will be constructed.

Definition 7.72 (Local and global representation)**.** *Let $(\mathcal{T}, \mathcal{R})$ be an augmented rank decomposition, $r \in E(\mathcal{T})$, $w \in V(\mathcal{T})$, $A = \mathcal{T}_r[w]$, and $R = \mathcal{R}_r[w]$. An r-table $(\mathcal{E}, \mathcal{I}, \mathcal{C})$ of w locally represents an A-restricted improvement embedding E if there exists $E_R \in \mathcal{E}$ so that E_R is the A_R-representative of E, $\mathcal{I}(E_R)$ is the number of nodes in the r-subtree of*

w r-cut by E, and $\mathcal{C}(E_R)$ is a concrete representative of E. The r-table of w globally represents E if the r-tables of all nodes w' in the r-subtree of w (including w itself) locally represent $E \cap \mathcal{T}_r[w']$.

Definition 7.73 (Linked r-table)**.** *A linked r-table of w is a 4-tuple $(\mathcal{E}, \mathcal{I}, \mathcal{C}, \mathcal{L})$ so that $(\mathcal{E}, \mathcal{I}, \mathcal{C})$ is an r-table of w and for each $E_R \in \mathcal{E}$ there exists an A-restricted improvement embedding E so that*

1. *E_R is the A_R-representative of E,*
2. *$(\mathcal{E}, \mathcal{I}, \mathcal{C})$ globally represents E, and*
3. *if w is non-leaf and has r-children w_1 and w_2 with r-tables $(\mathcal{E}_1, \mathcal{I}_1, \mathcal{C}_1)$ and $(\mathcal{E}_2, \mathcal{I}_2, \mathcal{C}_2)$, then $\mathcal{L}$ is a function mapping E_R to a pair $\mathcal{L}(E_R) = (E_1, E_2)$ such that $E_1 \in \mathcal{E}_1$, $E_2 \in \mathcal{E}_2$, E_1 is the $\mathcal{T}_r[w_1]_{\mathcal{R}_r[w_1]}$-representative of $E \cap \mathcal{T}_r[w_1]$, and E_2 is the $\mathcal{T}_r[w_2]_{\mathcal{R}_r[w_2]}$-representative of $E \cap \mathcal{T}_r[w_2]$.*

The existence of a linked r-table will be formally proved in Lemma 7.75 when also its construction is given. Note that a linked r-table of a node w can be represented in $2^{2^{\mathcal{O}(k)}}$ space. We start implementing the dynamic programming from the leaves.

Lemma 7.74. *Let $(\mathcal{T}, \mathcal{R})$ be an augmented rank decomposition, $r \in E(\mathcal{T})$, and w a leaf node of $\mathcal{T}$. A linked r-table of w can be constructed in time $2^{\mathcal{O}(k)}$.*

Proof. Let $A = \mathcal{T}_r[w]$. As $|A| = 1$, the number of A-restricted improvement embeddings is $2^{\mathcal{O}(k)}$ and we can iterate over all of them and construct the r-table directly by definition. Moreover, the r-table is by definition linked because A is the minimal representative of itself and so there is bijection between A-restricted improvement embeddings and their A_A-representatives. □

The next lemma is the main dynamic programming lemma. It specifies the computation of linked r-tables for non-leaf nodes.

Lemma 7.75. *Let $(\mathcal{T}, \mathcal{R})$ be an augmented rank decomposition, $r \in E(\mathcal{T})$, w a non-leaf node of $\mathcal{T}$, and w_1 and w_2 the r-children of w. Given linked r-tables of the nodes w_1 and w_2, a linked r-table $(\mathcal{E}, \mathcal{I}, \mathcal{C}, \mathcal{L})$ of w can be constructed in time $2^{2^{\mathcal{O}(k)}}$.*

Proof. Let $A = \mathcal{T}_r[w]$, $X = \mathcal{T}_r[w_1]$, $Y = \mathcal{T}_r[w_2]$, $R = \mathcal{R}_r[w]$, $R_X = \mathcal{R}_r[w_1]$, and $R_Y = \mathcal{R}_r[w_2]$. Let $(\mathcal{E}_1, \mathcal{I}_1, \mathcal{C}_1)$ be the given r-table of w_1 and $(\mathcal{E}_2, \mathcal{I}_2, \mathcal{C}_2)$ the given r-table of w_2.

Let E be a k-bounded A-restricted improvement embedding. Note that the number of nodes in the r-subtree of w that E r-cuts is $i_1 + i_2 + i_w$, where i_1 is the number of nodes

in the r-subtree of w_1 that $E \cap X$ r-cuts, i_2 is the number of nodes in the r-subtree of w_2 that $E \cap Y$ r-cuts, and $i_w = 1$ if E r-cuts w and 0 otherwise. Moreover, the fact whether E r-cuts w can be determined only by considering the A_R-representative of E, in particular by whether it is C_i-empty for at most one i.

We enumerate all pairs $(E_1, E_2) \in \mathcal{E}_1 \times \mathcal{E}_2$. If E_1 is compatible with E_2, then by Lemma 7.63, for any E_X and E_Y such that E_1 is a X_{R_X}-representative of E_X and E_2 is a Y_{R_Y}-representative of E_Y it holds that $E = E_X \cup E_Y$ is an A-restricted improvement embedding such that $E \cap X = E_X$, $E \cap Y = E_Y$, and the composition E_R of E_1 and E_2 is the A_R-representative of E. Let us fix such E_X and E_Y so that they are globally represented by $(\mathcal{E}_1, \mathcal{I}_1, \mathcal{C}_1)$ and $(\mathcal{E}_2, \mathcal{I}_2, \mathcal{C}_2)$, respectively. If $E_R \notin \mathcal{E}$, or $E_R \in \mathcal{E}$ but $\mathcal{I}(E_R) > \mathcal{I}_1(E_1) + \mathcal{I}_2(E_2) + i_w$, we insert E_R to $\mathcal{E}$ and set $\mathcal{I}(E_R) \leftarrow \mathcal{I}_1(E_1) + \mathcal{I}_2(E_2) + i_w$, $\mathcal{C}(E_R) \leftarrow \mathcal{C}_1(E_1) \cup \mathcal{C}_2(E_2)$ and $\mathcal{L}(E_R) \leftarrow (E_1, E_2)$. Now $(\mathcal{E}, \mathcal{I}, \mathcal{C})$ globally represents E.

The fact that for all A-restricted improvement embeddings E we actually considered a pair $(E_1, E_2) \in \mathcal{E}_1 \times \mathcal{E}_2$ so that E_1 is a X_{R_X}-representative of $E \cap X$ and E_2 is a Y_{R_Y}-representative of $E \cap Y$ follows from the definition of r-table and Lemma 7.56, i.e., the fact that $E \cap X$ is an X-restricted improvement embedding and $E \cap Y$ is an Y-restricted improvement embedding. □

Now the $\mathsf{Init}(\mathcal{T}, r)$ operation can be implemented in $2^{2^{\mathcal{O}(k)}} \cdot |\mathcal{T}|$ time by constructing a linked r-table of each node in the order from leaves to root with $|\mathcal{T}|$ applications of Lemma 7.75. For the $\mathsf{Move}(vw)$ operation, observe that the linked r-table of a node $x \in V(\mathcal{T})$ depends only on the r-subtree of x, and therefore by Lemma 7.29 it suffices to recompute only the linked r-table of v when using $\mathsf{Move}(vw)$. Therefore $\mathsf{Move}(vw)$ can be implemented in $2^{2^{\mathcal{O}(k)}}$ time by a single application of Lemma 7.75. The operation $\mathsf{Width}()$ is implemented without r-tables. It amounts to finding an embedding of $G[\mathcal{R}[uv], \mathcal{R}[vu]]$ to R_ℓ, for the smallest possible ℓ, by brute-force in time $2^{2^{\mathcal{O}(k)}}$. The $\mathsf{Output}()$ operation also is straightforward as we just output the augmented rank decomposition we are maintaining.

It remains to implement $\mathsf{CanImprove}()$, $\mathsf{EditSet}()$, and $\mathsf{Improve}(R, (N_1, N_2, N_3))$. The following is the main lemma for them, providing the implementations of $\mathsf{CanImprove}()$ and $\mathsf{EditSet}()$ and setting the stage for $\mathsf{Improve}(R, (N_1, N_2, N_3))$.

Lemma 7.76. *Let $(\mathcal{T}, \mathcal{R})$ be an augmented rank decomposition, $uv = r \in E(\mathcal{T})$, and $W = \mathcal{T}[uv]$. For each node $w \in V(\mathcal{T})$, let $(\mathcal{E}_w, \mathcal{I}_w, \mathcal{R}_w, \mathcal{L}_w)$ be the linked r-table of w. There is an algorithm that returns $\perp$ if there is no split of W, and otherwise returns a tuple $(R, N_1, N_2, N_3, C_1^R, C_2^R, C_3^R)$, where (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$, R is the edit set of (r, C_1, C_2, C_3), (N_1, N_2, N_3) is the neighborhood partition of R, and for each $i \in [3]$, C_i^R is a minimal representative of C_i. The algorithm runs in time $2^{2^{\mathcal{O}(k)}}(|R| + 1)$.*

Proof. Denote $X = W = \mathcal{T}_r[u]$, $Y = \overline{W} = \mathcal{T}_r[v]$, let $R_X = \mathcal{R}_r[u]$ and $R_Y = \mathcal{R}_r[v]$.

We iterate over all pairs $(E_u, E_v) \in \mathcal{E}_u \times \mathcal{E}_v$ satisfying the conditions in Lemma 7.68 and determine the sum-width of a corresponding split as in Lemma 7.68. Also, the minimum number of nodes of $\mathcal{T}$ r-cut by a split corresponding to (E_u, E_v) can be determined as $\mathcal{I}_u(E_u) + \mathcal{I}_v(E_v)$. If no such pair is found we return $\perp$. Otherwise we find a pair (E_u, E_v) so that E_u is the X_{R_X}-representative of an X-restricted improvement embedding E_X that is globally represented by $(\mathcal{E}_u, \mathcal{I}_u, \mathcal{C}_u)$ and whose tripartition $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$ is, E_v is the Y_{R_Y}-representative of an Y-restricted improvement embedding E_Y that is globally represented by $(\mathcal{E}_v, \mathcal{I}_v, \mathcal{C}_v)$ and whose tripartition $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$ is, and (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$. As $|\mathcal{E}_u| \cdot |\mathcal{E}_v| = 2^{2^{\mathcal{O}(k)}}$ and each pair can be checked in time $2^{2^{\mathcal{O}(k)}}$ (root-compatibility by Lemma 7.65), this phase has running time $2^{2^{\mathcal{O}(k)}}$.

Let $(f_1^C, f_2^C, f_3^C) = \mathcal{C}_u(E_u) \cup \mathcal{C}_v(E_v)$. Now f_i^C is a concrete representative of an embedding of $G[(X \cap C_i) \cup (Y \cap C_i), (X \setminus C_i) \cup (Y \setminus C_i)] = G[C_i, \overline{C_i}]$ to R_{k_i}, and therefore by Lemma 7.44 we obtain a representative of $(C_i, \overline{C_i})$ of size at most $2 \cdot 2^{k_i}$ from f_i^C. We compute a minimal representative C_i^R of C_i in time $2^{\mathcal{O}(k)}$ by Lemma 7.33

We compute the edit set and the neighborhood partition with a BFS-type algorithm that maintains a queue Q containing pairs (w, E_w), where $w \in V(\mathcal{T})$, with the invariant that if w is in the r-subtree of u, then E_w is the $\mathcal{T}_r[w]_{\mathcal{R}_r[w]}$-representative of $E_X \cap \mathcal{T}_r[w]$ and if w is in the r-subtree of v, then E_w is the $\mathcal{T}_r[w]_{\mathcal{R}_r[w]}$-representative of $E_Y \cap \mathcal{T}_r[w]$. We start by inserting the pairs (u, E_u) and (v, E_v) to Q. Then, we iteratively pop a pair (w, E_w) from the queue. If there is i such that E_w is C_j-empty for all $j \neq i$, then it holds that $\mathcal{T}_r[w] \subseteq C_i$, and therefore we insert w to N_i. Otherwise, we insert w to R, let w_1 and w_2 be the r-children of w, and let $(E_1, E_2) = \mathcal{L}_w(E_w)$. We insert (E_1, w_1) and (E_2, w_2) into Q. This maintains the invariant by the definition of a linked r-table.

For each node $w \in R \cup N_1 \cup N_2 \cup N_3$ we access tables indexed by E_w and determine C_i-emptiness, so the work is bounded by $|R \cup N_1 \cup N_2 \cup N_3| \cdot 2^{2^{\mathcal{O}(k)}} = |R| \cdot 2^{2^{\mathcal{O}(k)}}$. □

What is left is the $\mathsf{Improve}(R, (N_1, N_2, N_3))$ operation. Computing the improvement $\mathcal{T}'$ of $\mathcal{T}$ itself is a direct application of Lemma 7.23 and computing the linked r-tables of the new nodes is done by $|R|$ applications of Lemma 7.75, but in order to compute the linked r-tables we must first make $\mathcal{T}'$ augmented, that is, for each new edge uv compute a minimal representative of $(\mathcal{T}'[uv], \mathcal{T}'[vu])$. We use the minimal representatives of C_1, C_2, and C_3 returned by the algorithm of Lemma 7.76 for this.

Lemma 7.77. *Let $\mathcal{T}$ be a rank decomposition, $r \in E(\mathcal{T})$, and (r, C_1, C_2, C_3) a minimum split of $\mathcal{T}$. Let w be a non-leaf node of $\mathcal{T}$ and let w_1 and w_2 be the r-children of w. Let i,j, and ℓ be such that $\{i, j, \ell\} = [3]$. Given minimal representatives of $\mathcal{T}_r[w_1] \cap C_i$,*

$\mathcal{T}_r[w_2] \cap C_i$, $\overline{\mathcal{T}_r[w]}$, C_j, and C_ℓ, a minimal representative of $(\mathcal{T}_r[w] \cap C_i, \overline{\mathcal{T}_r[w] \cap C_i})$ can be computed in $2^{\mathcal{O}(k)}$ time.

Proof. As (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$, each of the given minimal representatives has size at most 2^k.

It holds that

$$\mathcal{T}_r[w] \cap C_i = (\mathcal{T}_r[w_1] \cap C_i) \cup (\mathcal{T}_r[w_2] \cap C_i),$$

so by Lemma 7.34 we obtain a representative of $\mathcal{T}_r[w] \cap C_i$ as the union of the given minimal representatives of $\mathcal{T}_r[w_1] \cap C_i$ and $\mathcal{T}_r[w_2] \cap C_i$. It also holds that

$$\overline{\mathcal{T}_r[w] \cap C_i} = \overline{\mathcal{T}_r[w]} \cup \overline{C_i} = \overline{\mathcal{T}_r[w]} \cup C_j \cup C_\ell,$$

so again by Lemma 7.34 we obtain a representative of $\overline{\mathcal{T}_r[w] \cap C_i}$ as the union of the given minimal representatives of $\overline{\mathcal{T}_r[w]}$, C_j, and C_ℓ. Now we have a representative of $(\mathcal{T}_r[w] \cap C_i, \overline{\mathcal{T}_r[w] \cap C_i})$ of size $2^{\mathcal{O}(k)}$, so we can use Lemma 7.33 to compute a minimal representative in time $2^{\mathcal{O}(k)}$. □

In particular, a minimal representative of $\overline{\mathcal{T}_r[w]}$ is available in $(\mathcal{T}, \mathcal{R})$ as $\mathcal{R}[pw]$, where p is the r-parent of w, and minimal representatives of $\mathcal{T}_r[w_1] \cap C_i$ and $\mathcal{T}_r[w_2] \cap C_i$ are available by doing the construction in an order towards the root r.

This completes the description of the improvement data structure for augmented rank decompositions and thus also the proof of Lemma 7.41.

7.5 Approximating branchwidth of graphs

In this section we prove the following theorem.

Theorem 1.7. *There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{\mathcal{O}(k)}n$ either outputs a branch decomposition of G of width at most $2k$ or determines that the branchwidth of G is larger than k.*

For branchwidth we do not need iterative compression as we can use the algorithm of Theorem 1.1 and the connection between branchwidth and treewidth [Robertson and Seymour, 1991] (presented as Lemma 2.18 in this thesis) to obtain a branch decomposition of G of width at most $3k$ in $2^{\mathcal{O}(k)}n$ time.

The following is the main lemma for proving Theorem 1.7, and this section is devoted to its proof.

Lemma 7.78. *There is a improvement data structure for branch decompositions of graphs with running time $t(k) = 2^{\mathcal{O}(k)}$.*

Combined with Theorem 7.32 and the aforementioned connections to treewidth, Lemma 7.78 implies Theorem 1.7.

The remaining part of this section is organized as follows. In Subsection 7.5.1 we define augmented branch decompositions, in Subsection 7.5.2 we introduce the objects manipulated in our dynamic programming and prove some properties of them. Then in Subsection 7.5.3 we give the improvement data structure for branch decompositions using this dynamic programming.

7.5.1 Augmented branch decompositions

In our improvement data structure we maintain an augmented branch decomposition. An augmented branch decompositions stores the *border description* of each set $\mathcal{T}[uv]$.

Definition 7.79 (Border description)**.** *Let $X \subseteq E(G)$. The border description of X is the pair $(\delta(X), f)$, where $f : \delta(X) \to \mathbb{Z}_{\geq 0}$ is the function so that $f(v)$ is the number of edges in X incident to v.*

An augmented branch decomposition is a branch decomposition $\mathcal{T}$ where for each edge $uv \in E(\mathcal{T})$ the border descriptions of $\mathcal{T}[uv]$ and $\mathcal{T}[vu]$ are stored. Note that an augmented branch decomposition can be represented in $\mathcal{O}(|\mathcal{T}| \cdot \mathsf{width}(\mathcal{T}))$ space.

The following lemma leads to an algorithm for computing the border descriptions. We assume that our representation of the input graph G supports reporting the degree of a given vertex in $\mathcal{O}(1)$ time.

Lemma 7.80. *Let X, Y be disjoint subsets of $E(G)$. Given the border descriptions of X and Y, the border description of $X \cup Y$ can be computed in $\mathcal{O}(|\delta(X)| + |\delta(Y)|)$ time.*

Proof. Let $(\delta(X), f)$ be the border description of X and $(\delta(Y), g)$ the border description of Y. It holds that $\delta(X \cup Y) \subseteq \delta(X) \cup \delta(Y)$, where $(\delta(X) \cup \delta(Y)) \setminus \delta(X \cup Y)$ can be identified as the vertices v for which $f(v) + g(v)$ is the degree of v. Now $(\delta(X \cup Y), f + g)$ is the border description of $X \cup Y$. □

Because $\delta(\overline{X}) = \delta(X)$ and the number of edges in $\overline{X}$ incident to v is the degree of v minus the number of edges in X incident to v, the border description of $\overline{X}$ can be computed from the border description of X in $\mathcal{O}(|\delta(X)|)$ time. It follows that given a branch decomposition $\mathcal{T}$, a corresponding augmented branch decomposition can be computed in $\mathcal{O}(|\mathcal{T}| \cdot \mathsf{width}(\mathcal{T}))$ time by using Lemma 7.80 $\mathcal{O}(|\mathcal{T}|)$ times.

7.5.2 Borders of tripartitions

We define the partial solutions stored in dynamic programming tables.

Definition 7.81 (Border of a tripartition)**.** *Let $A \subseteq E(G)$ and (C_1, C_2, C_3) a tripartition of A. The border of (C_1, C_2, C_3) is the 9-tuple*

$$(R_1, R_2, R_3, r_1, r_2, r_3, k_1, k_2, k_3),$$

where for each $i \in [3]$ it holds that $R_i = \delta(C_i) \cap \delta(A)$, $r_i = 0$ if $C_i = \emptyset$ and otherwise $r_i = 1$, and k_i is the number of vertices $v \in V(G) \setminus \delta(A)$ such that there exists an edge $e_1 \in C_i$ incident to v and an edge $e_2 \in A \setminus C_i$ incident to v, i.e., $k_i = |\delta(C_i) \cap \delta(A \setminus C_i) \setminus \delta(A)|$.

We call a border of tripartition k-bounded if for each i it holds that $k_i \leq k$. Note that if $|\delta(A)| \leq k$, then the number of k-bounded borders of tripartitions of A is $\leq (2^k)^3 2^3 k^3 = 2^{\mathcal{O}(k)}$, and each of them can be represented in $\mathcal{O}(k)$ space.

We define the *composition* of borders of tripartitions to combine partial solutions.

Definition 7.82 (Composition)**.** *Let X and Y be disjoint subsets of $E(G)$ and $A = X \cup Y$. Let $R_X = (R_1^X, R_2^X, R_3^X, r_1^X, r_2^X, r_3^X, k_1^X, k_2^X, k_3^X)$ be the border of a tripartition of X and $R_Y = (R_1^Y, R_2^Y, R_3^Y, r_1^Y, r_2^Y, r_3^Y, k_1^Y, k_2^Y, k_3^Y)$ the border of a tripartition of Y. Denote $F = (\delta(X) \cup \delta(Y)) \setminus \delta(A)$. The composition of R_Y and R_Y is the 9-tuple $(R_1, R_2, R_3, r_1, r_2, r_3, k_1, k_2, k_3)$, where for each $i \in [3]$,*

1. *$R_i = \delta(A) \cap (R_i^X \cup R_i^Y)$,*
2. *$r_i = \max(r_i^X, r_i^Y)$, and*
3. *$k_i = k_i^X + k_i^Y + |F \cap (R_i^X \cup R_i^Y) \cap (R_j^X \cup R_l^X \cup R_j^Y \cup R_l^Y)|$, where $\{i, j, l\} = [3]$.*

Note that when the sets $\delta(X)$, $\delta(Y)$, and $\delta(A)$ are given and have size $\leq k$, the composition can be computed in $\mathcal{O}(k)$ time.

Next we prove that the composition operation really combines partial solutions as expected.

Lemma 7.83. *Let X and Y be disjoint subsets of $E(G)$ and $A = X \cup Y$. If R_X is the border of a tripartition (C_1^X, C_2^X, C_3^X) and R_Y is the border of a tripartition (C_1^Y, C_2^Y, C_3^Y), then the composition of R_X and R_Y is the border of the tripartition $(C_1^X \cup C_1^Y, C_2^X \cup C_2^Y, C_3^X \cup C_3^Y)$.*

Proof. Let V^X be the set of vertices incident to X and V^Y the set of vertices incident to Y. Let also $F = (\delta(X) \cup \delta(Y)) \setminus \delta(A)$. Let $i \in [3]$ and let j and l be such that $\{i, j, l\} = [3]$. It holds that

$$\delta(C_i^X \cup C_i^Y) = (\delta(C_i^X) \cup \delta(C_i^Y)) \cap (\delta(C_j^X) \cup \delta(C_l^X) \cup \delta(C_j^Y) \cup \delta(C_l^Y) \cup \delta(\overline{A})).$$

Therefore by observing that $\delta(\overline{A}) = \delta(A)$, $\delta(A) \cap V^X \subseteq \delta(X)$, and $\delta(A) \cap V^Y \subseteq \delta(Y)$ we get

$$\delta(C_i^X \cup C_i^Y) \cap \delta(A) = (\delta(C_i^X) \cup \delta(C_i^Y)) \cap \delta(A) = (R_i^X \cup R_i^Y) \cap \delta(A),$$

so the sets R_1, R_2, and R_3 in the composition are correct.

By observing that $V^X \cap F \subseteq \delta(X)$ and $V^Y \cap F \subseteq \delta(Y)$, we have that

$$\begin{aligned}\delta(C_i^X \cup C_i^Y) \cap F =& F \cap (\delta(C_i^X) \cup \delta(C_i^Y)) \cap (\delta(C_j^X) \cup \delta(C_l^X) \cup \delta(C_j^Y) \cup \delta(C_l^Y)) \\ =& F \cap (R_i^X \cup R_i^Y) \cap (R_j^X \cup R_l^X \cup R_j^Y \cup R_l^Y).\end{aligned}$$

Since $V^X \setminus (F \cup \delta(A)) = V^X \setminus \delta(X)$ and $V^Y \setminus (F \cup \delta(A)) = V^Y \setminus \delta(Y)$ are disjoint, we get that

$$\begin{aligned}&|\delta(C_i^X \cup C_i^Y) \setminus (F \cup \delta(A))| \\ &\quad = |(\delta(C_i^X) \cap (\delta(C_j^X) \cup \delta(C_l^X)) \setminus \delta(X)| + |(\delta(C_i^Y) \cap (\delta(C_j^Y) \cup \delta(C_l^Y)) \setminus \delta(Y)| \\ &\quad = |\delta(C_i^X) \cap \delta(X \setminus C_i^X) \setminus \delta(X)| + |\delta(C_i^Y) \cap \delta(Y \setminus C_i^Y) \setminus \delta(Y)|.\end{aligned}$$

Hence the numbers k_1, k_2, and k_3 in the composition are correct. The numbers r_1, r_2, and r_3 are correct by observing that $C_i^X \cup C_i^Y$ is empty if and only if both C_i^X and C_i^Y are empty. □

The next lemma gives the method for determining if there exists a split of W based on borders of tripartitions.

Lemma 7.84. *Let $\mathcal{T}$ be a branch decomposition, $uv = r \in E(\mathcal{T})$, and $W = \mathcal{T}[uv]$. Denote $X = W$ and $Y = \overline{W}$. There exists a split (C_1, C_2, C_3) of W with $|\delta(C_i)| = k_i$ for each $i \in [3]$ if and only if there exists R_X and R_Y such that*

1. $R_X = (R_1^X, R_2^X, R_3^X, r_1^X, r_2^X, r_3^X, k_1^X, k_2^X, k_3^X)$ *is the border of the tripartition* $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$,

2. $R_Y = (R_1^Y, R_2^Y, R_3^Y, r_1^Y, r_2^Y, r_3^Y, k_1^Y, k_2^Y, k_3^Y)$ *is the border of the tripartition* $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$,

3. *the composition of* R_X *and* R_Y *is* $(\emptyset, \emptyset, \emptyset, r_1, r_2, r_3, k_1, k_2, k_3)$, *where* $k_i < |\delta(W)|$ *for each* i, *and*

4. *for each* i *it holds that* $k_i^X + |R_i^X| < |\delta(W)|$ *and* $k_i^Y + |R_i^Y| < |\delta(W)|$.

Proof. Suppose that such R_X and R_Y exists. By Lemma 7.83 and the fact that $\delta(E(G)) = \emptyset$, $(\emptyset, \emptyset, \emptyset, r_1, r_2, r_3, k_1, k_2, k_3)$ is the border of (C_1, C_2, C_3). By the definition of border we have that $k_i = |\delta(C_i)|$. It remains to prove that $|\delta(C_i \cap W)| = k_i^X + |R_i^X|$ and that $|\delta(C_i \cap \overline{W})| = k_i^Y + |R_i^Y|$ for each i. We have that

$$\begin{aligned}
\delta(W \cap C_i) =& (\delta(W \cap C_i) \cap \delta(\overline{W})) \cup (\delta(W \cap C_i) \cap \delta(W \setminus C_i)) \\
=& (\delta(X \cap C_i) \cap \delta(X)) \cup (\delta(X \cap C_i) \cap \delta(X \setminus C_i)) \\
=& (\delta(X \cap C_i) \cap \delta(X)) \cup (\delta(X \cap C_i) \cap \delta(X \setminus C_i) \cap \delta(X)) \\
& \cup (\delta(X \cap C_i) \cap \delta(X \setminus C_i) \setminus \delta(X)) \\
=& (\delta(X \cap C_i) \cap \delta(X)) \cup (\delta(X \cap C_i) \cap \delta(X \setminus C_i) \setminus \delta(X)).
\end{aligned}$$

Therefore, by the definition of border,

$$\begin{aligned}
&|(\delta(X \cap C_i) \cap \delta(X)) \cup (\delta(X \cap C_i) \cap \delta(X \setminus C_i) \setminus \delta(X))| \\
&\qquad = |\delta(X \cap C_i) \cap \delta(X)| + |\delta(X \cap C_i) \cap \delta(X \setminus C_i) \setminus \delta(X)| = |R_i^X| + k_i^X.
\end{aligned}$$

The other case is symmetric.

The above is the proof of the if direction. The proof for the only if direction is the same but starting from supposing that such split (C_1, C_2, C_3) of W exists and letting R_X be the border of $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$ and R_Y the border of $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$. □

7.5.3 Improvement data structure for graph branch decompositions

In the improvement data structure for branch decompositions we maintain an augmented branch decomposition $\mathcal{T}$ rooted at edge $r \in E(\mathcal{T})$ and a dynamic programming table that stores for each node $w \in V(\mathcal{T})$ all k-bounded borders of tripartitions of $\mathcal{T}_r[w]$ and

information about how many nodes in the r-subtree of w they r-cut. We call this dynamic programming table an r-table, to signify that it is directed towards r.

In this subsection we always assume that k is an integer such that $\mathsf{width}(\mathcal{T}) \leq k$, and therefore we only care about k-bounded borders of tripartitions. Next we formally define the contents of an r-table.

Definition 7.85 (r-table)**.** *Let $\mathcal{T}$ be a branch decomposition, $r \in E(\mathcal{T})$ an edge of $\mathcal{T}$, $w \in V(\mathcal{T})$, and $A = \mathcal{T}_r[w]$. The r-table of w is the pair $(\mathcal{B}, \mathcal{I})$, where $\mathcal{B}$ is the set of all k-bounded borders of tripartitions of A, and $\mathcal{I}$ is a function mapping each $R \in \mathcal{B}$ to the least integer i such that there exists a tripartition of A whose border R is and that r-cuts i nodes of the r-subtree of w.*

As there are $2^{\mathcal{O}(k)}$ k-bounded tripartitions of $\mathcal{T}_r[w]$, the r-table of w can be represented in $2^{\mathcal{O}(k)}$ space.

Lemma 7.86. *Let $\mathcal{T}$ be an augmented branch decomposition, $r \in E(\mathcal{T})$, and w a non-leaf node of $\mathcal{T}$ with r-children w_1 and w_2. Given the r-tables of w_1 and w_2, the r-table of w can be constructed in $2^{\mathcal{O}(k)}$ time.*

Proof. Denote $A = \mathcal{T}_r[w]$, $X = \mathcal{T}_r[w_1]$, and $Y = \mathcal{T}_r[w_2]$. Let $(\mathcal{B}_{w_1}, \mathcal{I}_{w_1})$ and $(\mathcal{B}_{w_2}, \mathcal{I}_{w_2})$ be the r-tables of w_1 and w_2.

We construct the r-table $(\mathcal{B}, \mathcal{I})$ of w as follows. We iterate over all pairs

$$(R_X, R_Y) \in \mathcal{B}_{w_1} \times \mathcal{B}_{w_2}$$

and let $\mathcal{B}$ be the set of compositions of those pairs. This correctly constructs $\mathcal{B}$ by Lemma 7.83 and the observation that if (C_1, C_2, C_3) is a tripartition of A whose border is k-bounded, then $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$ is a tripartition of X whose border is k-bounded and $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$ is a tripartition of Y whose border is k-bounded. For each $R \in \mathcal{B}$ we set $\mathcal{I}(R)$ as the minimum value of $\mathcal{I}_{w_1}(R_X) + \mathcal{I}_{w_2}(R_Y) + i_w$ over such pairs R_X, R_Y whose composition R is, where $i_w = 1$ if R is of form $(\ldots, r_1, r_2, r_3, \ldots)$ where $r_1 + r_2 + r_3 \geq 2$, and $i_w = 0$ otherwise. This correctly constructs $\mathcal{I}$ by the observations that (C_1, C_2, C_3) r-cuts a node w' in the r-subtree of w_1 if and only if $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$ r-cuts w', (C_1, C_2, C_3) r-cuts a node w' in the r-subtree of w_2 if and only if $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$ r-cuts w', and that (C_1, C_2, C_3) r-cuts w if and only if $r_1 + r_2 + r_3 \geq 2$.

As $|\mathcal{B}_{w_1}||\mathcal{B}_{w_2}| = 2^{\mathcal{O}(k)}$, the running time is $2^{\mathcal{O}(k)}$. □

Now the $\mathsf{Init}(\mathcal{T}, r)$ operation can be implemented in $2^{\mathcal{O}(k)} \cdot |\mathcal{T}|$ time by first making $\mathcal{T}$ augmented by $2 \cdot |\mathcal{T}|$ applications of Lemma 7.80 and then constructing the k-bounded r-tables of all nodes in the order from leaves to root by $|\mathcal{T}|$ applications of Lemma 7.86. For the $\mathsf{Move}(vw)$ operation, we note the r-table of a node $x \in V(\mathcal{T})$ depends only on the r-subtree of x, and therefore by Lemma 7.29 it suffices to recompute only the r-table of the node v when using $\mathsf{Move}(vw)$. Therefore $\mathsf{Move}(vw)$ can be implemented in $2^{\mathcal{O}(k)}$ time by a single application of Lemma 7.86. The $\mathsf{Width}()$ operation returns $|\delta(\mathcal{T}[uv])|$, which is available because $\mathcal{T}$ is augmented. The $\mathsf{Output}()$ operation is also straightforward as it just returns the branch decomposition we are maintaining.

The following lemma implements the operations $\mathsf{CanImprove}()$ and $\mathsf{EditSet}()$ based on Lemma 7.84.

Lemma 7.87. *Let $\mathcal{T}$ be a branch decomposition, $uv = r \in E(\mathcal{T})$, and $W = \mathcal{T}[uv]$. For each node $w \in V(\mathcal{T})$ let $(\mathcal{B}_w, \mathcal{I}_w)$ be the r-table of w. There is an algorithm that returns $\perp$ if there is no split of W, and otherwise returns a tuple (R, N_1, N_2, N_3), where (r, C_1, C_2, C_3) is a minimum split of $\mathcal{T}$, R is the edit set of (r, C_1, C_2, C_3) and (N_1, N_2, N_3) is the neighborhood partition of R. The algorithm runs in time $2^{\mathcal{O}(k)}(|R| + 1)$.*

Proof. Denote $X = W = \mathcal{T}_r[u]$, $Y = \overline{W} = \mathcal{T}_r[v]$. We iterate over all pairs $(R_X, R_Y) \in \mathcal{B}_u \times \mathcal{B}_v$, using Lemma 7.84 to either conclude that there exists no split of W or to find a pair (R_X, R_Y) such that there is a minimum split (r, C_1, C_2, C_3) of $\mathcal{T}$ so that R_X is the border of $(C_1 \cap X, C_2 \cap X, C_3 \cap X)$, R_Y is the border of $(C_1 \cap Y, C_2 \cap Y, C_3 \cap Y)$, and the number of nodes of $\mathcal{T}$ r-cut by (C_1, C_2, C_3) is $\mathcal{I}_u(R_X) + \mathcal{I}_v(R_Y)$. In time $2^{\mathcal{O}(k)}$ we either find such a pair or conclude that there is no split of W.

We compute the edit set and the neighborhood partition with a BFS-type algorithm that maintains a queue Q containing pairs (w, R_w), where $w \in V(\mathcal{T})$, with the invariant that there exists a minimum split (r, C_1, C_2, C_3) of $\mathcal{T}$ so that for all (w, R_w) that appear in the queue, R_w is the border of $(C_1 \cap \mathcal{T}_r[w], C_2 \cap \mathcal{T}_r[w], C_3 \cap \mathcal{T}_r[w])$. We start by inserting the pairs (u, R_X) and (v, R_Y) to Q. We iteratively pop a pair (w, R_w) from the queue. Denote $R_w = (\ldots, r_1, r_2, r_3, \ldots)$. If there is i such that $r_j = 0$ for both $j \neq i$, then it holds that $\mathcal{T}_r[w] \subseteq C_i$, and therefore we insert w to N_i. Otherwise, we insert w to R, let w_1 and w_2 be the r-children of w, and find a pair $(R_{w_1}, R_{w_2}) \in \mathcal{B}_{w_1} \times \mathcal{B}_{w_2}$ so that the composition of R_{w_1} and R_{w_2} is R_w and $\mathcal{I}_{w_1}(R_{w_1}) + \mathcal{I}_{w_2}(R_{w_2}) + 1 = \mathcal{I}_w(R_w)$. This kind of pair exists and maintains the invariant by the definition of r-table and Lemma 7.83.

For each node $w \in R \cup N_1 \cup N_2 \cup N_3$ we iterate over $\mathcal{B}_{w_1} \times \mathcal{B}_{w_2}$ and access some tables, so the total amount of work is bounded by $2^{\mathcal{O}(k)} \cdot |R \cup N_1 \cup N_2 \cup N_3| = 2^{\mathcal{O}(k)} \cdot |R|$. □

What is left is the $\mathsf{Improve}(R, (N_1, N_2, N_3))$ operation. Computing the improvement of $\mathcal{T}$ is done in $\mathcal{O}(|R|)$ time by applying Lemma 7.23. Computing the border descriptions of the newly inserted edges can be done in a bottom-up fashion starting from $N_1 \cup N_2 \cup N_3$ by $2|R|$ applications of Lemma 7.80. Then, computing the r-tables of the newly inserted nodes can be done also in a similar fashion in $2^{\mathcal{O}(k)} \cdot |R|$ time by $|R|$ applications of Lemma 7.86.

This completes the description of the improvement data structure for branch decompositions of graphs and thus also the proof of Lemma 7.78.

Chapter 8

Conclusions

In this thesis, we introduced the *local improvement technique* for computing graph width parameters. We applied this technique to a range of problems about FPT algorithms for computing graph width parameters in various settings, from approximation to exact, from static to dynamic, and from treewidth to rankwidth, solving several open problems from the literature. Perhaps it could be argued that the local improvement technique represents the third major technique for designing FPT algorithms for graph width parameters, in addition to the top-down construction of Robertson and Seymour, and the dynamic programming using typical sequences by Bodlaender, Kloks, Lagergren, and Arnborg.

In this chapter, we first summarize our contributions, then discuss follow-up works, and then future directions and open problems.

8.1 Summary of contributions

We briefly summarize the contributions of Chapters 4 to 7 and provide additional remarks.

Fast 2-approximation algorithm for treewidth

In Chapter 4, we gave a $\mathcal{O}(2^{10.8k}n)$ time 2-approximation algorithm for treewidth. This is an improvement over the 5-approximation algorithm by Bodlaender et al. [2016a], both in terms of the approximation ratio and the running time as a function of k. Bodlaender et al. did not explicitly analyze the factor $2^{\mathcal{O}(k)}$ in their algorithm's running time, nor did they attempt to optimize this factor in any way, but we note that their algorithm makes use of dynamic programming with running time $\Omega(9^w)$ on a tree decomposition

of width w, where an upper bound for w is $30k$, yielding a rough estimate of 2^{95k} for this factor. While our algorithm constitutes progress in improving the dependency on k, the worst-case bound of $2^{10.8k}$ is still far from practical. Nevertheless, we believe our techniques could be useful in practical implementations. In fact, the MSVS heuristic proposed by [Koster, 1999; Koster et al., 2002] already shares some similarities with our algorithm.

More important than the running time bounds is the new technique. We designed a tree decomposition improvement operation based on ideas from the proofs on lean tree decompositions from the graph theory literature, and introduced several additional techniques to improve the approximation ratio to 2 and make the resulting algorithm run in linear time. To the best of our knowledge, these ideas have not been previously used in the context of computing graph width parameters. As shown by Chapters 5 to 7 of this thesis, these ideas and their generalizations turned out to also be applicable to several other problems in the field.

Exact and $(1+\varepsilon)$-approximation algorithms for treewidth

In Chapter 5, we gave a $2^{\mathcal{O}(k^2)}n^4$ time algorithm for deciding if a given graph has treewidth at most k and computing the corresponding tree decomposition, and also a $k^{\mathcal{O}(k/\varepsilon)}n^4$ time $(1+\varepsilon)$-approximation algorithm for the same problem. The former algorithm answers the long-standing open question of whether there exists a $2^{o(k^3)}n^{\mathcal{O}(1)}$ time algorithm for treewidth, being the first improvement in the dependence on k in exact FPT algorithms for treewidth since the $2^{\mathcal{O}(k^3)}n^{\mathcal{O}(1)}$ time algorithms of Bodlaender and Kloks [1996], and Lagergren and Arnborg [1991].

For our algorithms, we introduced a problem called Subset Treewidth, extended the local improvement technique to show that algorithms for Subset Treewidth imply algorithms for treewidth, and finally gave branching algorithms for solving Subset Treewidth. These techniques are in contrast to previous FPT (and also XP) algorithms for computing treewidth exactly, all of which use dynamic programming.

In both of the algorithms of Chapter 5, the running time dependence n^4 on the number of vertices n is unusually high for FPT algorithms for treewidth. The bottleneck lies in the polynomial-time process of iteratively improving the torso tree decomposition $(X, \mathcal{T}_X)$ in Subsection 5.2.3. In particular, the running times in Theorems 1.2 and 1.3 could be stated in a tighter way as $2^{\mathcal{O}(k^2)}n^3 + k^{\mathcal{O}(1)}n^4$ and $k^{\mathcal{O}(k/\varepsilon)}n^3 + k^{\mathcal{O}(1)}n^4$, respectively.

The statement of Theorem 5.2 connecting treewidth and Subset Treewidth is technical because we paid attention to the factors polynomial in n, so let us here note that

Lemma 5.14 used together with iterative compression implies the following elegant connection between treewidth and Subset Treewidth.

Theorem 8.1. *For any function $f : \mathbb{N} \to \mathbb{N}$, there is an $f(k) \cdot n^{\mathcal{O}(1)}$ time algorithm for treewidth if and only if there is an $f(k) \cdot n^{\mathcal{O}(1)}$ time algorithm for Subset Treewidth.*

Here, the "if"-direction stems from iterative application of Lemma 5.14, inserting vertices into the graph one by one. The "only if"-direction follows from the fact that the statement of Subset Treewidth allows us to conclude that the treewidth of G is more than k, which implies that any algorithm for treewidth is trivially also an algorithm for Subset Treewidth.

The proof that algorithms for Subset Treewidth imply algorithms for treewidth significantly generalizes the techniques introduced in Chapter 4. We further made use of this generalization in the refinement operation of Chapter 6.

Dynamic treewidth

In Chapter 6 we gave a data structure for maintaining a tree decomposition of width at most $6k + 5$ of a dynamic graph G of treewidth at most k under edge insertions and deletions, with amortized update time $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$. Our data structure furthermore supports maintaining any dynamic programming scheme on the tree decomposition, with overhead depending on the running time of computing the state of a node based on the states of its children, which is for example $2^{\mathcal{O}(k)}$ if we wish to maintain the cardinality of the maximum independent set of G.

Our data structure is the first non-trivial solution to the dynamic treewidth problem that maintains a tree decomposition with width bounded by a function of an arbitrary fixed treewidth bound k. This addresses open problems asked by [Alman et al., 2020; Bodlaender, 1993; Chen et al., 2021; Dvořák et al., 2014; Majewski et al., 2023], although it is not clear whether we can count these problems as fully resolved, as the running time $2^{k^{\mathcal{O}(1)}\sqrt{\log n \log\log n}}$ is probably not optimal.

The main ingredient of our data structure is the refinement operation, which builds on the generalized version of the local improvement technique introduced in Chapter 5, combining it with the amortization ideas introduced in Chapter 4 and several other techniques from the literature of treewidth computing.

As we discussed in Subsection 3.3.3, treewidth is an important ingredient in many algorithms, so we hope that the dynamic treewidth data structure will be useful for

lifting these algorithms to the dynamic setting. Let us mention here a direct application about H-minor-containment for planar graphs H, and postpone further discussion about applications to the subsequent sections of this chapter.

Corollary 8.2. *Let H be a planar graph. There exists a data structure, that for an n-vertex dynamic graph G that is updated by edge insertions and deletions, maintains whether H is a minor of G. The amortized initialization time on an edgeless n-vertex graph G is $f(|V(H)|) \cdot n$, and the amortized updated time is $2^{f(|V(H)|)\cdot\sqrt{\log n \log\log n}}$, for some computable function f.*

Proof. By the Grid Minor Theorem of Robertson and Seymour [1986b] (discussed as Theorem 3.13 in Subsection 3.3.1), there exists an integer $f(|V(H)|)$ so that if the treewidth of G is more than $f(|V(H)|)$, then G contains H as a minor. Now, we initialize the dynamic treewidth data structure of Theorem 1.4 with the treewidth bound $f(|V(H)|)$ and a CMSO_2 sentence φ_H that is true in G if and only if G contains H as a minor. Now, whenever the data structure of Theorem 1.4 displays the marker "Treewidth too large", we know by the Grid Minor Theorem that H must be a minor of G. Otherwise, the data structure tells whether φ_H is true in G, which tells whether H is a minor of G. □

Fast 2-approximation algorithms for rankwidth and branchwidth

In Chapter 7 we gave a framework for designing FPT 2-approximation algorithms for instantiations of branchwidth of connectivity functions. The main applications of this framework are 2-approximation algorithms for rankwidth and branchwidth of graphs, with running times $2^{2^{\mathcal{O}(k)}}n^2$ and $2^{\mathcal{O}(k)}n$, respectively. The algorithm for rankwidth answers the open question of whether there exists a $g(k)$-approximation algorithm for rankwidth running in time $f(k) \cdot n^c$, for $c < 3$ and functions $g(k)$ and $f(k)$. This problem had been open since the work of Oum [2008b] and was explicitly asked by Oum [2017].

In particular, our algorithm for rankwidth has the major implication that now all CMSO_1-expressible problems can be solved in quadratic time on graphs of bounded rankwidth, even when no decomposition is given as an input, in contrast to the previous cubic time.

Our framework extends the local improvement technique from the setting of treewidth to the general setting of branchwidth of connectivity functions. We note that even though a concept similar to lean tree decompositions was introduced for this setting by Geelen et al. [2002], it appears that the definitions of Geelen et al. are too weak to obtain approximation algorithms. Therefore we designed our own adaptation of the local improvement technique to this setting.

8.2 Follow-up work

Before discussing future directions, we note that the author, along with coauthors, has already addressed some natural follow-up questions arising from the contributions of this thesis, particularly from Chapters 6 and 7. We mention two follow-up works not included in the thesis due to time constraints.

Dynamic Baker's scheme. Given the dynamic treewidth algorithm, it is natural to ask if it can be used to lift classical applications of treewidth to the dynamic setting. Together with Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski, we explored this question about the Baker's scheme (see Subsection 3.3.3) in [Korhonen et al., 2024]. We partly answered this question, giving a $(1-\varepsilon)$-approximation algorithm for maximum weight independent set in dynamic apex-minor-free graphs with update time $f(\varepsilon) \cdot n^{o(1)}$, where $f(\varepsilon)$ is doubly-exponential in $\mathcal{O}(1/\varepsilon^2)$, and also a similar algorithm for weighted dominating set, but with an extra restriction of bounded degree. Ultimately, we did not use the dynamic treewidth data structure in this algorithm, but instead used an ad-hoc method for dealing with tree decompositions in the dynamic setting tailored for the Baker's scheme.

Dynamic rankwidth. With Marek Sokołowski, we recently generalized the dynamic treewidth data structure of Chapter 6 to rankwidth [Korhonen and Sokołowski, 2024]. We obtained a similar result as in Theorem 1.4 but for rankwidth and CMSO_1 instead of treewidth and CMSO_2. This dynamic rankwidth algorithm builds upon the dynamic treewidth algorithm, and also further develops several techniques introduced in Chapter 7. By using the dynamic rankwidth algorithm and additional ideas, we were able to also give a $f(k) \cdot n \cdot 2^{\sqrt{\log n} \log \log n} + \mathcal{O}(m)$ time algorithm for computing optimal rank decompositions, for some computable function f. This algorithm also outputs a $(2^{k+1}-1)$-expression for cliquewidth within the same running time, implying that CMSO_1-expressible problems can be solved with that running time on graphs of cliquewidth k.

8.3 Future directions and open problems

Finally, let us discuss future directions and pose some open problems.

Further applications of local improvement

Let us start by discussing further potential settings in which our local improvement technique could be applied.

The first obvious class of applications is width parameters that can be expressed as special cases of branchwidth of connectivity functions. In Chapter 7 we applied our framework to rankwidth and branchwidth of graphs. However, parameters such as carving width, matroid branchwidth, and hypergraph branchwidth are also expressible within this framework (see e.g. [Jeong et al., 2021]). The main reason for not pursuing these directions further is that it is not clear if there are applications in these settings that make significant improvements compared to already existing algorithms.

The techniques introduced in [Korhonen and Sokołowski, 2024] suggest that an analogue of the Subset Treewidth problem also exists in the setting of branchwidth of connectivity functions. It could be interesting to explore whether this would be useful for faster exact FPT algorithms in this setting. In particular, could we design an exact FPT algorithm for computing the branchwidth of graphs by using an approach similar to the one for treewidth in Chapter 5? This is interesting not only for the sake of computing branchwidth but the different formalism of branchwidth could help to gain a deeper understanding of these techniques.

It appears that the local improvement technique requires some "submodularity properties" of decompositions to work, ruling out its application to parameters like α-treewidth [Yolov, 2018] and fractional hypertreewidth [Grohe and Marx, 2014], at least at first glance. A width parameter that does not fall into the framework of branchwidth of connectivity functions but enjoys these properties is the tree-cut width [Wollan, 2015], and indeed there exists an analogue of lean tree decompositions in that setting [Giannopoulou et al., 2021].

Another direction is the path-like parameters, like pathwidth, cutwidth, and the linear analogue of branchwidth of connectivity functions. In these settings, analogues of lean tree decompositions are known to exist [Kanté et al., 2023; Lagergren, 1998], but at least superficially it appears that techniques related to them are not applicable for approximation algorithms. As a concrete open problem, we ask the following.

Question 1. *Is there a constant-factor approximation algorithm for pathwidth with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$, where k is the pathwidth and n is the number of vertices?*

Computing treewidth

We then turn to the topic of computing treewidth. In Chapter 4 we made progress in the questions of what is the smallest approximation ratio with which treewidth can be approximated (1) in $2^{\mathcal{O}(k)}n$ time, or (2) in $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time. In Chapter 5 we addressed the questions of what is the asymptotically smallest function $f(k)$ so that treewidth can be (1) computed exactly in $f(k) \cdot n^{\mathcal{O}(1)}$ time, or (2) $(1+\varepsilon)$-approximated, for every fixed $\varepsilon > 0$, in $f(k) \cdot n^{\mathcal{O}(1)}$ time. While all of these questions remain open and would be very interesting to make further progress in, let us specifically highlight a case that is an important challenge in the author's opinion.

Question 2. *Is there a* 1.99*-approximation algorithm for treewidth, running in time* $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$*, where* k *is the treewidth and* n *the number of vertices?*

Possible routes for solving Question 2 could be an improved algorithm for Partitioned Subset Treewidth when the number of parts is constant, or an entirely new approach.

On the lower bound side, we are not aware of any lower bounds under the ETH for computing treewidth, apart from the lower bound that follows from the NP-hardness proof of Arnborg et al. [1987]. By tracing the chain of reductions in the proof, we observe in Appendix B of the arXiv version of Article 2 [Korhonen and Lokshtanov, 2022] that it implies the following lower bound under the ETH.

Proposition 8.3. *Assuming the ETH, there is no* $2^{o(\sqrt{n})}$ *time algorithm for computing treewidth.*

This of course implies also a $2^{o(\sqrt{k})}$ lower bound on the dependence on k, but nothing better since the graph produced by the reduction is co-bipartite. It would be very surprising if there were a subexponential time algorithm for treewidth, so on the lower bound side we ask the following question.

Question 3. *Does the ETH imply that there is no* $2^{o(n)}$ *time algorithm for computing treewidth?*

An easier version of Question 3 would be to prove a lower bound $2^{o(k)}$ for the dependence on the treewidth k in FPT algorithms. While we believe that $2^{\mathcal{O}(n)}$ should be the right running time in the non-parameterized setting, we do not conjecture where between $2^{\mathcal{O}(k)}$ and $2^{\mathcal{O}(k^2)}$ the right answer for the dependence on k in FPT algorithms should lie.

On the approximation side, the lower bound situation is even more dire, as even the NP-hardness of constant-factor approximation is only known conditional to the Small Set

Expansion conjecture [Raghavendra and Steurer, 2010; Wu et al., 2014]. In this setting, the possibility for subexponential time algorithms seems more realistic than for exact algorithms, so we ask the following question.

Question 4. *Is there a $2^{o(n)}$ time constant-factor approximation algorithm for treewidth?*

Question 4 is closely related to the question of approximating balanced separator in subexponential time, which is also open, see e.g. [Manurangsi and Trevisan, 2018].

Let us then discuss the running time dependence n^4 on the number of vertices n in the algorithms of Chapter 5, which is unusually high as we already noted. We believe that improving the dependence on n in these algorithms significantly is an interesting problem. Improving it below n^3 should require new techniques, and below n^2 could need a completely new approach.

It seems that although the Bodlaender-Kloks dynamic programming is more than 30 years old, we are not aware of arguments for why it could not simply be optimized to run in $2^{o(k^3)}n$ time. As a possible direction for arguing that Bodlaender-Kloks is the optimal "dynamic programming algorithm" for treewidth, we ask the following question about the communication complexity of computing treewidth.

Question 5. *Let $k \in \mathbb{Z}_{\geq 1}$, G be a graph, and (A, S, B) a separation of G of order $|S| \leq \mathcal{O}(k)$. Suppose Alice holds the graph $G[A \cup S]$ and Bob holds the graph $G[B \cup S]$, and both know the set S. How many bits of information do they need to exchange to know whether the treewidth of G is at most k?*

Note that the Bodlaender-Kloks dynamic programming gives an upper bound of $2^{\mathcal{O}(k^3)}$ for the number of bits needed, but on the other hand, the algorithm of Chapter 5 does not seem to imply an improved upper bound.

Dynamic treewidth

For dynamic treewidth, the main open problem is whether the running time of the data structure can be improved. We ask three questions with increasing difficulty about this.

Question 6. *Can the amortized update time of the dynamic treewidth data structure of Theorem 1.4 be improved to (1) $(\log n)^{f(k)}$, (2) $f(k) \cdot (\log n)^{\mathcal{O}(1)}$, or (3) $f(k) \cdot \log n$, for some function $f(k)$ depending on the treewidth bound k?*

We would be happy to answer these questions even with the width bound $6k + 5$ of the maintained tree decomposition in Theorem 1.4 weakened to any function $g(k)$ on the

treewidth bound k. In the author's opinion, Question 6 is among the most important open problems we ask here, due to its applications and fundamental nature. In the static setting, the complexity of computing treewidth for fixed k was resolved over 30 years ago by Bodlaender [1996]. Question 6 can be seen as asking for an analogue of Bodlaender's result in the dynamic setting.

We note that most of the techniques introduced in Chapter 6 appear to be suitable even for achieving an update time of form $f(k) \cdot (\log n)^{\mathcal{O}(1)}$. In particular, it could be that only a slightly more clever height-reduction scheme would be required to improve Theorem 1.4 to solve the cases (1) and (2) of Question 6.

Other questions related to improving Theorem 1.4 are whether the amortized bounds can be turned into worst-case bounds and whether the dependence on k could be improved. While these questions are interesting, it is maybe too early to focus on them at this point, before making progress on Question 6.

Let us then discuss potential applications of dynamic treewidth. We already mentioned that to produce potential applications, one can take an application of treewidth in the static setting, and ask whether it could be made dynamic, as we did in [Korhonen et al., 2024] with the Baker's scheme. As the area of dynamic FPT algorithms has recently received increased attention (see e.g. [Alman et al., 2020; Chen et al., 2021; Grez et al., 2022; Iwata and Oka, 2014; Olkowski et al., 2023]), we hope that dynamic treewidth would see multiple applications in that context. One concrete fundamental question is about dynamic planar k-disjoint paths.

Question 7. *Is there a dynamic data structure for the k-disjoint paths problem on dynamic planar graphs with (amortized) update time $f(k) \cdot n^{o(1)}$, where $f(k)$ is some function depending on the number of terminal pairs k.*

Applying dynamic treewidth to improve the running times of static FPT algorithms, like we did in [Korhonen and Sokołowski, 2024] with dynamic rankwidth, is another interesting direction. Even more interesting would be if there would be applications of dynamic treewidth outside of traditional FPT problems. For example, could replacing dynamic trees in some applications with dynamic graphs of treewidth, say $\mathcal{O}(\log \log n)$, be fruitful?

Computing rankwidth and cliquewidth

In Chapter 7 we made progress on the fundamental question of the complexity of CMSO_1-expressible problems on graphs of bounded rankwidth, and in [Korhonen and

Sokołowski, 2024] we further improved upon the result of Chapter 7. Clearly, the ultimate goal is to obtain a linear-time algorithm.

Question 8. *Is there an algorithm, that given an n-vertex m-edge graph G, integer k, and a* CMSO_1 *sentence φ, in time $f(k,\varphi)\cdot(n+m)$ either decides whether φ is true in G or concludes that the rankwidth of G is more than k, for some computable function f.*

We formulated Question 8 as about CMSO_1, as strictly speaking algorithms for computing rankwidth are not known to directly imply an answer for it. However, the approach for solving Question 8 should be to give a $f(k)\cdot(n+m)$ time $g(k)$-approximation algorithm for rankwidth, for some functions $f(k)$ and $g(k)$ depending on the rankwidth k.

In the case of treewidth, typical dynamic programming algorithms on tree decompositions of width k run in time $2^{\mathcal{O}(k)}n$, and we can obtain this running time parameterized by the treewidth k of a given graph even when we are not given a decomposition. For cliquewidth, typical dynamic programming algorithms also run in time $2^{\mathcal{O}(k)}n$ when given a k-expression, but we do not know how to obtain such running times parameterized by the cliquewidth k when not given a k-expression. As discussed before, Oum et al. [2014] obtained $2^{\mathcal{O}(k\log k)}n^{\mathcal{O}(1)}$ time algorithms parameterized by cliquewidth k. We ask if this can be improved.

Question 9. *Is there a $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithm for maximum independent set parameterized by cliquewidth k, when a k-expression is not given?*

Question 9 was also asked by Oum et al. [2014]. Another well-known open problem about cliquewidth, asked by Fellows et al. [2009], is whether computing cliquewidth exactly is FPT parameterized by cliquewidth, or even XP. This question appears to be open even for $g(k)$-approximation, for a subexponential function $g(k)$ depending on the cliquewidth k.

Bibliography

I. Adler, F. Dorn, F. V. Fomin, I. Sau, and D. M. Thilikos. Fast minor testing in planar graphs. *Algorithmica*, 64(1):69–84, 2012. URL https://doi.org/10.1007/s00453-011-9563-9.

I. Adler, S. G. Kolliopoulos, P. K. Krause, D. Lokshtanov, S. Saurabh, and D. M. Thilikos. Irrelevant vertices for the planar disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 122:815–843, 2017. URL https://doi.org/10.1016/j.jctb.2016.10.001.

J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN 2002)*, volume 2286 of *LNCS*, pages 613–627. Springer, 2002. URL https://doi.org/10.1007/3-540-45995-2_52.

J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, 33 (4):461–493, 2002. URL https://doi.org/10.1007/s00453-001-0116-5.

B. Alecu, M. Chudnovsky, S. Hajebi, and S. Spirkl. Induced subgraphs and tree decompositions XIII. Basic obstructions in $\mathcal{H}$-free graphs for finite $\mathcal{H}$. *arXiv math*, abs/2311.05066, 2023. URL https://doi.org/10.48550/arXiv.2311.05066.

M. Alekhnovich and A. A. Razborov. Satisfiability, branch-width and tseitin tautologies. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS 2002)*, pages 593–603. IEEE, 2002. URL https://doi.org/10.1109/SFCS.2002.1181983.

M. Alekhnovich and A. A. Razborov. Satisfiability, branch-width and tseitin tautologies. *Computational Complexity*, 20(4):649–678, 2011. URL https://doi.org/10.1007/s00037-011-0033-1.

J. Alman, M. Mnich, and V. V. Williams. Dynamic parameterized problems and algorithms. *ACM Transactions on Algorithms*, 16(4):45:1–45:46, 2020. URL https://doi.org/10.1145/3395037.

S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005. URL https://doi.org/10.1145/1103963.1103966.

A. Amarilli and M. Monet. Weighted counting of matchings in unbounded-treewidth graph families. In *Proceedings of the 47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, volume 241 of *LIPIcs*, pages 9:1–9:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL https://doi.org/10.4230/LIPIcs.MFCS.2022.9. Full version: https://arxiv.org/abs/2205.00851.

A. Amarilli, P. Bourhis, and P. Senellart. Tractable lineages on treelike instances: Limits and extensions. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2016)*, pages 355–370. ACM, 2016. URL https://doi.org/10.1145/2902251.2902301.

A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. *arXiv CoRR*, abs/1812.09519, 2018. URL https://doi.org/10.48550/arXiv.1812.09519.

A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2019)*, pages 89–103. ACM, 2019. URL https://doi.org/10.1145/3294052.3319702.

A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *Theory of Computing Systems*, 64(5):861–914, 2020. URL https://doi.org/10.1007/s00224-019-09930-2.

E. Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, 2010. URL https://doi.org/10.1007/s00453-008-9180-4.

S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989. URL https://doi.org/10.1016/0166-218X(89)90031-0.

S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8:277–284, 1987. URL https://doi.org/10.1137/0608024.

S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991. URL https://doi.org/10.1016/0196-6774(91)90006-K.

P. Austrin, P. Kaski, and K. Kubjas. Tensor network complexity of multilinear maps. *Theory of Computing*, 18:1–54, 2022. URL https://doi.org/10.4086/toc.2022.v018a016.

B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs (preliminary version). In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*, pages 265–273. IEEE, 1983. URL https://doi.org/10.1109/SFCS.1983.7.

B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, 1994. URL https://doi.org/10.1145/174644.174650.

J. Baste, I. Sau, and D. M. Thilikos. Hitting minors on bounded treewidth graphs. IV. An optimal algorithm. *SIAM Journal on Computing*, 52(4):865–912, 2023. URL https://doi.org/10.1137/21m140482x.

P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004. URL https://doi.org/10.1613/jair.1410.

M. Belbasi and M. Fürer. Finding all leftmost separators of size $\leq k$. In *Proceedings of 15th International Conference on Combinatorial Optimization and Applications (COCOA 2021)*, volume 13135 of *LNCS*, pages 273–287. Springer, 2021. URL https://doi.org/10.1007/978-3-030-92681-6_23. Full version: https://arxiv.org/abs/2111.02614.

M. Belbasi and M. Fürer. An improvement of Reed's treewidth approximation. *Journal of Graph Algorithms and Applications*, 26(2):257–282, 2022. URL https://doi.org/10.7155/jgaa.00593.

P. Bellenbaum and R. Diestel. Two short proofs concerning tree-decompositions. *Combinatorics, Probability and Computing*, 11(6):541–547, 2002. URL https://doi.org/10.1017/S0963548302005369.

B. Bergougnoux and M. M. Kanté. More applications of the d-neighbor equivalence: Acyclicity and connectivity constraints. *SIAM Journal on Discrete Mathematics*, 35 (3):1881–1926, 2021. URL https://doi.org/10.1137/20M1350571.

B. Bergougnoux, T. Korhonen, and J. Nederlof. Tight lower bounds for problems parameterized by rank-width. In *Proceedings of the 40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *LIPIcs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL

https://doi.org/10.4230/LIPIcs.STACS.2023.11. Full version: https://arxiv.org/abs/2210.02117.

U. Bertele and F. Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973. URL https://doi.org/10.1016/0097-3165(73)90016-2.

A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: Fast subset convolution. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC 2007)*, pages 67–74. ACM, 2007. URL https://doi.org/10.1145/1250790.1250801.

A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, 39(2):546–563, 2009. URL https://doi.org/10.1137/070683933.

H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming (ICALP 1988)*, volume 317 of *LNCS*, pages 105–118. Springer, 1988. URL https://doi.org/10.1007/3-540-19488-6_110.

H. L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science, (WG 1993)*, volume 790 of *LNCS*, pages 112–124. Springer, 1993. URL https://doi.org/10.1007/3-540-57899-4_45.

H. L. Bodlaender. Improved self-reduction algorithms for graphs with bounded treewidth. *Discrete Applied Mathematics*, 54(2-3):101–115, 1994. URL https://doi.org/10.1016/0166-218X(94)90018-3.

H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. URL https://doi.org/10.1137/S0097539793251219.

H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27(6):1725–1746, 1998. URL https://doi.org/10.1137/S0097539795289859.

H. L. Bodlaender and T. Kloks. Better algorithms for the pathwidth and treewidth of graphs. In *Proceedings of the 18th International Colloquium of Automata, Languages and Programming (ICALP 1991)*, volume 510 of *LNCS*, pages 544–555. Springer, 1991. URL https://doi.org/10.1007/3-540-54233-7_162.

H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996. URL https://doi.org/10.1006/jagm.1996.0049.

H. L. Bodlaender and A. M. C. A. Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006. URL https://doi.org/10.1016/j.disc.2005.12.017.

H. L. Bodlaender and D. M. Thilikos. Constructive linear time algorithms for branchwidth. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP 1997)*, volume 1256 of *LNCS*, pages 627–637. Springer, 1997. URL https://doi.org/10.1007/3-540-63165-8_217. Full version: http://hdl.handle.net/2117/96447.

H. L. Bodlaender and D. M. Thilikos. Computing small search numbers in linear time. In *Proceedings of the First International Workshop on Parameterized and Exact Computation (IWPEC 2004)*, volume 3162 of *LNCS*, pages 37–48. Springer, 2004. URL https://doi.org/10.1007/978-3-540-28639-4_4.

H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995. URL https://doi.org/10.1006/jagm.1995.1009.

H. L. Bodlaender, L. Cai, J. Chen, M. R. Fellows, J. A. Telle, and D. Marx. Open problems in parameterized and exact computation – IWPEC 2006. Technical Report UU-CS-2006-052, Department of Information and Computing Sciences, Utrecht University, 2006. URL https://dspace.library.uu.nl/handle/1874/22186.

H. L. Bodlaender, M. R. Fellows, and D. M. Thilikos. Derivation of algorithms for cutwidth and related graph layout parameters. *Journal of Computer and System Sciences*, 75(4):231–244, 2009. URL https://doi.org/10.1016/j.jcss.2008.10.003.

H. L. Bodlaender, M. Cygan, S. Kratsch, and J. Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Information and Computation*, 243:86–111, 2015. URL https://doi.org/10.1016/j.ic.2014.12.008.

H. L. Bodlaender, P. G. Drange, M. S. Dregi, F. V. Fomin, D. Lokshtanov, and M. Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45(2):317–378, 2016a. URL https://doi.org/10.1137/130947374.

H. L. Bodlaender, F. V. Fomin, D. Lokshtanov, E. Penninkx, S. Saurabh, and D. M. Thilikos. (Meta) Kernelization. *Journal of the ACM*, 63(5):44:1–44:69, 2016b. URL https://doi.org/10.1145/2973749.

H. L. Bodlaender, L. Jaffke, and J. A. Telle. Typical sequences revisited - Computing width parameters of graphs. *Theory of Computing Systems*, 67(1):52–88, 2023. URL https://doi.org/10.1007/s00224-021-10030-3.

M. Bojanczyk and M. Pilipczuk. Definability equals recognizability for graphs of bounded treewidth. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*, pages 407–416. ACM, 2016. URL https://doi.org/10.1145/2933575.2934508. Full version: https://arxiv.org/abs/1605.03045.

M. Bojanczyk, M. Grohe, and M. Pilipczuk. Definable decompositions for graphs of bounded linear cliquewidth. *Logical Methods in Computer Science*, 17(1), 2021. URL https://doi.org/10.23638/LMCS-17(1:5)2021.

M. Bojańczyk and M. Pilipczuk. Optimizing tree decompositions in MSO. *Logical Methods in Computer Science*, 18(1), 2022. URL https://doi.org/10.46298/lmcs-18(1:26)2022.

É. Bonnet, E. J. Kim, S. Thomassé, and R. Watrigant. Twin-width I: Tractable FO model checking. *Journal of the ACM*, 69(1):3:1–3:46, 2022. URL https://doi.org/10.1145/3486655.

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(5&6):555–581, 1992. URL https://doi.org/10.1007/BF01758777.

B. Bui-Xuan, J. A. Telle, and M. Vatshelle. H-join decomposable graphs and algorithms with runtime single exponential in rankwidth. *Discrete Applied Mathematics*, 158(7):809–819, 2010. URL https://doi.org/10.1016/j.dam.2009.09.009.

B. Bui-Xuan, J. A. Telle, and M. Vatshelle. Boolean-width of graphs. *Theoretical Computer Science*, 412(39):5187–5204, 2011. URL https://doi.org/10.1016/j.tcs.2011.05.022.

S. Buss and J. Nordström. Proof complexity and SAT solving. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 233–350. IOS Press, 2021. URL https://doi.org/10.3233/FAIA200990.

A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC 1977)*, pages 77–90. ACM, 1977. URL https://doi.org/10.1145/800105.803397.

V. Chandrasekaran, N. Srebro, and P. Harsha. Complexity of inference in graphical models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI 2008)*, pages 70–78. AUAI Press, 2008. Accessed from `https://arxiv.org/abs/1206.3240`.

C. Chekuri and J. Chuzhoy. Polynomial bounds for the grid-minor theorem. *Journal of the ACM*, 63(5):40:1–40:65, 2016. URL `https://doi.org/10.1145/2820609`.

C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000. URL `https://doi.org/10.1016/S0304-3975(99)00220-0`.

J. Chen, Y. Liu, and S. Lu. An improved parameterized algorithm for the minimum node multiway cut problem. *Algorithmica*, 55(1):1–13, 2009. URL `https://doi.org/10.1007/s00453-007-9130-6`.

J. Chen, W. Czerwiński, Y. Disser, A. E. Feldmann, D. Hermelin, W. Nadara, M. Pilipczuk, M. Pilipczuk, M. Sorge, B. Wróblewski, and A. Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, pages 796–809. SIAM, 2021. URL `https://doi.org/10.1137/1.9781611976465.50`. Full version: `https://arxiv.org/abs/2006.00571`.

L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *Proceedings of the 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2022)*, pages 612–623. IEEE, 2022. URL `https://doi.org/10.1109/FOCS54457.2022.00064`. Full version: `https://arxiv.org/abs/2203.00671`.

K. Cho, E. Oh, and S. Oh. Parameterized algorithm for the disjoint path problem on planar graphs: Exponential in k^2 and linear in n. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, pages 3734–3758. SIAM, 2023. URL `https://doi.org/10.1137/1.9781611977554.ch144`. Full version: `https://arxiv.org/abs/2211.03341`.

J. Chuzhoy and Z. Tan. Towards tight(er) bounds for the excluded grid theorem. *Journal of Combinatorial Theory, Series B*, 146:219–265, 2021. URL `https://doi.org/10.1016/j.jctb.2020.09.010`.

R. F. Cohen, S. Sairam, R. Tamassia, and J. S. Vitter. Dynamic algorithms for optimization problems in bounded tree-width graphs. In *Proceedings of the 3rd Integer Programming and Combinatorial Optimization Conference (IPCO 1993)*, pages 99–112. CIACO, 1993. The author notes he could not access this reference.

W. Cook and P. D. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15:233–248, 2003. URL https://doi.org/10.1287/ijoc.15.3.233.16078.

D. G. Corneil and U. Rotics. On the relationship between clique-width and treewidth. *SIAM Journal on Computing*, 34(4):825–847, 2005. URL https://doi.org/10.1137/S0097539701385351.

B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990. URL https://doi.org/10.1016/0890-5401(90)90043-H.

B. Courcelle. The monadic second-order logic of graphs V: On closing the gap between definability and recognizability. *Theoretical Computer Science*, 80(2):153–202, 1991. URL https://doi.org/10.1016/0304-3975(91)90387-H.

B. Courcelle. The monadic second-order logic of graphs VIII: Orientations. *Annals of Pure and Applied Logic*, 72(2):103–143, 1995. URL https://doi.org/10.1016/0168-0072(95)94698-V.

B. Courcelle and J. Engelfriet. A logical characterization of the sets of hypergraphs defined by hyperedge replacement grammars. *Mathematical Systems Theory*, 28(6):515–552, 1995.

B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic — A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2012. ISBN 978-0-521-89833-1. URL https://doi.org/10.1017/CBO9780511977619.

B. Courcelle and S. Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3):77–114, 2000. URL https://doi.org/10.1016/S0166-218X(99)00184-5.

B. Courcelle and S. Oum. Vertex-minors, monadic second-order logic, and a conjecture by Seese. *Journal of Combinatorial Theory, Series B*, 97(1):91–126, 2007. URL https://doi.org/10.1016/j.jctb.2006.04.003.

B. Courcelle, J. Engelfriet, and G. Rozenberg. Handle-rewriting hypergraph grammars. *Journal of Computer and System Sciences*, 46(2):218–270, 1993. URL https://doi.org/10.1016/0022-0000(93)90004-G.

B. Courcelle, J. A. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory of Computing Systems*, 33(2):125–150, 2000. URL https://doi.org/10.1007/s002249910009.

B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1-2):23–52, 2001. URL https://doi.org/10.1016/S0166-218X(00)00221-3.

W. H. Cunningham and J. Geelen. On integer programming and the branch-width of the constraint matrix. In *Proceedings of the 12th International Integer Programming and Combinatorial Optimization Conference (IPCO 2007)*, volume 4513 of *LNCS*, pages 158–166. Springer, 2007. URL https://doi.org/10.1007/978-3-540-72792-7_13.

M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. M. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011)*, pages 150–159. IEEE, 2011. URL https://doi.org/10.1109/FOCS.2011.23.

M. Cygan, D. Marx, M. Pilipczuk, and M. Pilipczuk. The planar directed k-Vertex-Disjoint Paths problem is fixed-parameter tractable. In *Proceedings of the 54th Annual Symposium on Foundations of Computer Science (FOCS 2013)*, pages 197–207. IEEE, 2013. URL https://doi.org/10.1109/FOCS.2013.29. Full version: https://arxiv.org/abs/1304.4207.

M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015. URL https://doi.org/10.1007/978-3-319-21275-3.

M. Cygan, M. Pilipczuk, and M. Pilipczuk. Known algorithms for edge clique cover are probably optimal. *SIAM Journal on Computing*, 45(1):67–83, 2016. URL https://doi.org/10.1137/130947076.

M. Cygan, D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. Minimum bisection is fixed-parameter tractable. *SIAM Journal on Computing*, 48(2):417–450, 2019. URL https://doi.org/10.1137/140988553.

M. Cygan, P. Komosa, D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, S. Saurabh, and M. Wahlström. Randomized contractions meet lean decompositions. *ACM Transactions on Algorithms*, 17(1):6:1–6:30, 2021. URL https://doi.org/10.1145/3426738.

M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. M. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. *ACM Transactions on Algorithms*, 18(2):17:1–17:31, 2022. URL https://doi.org/10.1145/3506707.

C. Dallard, F. V. Fomin, P. A. Golovach, T. Korhonen, and M. Milanic. Computing tree decompositions with small independence number. *arXiv CoRR*, abs/2207.09993, 2022. URL https://doi.org/10.48550/arXiv.2207.09993.

V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science*, 329(1-3):315–323, 2004. URL https://doi.org/10.1016/j.tcs.2004.08.008.

G. Damiand, M. Habib, and C. Paul. A simple paradigm for graph recognition: Application to cographs and distance hereditary graphs. *Theoretical Computer Science*, 263(1-2): 99–111, 2001. URL https://doi.org/10.1016/S0304-3975(00)00234-6.

A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. URL https://doi.org/10.1613/jair.989.

A. Dawar, M. Grohe, S. Kreutzer, and N. Schweikardt. Approximation schemes for first-order definable optimisation problems. In *Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 411–420. IEEE, 2006. URL https://doi.org/10.1109/LICS.2006.13.

A. Dawar, M. Grohe, and S. Kreutzer. Locally excluding a minor. In *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 270–279. IEEE, 2007. URL https://doi.org/10.1109/LICS.2007.31.

A. de Colnet and S. Mengel. Characterizing tseitin-formulas with short regular resolution refutations. *Journal of Artificial Intelligence Research*, 76:265–286, 2023. URL https://doi.org/10.1613/jair.1.13521.

R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999. URL https://doi.org/10.1016/S0004-3702(99)00059-4.

R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989. URL https://doi.org/10.1016/0004-3702(89)90037-4.

E. D. Demaine and M. Hajiaghayi. Linearity of grid minors in treewidth with applications through bidimensionality. *Combinatorica*, 28(1):19–36, 2008. URL https://doi.org/10.1007/s00493-008-2140-4.

E. D. Demaine and M. T. Hajiaghayi. Equivalence of local treewidth and linear local treewidth and its algorithmic applications. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 840–849. SIAM, 2004. URL https://dl.acm.org/doi/10.5555/982792.982919.

E. D. Demaine and M. T. Hajiaghayi. Bidimensionality: New connections between FPT algorithms and ptass. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 590–601. SIAM, 2005. URL https://dl.acm.org/doi/10.5555/1070432.1070514.

E. D. Demaine, F. V. Fomin, M. Hajiaghayi, and D. M. Thilikos. Subexponential parameterized algorithms on graphs of bounded genus and H-minor-free graphs. *Journal of the ACM*, 52(6):866–893, 2005a. URL https://doi.org/10.1145/1101821.1101823.

E. D. Demaine, M. T. Hajiaghayi, and D. M. Thilikos. Exponential speedup of fixed-parameter algorithms for classes of graphs excluding single-crossing graphs as minors. *Algorithmica*, 41(4):245–267, 2005b. URL https://doi.org/10.1007/s00453-004-1125-y.

E. D. Demaine, M. Hajiaghayi, and D. M. Thilikos. The bidimensional theory of bounded-genus graphs. *SIAM Journal on Discrete Mathematics*, 20(2):357–371, 2006. URL https://doi.org/10.1137/040616929.

E. D. Demaine, M. Hajiaghayi, and K. Kawarabayashi. Algorithmic graph minor theory: Improved grid minor bounds and Wagner's contraction. *Algorithmica*, 54(2):142–180, 2009. URL https://doi.org/10.1007/s00453-007-9138-y.

I. Dinur and D. Steurer. Analytical approach to parallel repetition. In *Proceedings of the 46th Symposium on Theory of Computing (STOC 2014)*, pages 624–633. ACM, 2014. URL https://doi.org/10.1145/2591796.2591884. Full version: https://arxiv.org/abs/1305.1979.

S. Dong, Y. T. Lee, and G. Ye. A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2021)*, pages 1784–1797. ACM, 2021. URL https://doi.org/10.1145/3406325.3451056. Full version: https://arxiv.org/abs/2011.05365.

R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer, 1999. ISBN 978-0-387-94883-6. URL https://doi.org/10.1007/978-1-4612-0515-9.

R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013. ISBN 978-1-4471-5558-4. URL https://doi.org/10.1007/978-1-4471-5559-1.

J. Dreier, N. Mählmann, and S. Siebertz. First-order model checking on structurally sparse graph classes. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023)*. ACM, 2023. URL https://doi.org/10.1145/3564246.3585186. Full version: https://arxiv.org/abs/2302.03527.

Z. Dvorák. Thin graph classes and polynomial-time approximation schemes. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 1685–1701. SIAM, 2018. URL https://doi.org/10.1137/1.9781611975031.110. Full version: https://arxiv.org/abs/1704.00125.

Z. Dvorák, D. Král, and R. Thomas. Testing first-order properties for subclasses of sparse graphs. *Journal of the ACM*, 60(5):36:1–36:24, 2013. URL https://doi.org/10.1145/2499483.

Z. Dvořák, M. Kupec, and V. Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA 2014)*, volume 8737 of *LNCS*, pages 334–345. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44777-2_28.

M. Elberfeld and P. Schweitzer. Canonizing graphs of bounded tree width in logspace. *ACM Transactions on Computation Theory*, 9(3):12:1–12:29, 2017. URL https://doi.org/10.1145/3132720.

M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Proceedings of the 51th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)*, pages 143–152. IEEE, 2010. URL https://doi.org/10.1109/FOCS.2010.21. Full version: https://eccc.weizmann.ac.il/report/2010/062/.

D. Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27 (3):275–291, 2000. URL https://doi.org/10.1007/s004530010020.

D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification. I. Planary testing and minimum spanning trees. *Journal of Computer and System Sciences*, 52(1):3–27, 1996. URL https://doi.org/10.1006/jcss.1996.0002.

J. Erde. A unified treatment of linked and lean tree-decompositions. *Journal of Combinatorial Theory, Series B*, 130:114–143, 2018. URL https://doi.org/10.1016/j.jctb.2017.12.001.

W. Espelage, F. Gurski, and E. Wanke. How to solve NP-hard graph problems on clique-width bounded graphs in polynomial time. In *Proceedings of the 27th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2001)*, volume 2204 of *LNCS*, pages 117–128. Springer, 2001. URL https://doi.org/10.1007/3-540-45477-2_12.

C. C. Fast and I. V. Hicks. A branch decomposition algorithm for the p-median problem. *INFORMS Journal on Computing*, 29(3):474–488, 2017. URL https://doi.org/10.1287/ijoc.2016.0743.

T. Feder and M. Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998. URL https://doi.org/10.1137/S0097539794266766.

U. Feige, M. Hajiaghayi, and J. R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM Journal on Computing*, 38(2):629–657, 2008. URL https://doi.org/10.1137/05064299X.

M. R. Fellows and M. A. Langston. On search, decision, and the efficiency of polynomial-time algorithms. *Journal of Computer and System Sciences*, 49(3):769–779, 1994. URL https://doi.org/10.1016/S0022-0000(05)80079-0.

M. R. Fellows, F. A. Rosamond, U. Rotics, and S. Szeider. Clique-width is NP-complete. *SIAM Journal on Discrete Mathematics*, 23(2):909–939, 2009. URL https://doi.org/10.1137/070687256.

J. Flum and M. Grohe. Fixed-parameter tractability, definability, and model-checking. *SIAM Journal on Computing*, 31(1):113–145, 2001. URL https://doi.org/10.1137/S0097539799360768.

J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006. ISBN 978-3-540-29952-3. URL https://doi.org/10.1007/3-540-29953-X.

F. V. Fomin and D. M. Thilikos. Dominating sets in planar graphs: Branch-width and exponential speed-up. *SIAM Journal on Computing*, 36:281–309, 2006. URL https://doi.org/10.1137/S0097539702419649.

F. V. Fomin, P. A. Golovach, D. Lokshtanov, and S. Saurabh. Intractability of clique-width parameterizations. *SIAM Journal on Computing*, 39(5):1941–1956, 2010. URL https://doi.org/10.1137/080742270.

F. V. Fomin, P. A. Golovach, and D. M. Thilikos. Contraction obstructions for treewidth. *Journal of Combinatorial Theory, Series B*, 101(5):302–314, 2011a. URL https://doi.org/10.1016/j.jctb.2011.02.008.

F. V. Fomin, D. Lokshtanov, V. Raman, and S. Saurabh. Subexponential algorithms for partial cover problems. *Information Processing Letters*, 111(16):814–818, 2011b. URL https://doi.org/10.1016/j.ipl.2011.05.016.

F. V. Fomin, I. Todinca, and Y. Villanger. Large induced subgraphs via triangulations and CMSO. *SIAM Journal on Computing*, 44(1):54–87, 2015. URL https://doi.org/10.1137/140964801.

F. V. Fomin, D. Lokshtanov, F. Panolan, and S. Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *Journal of the ACM*, 63(4):29:1–29:60, 2016. URL https://doi.org/10.1145/2886094.

F. V. Fomin, D. Lokshtanov, F. Panolan, and S. Saurabh. Representative families of product families. *ACM Transactions on Algorithms*, 13(3):36:1–36:29, 2017. URL https://doi.org/10.1145/3039243.

F. V. Fomin, D. Lokshtanov, S. Saurabh, M. Pilipczuk, and M. Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms*, 14(3):34:1–34:45, 2018. URL https://doi.org/10.1145/3186898.

F. V. Fomin, D. Lokshtanov, S. Saurabh, and D. M. Thilikos. Bidimensionality and kernels. *SIAM Journal on Computing*, 49(6):1397–1422, 2020. URL https://doi.org/10.1137/16M1080264.

L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956. URL https://doi.org/10.4153/CJM-1956-045-5.

E. Fox-Epstein, P. N. Klein, and A. Schild. Embedding planar graphs into low-treewidth graphs with applications to efficient approximation schemes for metric problems. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pages 1069–1088. SIAM, 2019. URL https://doi.org/10.1137/1.9781611975482.66.

G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997. URL https://doi.org/10.1006/jagm.1996.0835.

G. N. Frederickson. Maintaining regular properties dynamically in k-terminal graphs. *Algorithmica*, 22(3):330–350, 1998. URL https://doi.org/10.1007/PL00009227.

E. C. Freuder. Complexity of k-tree structured constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI 1990)*, pages 4–9. AAAI Press / The MIT Press, 1990. URL http://www.aaai.org/Library/AAAI/1990/aaai90-001.php.

M. Frick and M. Grohe. Deciding first-order properties of locally tree-decomposable structures. *Journal of the ACM*, 48(6):1184–1206, 2001. URL https://doi.org/10.1145/504794.504798.

H. Gaifman. On local and non-local properties. In J. Stern, editor, *Proceedings of the Herbrand Symposium*, volume 107 of *Studies in Logic and the Foundations of Mathemat-*

ics, pages 105–135. Elsevier, 1982. URL https://doi.org/10.1016/S0049-237X(08)71879-2.

R. Ganian and P. Hliněný. On parse trees and myhill-nerode-type tools for handling graphs of bounded rank-width. *Discrete Applied Mathematics*, 158(7):851–867, 2010. URL https://doi.org/10.1016/j.dam.2009.10.018.

R. Ganian, P. Hliněný, and J. Obdrzálek. Better algorithms for satisfiability problems for formulas of bounded rank-width. *Fundamenta Informaticae*, 123(1):59–76, 2013. URL https://doi.org/10.3233/FI-2013-800.

R. Ganian, P. Hliněný, A. Langer, J. Obdrzálek, P. Rossmanith, and S. Sikdar. Lower bounds on the complexity of MSO_1 model-checking. *Journal of Computer and System Sciences*, 80(1):180–194, 2014. URL https://doi.org/10.1016/j.jcss.2013.07.005.

J. Geelen, O. Kwon, R. McCarty, and P. Wollan. The grid theorem for vertex-minors. *Journal of Combinatorial Theory, Series B*, 158(1):93–116, 2023. URL https://doi.org/10.1016/j.jctb.2020.08.004.

J. F. Geelen, A. M. H. Gerards, and G. Whittle. Branch-width and well-quasi-ordering in matroids and graphs. *Journal of Combinatorial Theory, Series B*, 84(2):270–290, 2002. URL https://doi.org/10.1006/jctb.2001.2082.

M. U. Gerber and D. Kobler. Algorithms for vertex-partitioning problems on graphs with fixed clique-width. *Theoretical Computer Science*, 299(1-3):719–734, 2003. URL https://doi.org/10.1016/S0304-3975(02)00725-9.

A. C. Giannopoulou, M. Pilipczuk, J. Raymond, D. M. Thilikos, and M. Wrochna. Cutwidth: Obstructions and algorithmic aspects. *Algorithmica*, 81(2):557–588, 2019. URL https://doi.org/10.1007/s00453-018-0424-7.

A. C. Giannopoulou, O. Kwon, J. Raymond, and D. M. Thilikos. A Menger-like property of tree-cut width. *Journal of Combinatorial Theory, Series B*, 148:1–22, 2021. URL https://doi.org/10.1016/j.jctb.2020.12.005.

V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence (UAI 2004)*, pages 201–208. AUAI Press, 2004. Accessed from https://arxiv.org/abs/1207.4109.

G. Goranci, H. Räcke, T. Saranurak, and Z. Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, pages 2212–2228. SIAM,

2021. URL https://doi.org/10.1137/1.9781611976465.132. Full version: https://arxiv.org/abs/2005.02369.

G. Gottlob and R. Pichler. Hypergraphs in model checking: Acyclicity and hypertree-width versus clique-width. *SIAM Journal on Computing*, 33(2):351–378, 2004. URL https://doi.org/10.1137/S0097539701396807.

G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002. URL https://doi.org/10.1006/jcss.2001.1809.

A. Grez, F. Mazowiecki, M. Pilipczuk, G. Puppis, and C. Riveros. Dynamic data structures for timed automata acceptance. *Algorithmica*, 84(11):3223–3245, 2022. URL https://doi.org/10.1007/s00453-022-01025-8.

M. Grohe. Local tree-width, excluded minors, and approximation algorithms. *Combinatorica*, 23(4):613–632, 2003. URL https://doi.org/10.1007/s00493-003-0037-9.

M. Grohe. Computing crossing numbers in quadratic time. *Journal of Computer and System Sciences*, 68(2):285–302, 2004. URL https://doi.org/10.1016/j.jcss.2003.07.008.

M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM*, 54(1):1:1–1:24, 2007. URL https://doi.org/10.1145/1206035.1206036.

M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms*, 11(1):4:1–4:20, 2014. URL https://doi.org/10.1145/2636918.

M. Grohe and D. Marx. Structure theorem and isomorphism test for graphs with excluded topological subgraphs. *SIAM Journal on Computing*, 44(1):114–159, 2015. URL https://doi.org/10.1137/120892234.

M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing (STOC 2001)*, pages 657–666. ACM, 2001. URL https://doi.org/10.1145/380752.380867.

M. Grohe, S. Kreutzer, and S. Siebertz. Deciding first-order properties of nowhere dense graphs. *Journal of the ACM*, 64(3):17:1–17:32, 2017. URL https://doi.org/10.1145/3051095.

Q. Gu and H. Tamaki. Improved bounds on the planar branchwidth with respect to the largest grid minor size. *Algorithmica*, 64(3):416–453, 2012. URL https://doi.org/10.1007/s00453-012-9627-5.

T. Hagerup. Dynamic algorithms for graphs of bounded treewidth. *Algorithmica*, 27(3): 292–315, 2000. URL https://doi.org/10.1007/s004530010021.

R. Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976. URL https://doi.org/10.1007/BF01917434.

P. Hliněný. A parametrized algorithm for matroid branch-width. *SIAM Journal on Computing*, 35(2):259–277, 2005. URL https://doi.org/10.1137/S0097539702418589.

P. Hliněný. Branch-width, parse trees, and monadic second-order logic for matroids. *Journal of Combinatorial Theory, Series B*, 96(3):325–351, 2006. URL https://doi.org/10.1016/j.jctb.2005.08.005.

P. Hliněný and S. Oum. Finding branch-decompositions and rank-decompositions. *SIAM Journal on Computing*, 38(3):1012–1032, 2008. URL https://doi.org/10.1137/070685920.

P. Hliněný and D. Seese. Trees, grids, and MSO decidability: From graphs to matroids. *Theoretical Computer Science*, 351(3):372–393, 2006. URL https://doi.org/10.1016/j.tcs.2005.10.006.

P. Hliněný, S. Oum, D. Seese, and G. Gottlob. Width parameters beyond tree-width and their applications. *Comput. J.*, 51(3):326–362, 2008. URL https://doi.org/10.1093/comjnl/bxm052.

H. B. Hunt, M. V. Marathe, V. Radhakrishnan, S. S. Ravi, D. J. Rosenkrantz, and R. E. Stearns. NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs. *Journal of Algorithms*, 26(2):238–274, 1998. URL https://doi.org/10.1006/jagm.1997.0903.

R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. URL https://doi.org/10.1006/jcss.2000.1727.

R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity. *Journal of Computer and System Sciences*, 63(4):512–530, 2001. URL https://doi.org/10.1006/jcss.2001.1774.

T. Inamdar, D. Lokshtanov, S. Saurabh, and V. Surianarayanan. Parameterized complexity of fair bisection: (FPT-approximation meets unbreakability). In *Proceedings of the 31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *LIPIcs*, pages 63:1–63:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL https://doi.org/10.4230/LIPIcs.ESA.2023.63.

D. Itsykson, A. Riazanov, D. Sagunov, and P. Smirnov. Near-optimal lower bounds on regular resolution refutations of tseitin formulas for all constant-degree graphs. *Computational Complexity*, 30(2):13, 2021. URL https://doi.org/10.1007/s00037-021-00213-2.

D. Itsykson, A. Riazanov, and P. Smirnov. Tight bounds for Tseitin formulas. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *LIPIcs*, pages 6:1–6:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL https://doi.org/10.4230/LIPIcs.SAT.2022.6.

Y. Iwata and K. Oka. Fast dynamic graph algorithms for parameterized problems. In *Proceedings of the 14th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2014)*, volume 8503 of *LNCS*, pages 241–252. Springer, 2014. URL https://doi.org/10.1007/978-3-319-08404-6_21.

B. M. P. Jansen, J. J. H. de Kroon, and M. Wlodarczyk. 5-approximation for $\mathcal{H}$-treewidth essentially as fast as $\mathcal{H}$-deletion parameterized by solution size. In *Proceedings of the 31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *LIPIcs*, pages 66:1–66:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL https://doi.org/10.4230/LIPIcs.ESA.2023.66.

J. Jeong, E. J. Kim, and S. Oum. The "art of trellis decoding" is fixed-parameter tractable. *IEEE Trans. Inf. Theory*, 63(11):7178–7205, 2017. URL https://doi.org/10.1109/TIT.2017.2740283.

J. Jeong, E. J. Kim, and S. Oum. Finding branch-decompositions of matroids, hypergraphs, and more. *SIAM Journal on Discrete Mathematics*, 35(4):2544–2617, 2021. URL https://doi.org/10.1137/19M1285895.

M. Jerrum and M. Snir. Some exact complexity results for straight-line computations over semirings. *Journal of the ACM*, 29(3):874–897, 1982. URL https://doi.org/10.1145/322326.322341.

M. Johnson, B. Martin, J. J. Oostveen, S. Pandey, D. Paulusma, S. Smith, and E. J. van Leeuwen. Complexity framework for forbidden subgraphs. *arXiv CoRR*, abs/2211.12887, 2022. URL https://doi.org/10.48550/arXiv.2211.12887.

T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001. URL https://doi.org/10.1006/jctb.2000.2031.

S. Jukna. *Tropical Circuit Complexity: Limits of Pure Dynamic Programming*. Springer, 2023. ISBN 978-3-031-42353-6. URL https://doi.org/10.1007/978-3-031-42354-3.

M. M. Kanté, E. J. Kim, O. Kwon, and S. Oum. Obstructions for matroids of path-width at most k and graphs of linear rank-width at most k. *Journal of Combinatorial Theory, Series B*, 160:15–35, 2023. URL https://doi.org/10.1016/j.jctb.2022.12.004.

K. Kawarabayashi and Y. Kobayashi. Linear min-max relation between the treewidth of an H-minor-free graph and its largest grid minor. *Journal of Combinatorial Theory, Series B*, 141:165–180, 2020. URL https://doi.org/10.1016/j.jctb.2019.07.007.

K. Kawarabayashi and S. Kreutzer. The directed grid theorem. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC 2015)*, pages 655–664. ACM, 2015. URL https://doi.org/10.1145/2746539.2746586. Full version: https://arxiv.org/abs/1411.5681.

K. Kawarabayashi and P. Wollan. A simpler algorithm and shorter proof for the graph minor decomposition. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC 2011)*, pages 451–458. ACM, 2011. URL https://doi.org/10.1145/1993636.1993697. Full version: http://research.nii.ac.jp/~k_keniti/easystruct.pdf.

K. Kawarabayashi, Y. Kobayashi, and B. A. Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012. URL https://doi.org/10.1016/j.jctb.2011.07.004.

S. Khot. On the power of unique 2-prover 1-round games. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, pages 767–775. ACM, 2002. URL https://doi.org/10.1145/509907.510017.

E. J. Kim, S. Oum, C. Paul, I. Sau, and D. M. Thilikos. An FPT 2-approximation for tree-cut decomposition. *Algorithmica*, 80(1):116–135, 2018. URL https://doi.org/10.1007/s00453-016-0245-5.

U. Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2:7–17, 1992. URL https://doi.org/10.1007/BF01890544.

P. N. Klein, S. A. Plotkin, and S. Rao. Excluded minors, network decomposition, and multicommodity flow. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC 1993)*, pages 682–690. ACM, 1993. URL https://doi.org/10.1145/167088.167261.

J. Kleinberg and É. Tardos. *Algorithm design*. Pearson, 2005. ISBN 0-321-29535-8.

T. Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994. ISBN 3-540-58356-4. URL https://doi.org/10.1007/BFb0045375.

D. Kobler and U. Rotics. Edge dominating set and colorings on graphs with fixed clique-width. *Discrete Applied Mathematics*, 126(2-3):197–221, 2003. URL https://doi.org/10.1016/S0166-218X(02)00198-1.

P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302–332, 2000. URL https://doi.org/10.1006/jcss.2000.1713.

B. Komarath, A. Pandey, and C. S. Rahul. Monotone arithmetic complexity of graph homomorphism polynomials. *Algorithmica*, 85(9):2554–2579, 2023. URL https://doi.org/10.1007/s00453-023-01108-0.

T. Korhonen. Lower bounds on dynamic programming for maximum weight independent set. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *LIPIcs*, pages 87:1–87:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL https://doi.org/10.4230/LIPIcs.ICALP.2021.87.

T. Korhonen. Grid induced minor theorem for graphs of small degree. *Journal of Combinatorial Theory, Series B*, 160:206–214, 2023. URL https://doi.org/10.1016/j.jctb.2023.01.002.

T. Korhonen and D. Lokshtanov. An improved parameterized algorithm for treewidth. *arXiv CoRR*, abs/2211.07154, 2022. URL https://doi.org/10.48550/arXiv.2211.07154.

T. Korhonen and M. Sokołowski. Almost-linear time parameterized algorithm for rankwidth via dynamic rankwidth. *arXiv CoRR*, abs/2402.12364, 2024. URL https://doi.org/10.48550/arXiv.2402.12364. Accepted to appear in STOC 2024.

T. Korhonen, W. Nadara, M. Pilipczuk, and M. Sokołowski. Fully dynamic approximation schemes on planar and apex-minor-free graphs. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2024)*, pages 296–313. SIAM, 2024. URL https://doi.org/10.1137/1.9781611977912.12. Full version: https://arxiv.org/abs/2310.20623.

A. M. C. A. Koster. *Frequency Assignment: Models and Algorithms*. PhD thesis, Maastricht University, 1999. URL https://doi.org/10.26481/dis.19991104ak.

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(3):170–180, 2002. URL https://doi.org/10.1002/net.10046.

S. Kreutzer and S. Tazari. On brambles, grid-like minors, and parameterized intractability of Monadic Second-Order logic. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pages 354–364. SIAM, 2010a. URL https://doi.org/10.1137/1.9781611973075.30. Full version: https://arxiv.org/abs/0907.3076.

S. Kreutzer and S. Tazari. Lower bounds for the complexity of Monadic Second-Order logic. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 189–198. IEEE, 2010b. URL https://doi.org/10.1109/LICS.2010.39.

K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15:271–283, 1930. URL https://eudml.org/doc/212352.

J. H. P. Kwisthout, H. L. Bodlaender, and L. C. van der Gaag. The necessity of bounded treewidth for efficient inference in bayesian networks. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 237–242. IOS Press, 2010. URL https://doi.org/10.3233/978-1-60750-606-5-237.

J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS 1990)*, pages 173–182. IEEE, 1990. URL https://doi.org/10.1109/FSCS.1990.89536.

J. Lagergren. Efficient parallel algorithms for graphs of bounded tree-width. *Journal of Algorithms*, 20(1):20–44, 1996. URL https://doi.org/10.1006/jagm.1996.0002.

J. Lagergren. Upper bounds on the size of obstructions and intertwines. *Journal of Combinatorial Theory, Series B*, 73(1):7–40, 1998. URL https://doi.org/10.1006/jctb.1997.1788.

J. Lagergren and S. Arnborg. Finding minimal forbidden minors using a finite congruence. In *Proceedings of the 18th International Colloquium of Automata, Languages and Programming (ICALP 1991)*, volume 510 of *LNCS*, pages 532–543. Springer, 1991. URL https://doi.org/10.1007/3-540-54233-7_161.

M. Lampis. Finer tight bounds for coloring on clique-width. *SIAM Journal on Discrete Mathematics*, 34(3):1538–1558, 2020. URL https://doi.org/10.1137/19M1280326.

S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2):157–224, 1988. URL https://www.jstor.org/stable/2345762.

A. Leaf and P. D. Seymour. Tree-width and planar minors. *Journal of Combinatorial Theory, Series B*, 111:38–53, 2015. URL https://doi.org/10.1016/j.jctb.2014.09.003.

F. T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999. URL https://doi.org/10.1145/331524.331526.

Y. Li, A. A. Razborov, and B. Rossman. On the AC^0 complexity of subgraph isomorphism. *SIAM Journal on Computing*, 46(3):936–971, 2017. URL https://doi.org/10.1137/14099721X.

D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. Fixed-parameter tractable canonization and isomorphism test for graphs of bounded treewidth. *SIAM Journal on Computing*, 46(1):161–189, 2017. URL https://doi.org/10.1137/140999980.

D. Lokshtanov, D. Marx, and S. Saurabh. Slightly superexponential parameterized problems. *SIAM Journal on Computing*, 47(3):675–702, 2018a. URL https://doi.org/10.1137/16M1104834.

D. Lokshtanov, D. Marx, and S. Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. *ACM Transactions on Algorithms*, 14(2):13:1–13:30, 2018b. URL https://doi.org/10.1145/3170442.

D. Lokshtanov, P. Misra, M. Pilipczuk, S. Saurabh, and M. Zehavi. An exponential time parameterized algorithm for planar disjoint paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*, pages 1307–1316. ACM, 2020a. URL https://doi.org/10.1145/3357713.3384250. Full version: https://arxiv.org/abs/2103.17041.

D. Lokshtanov, S. Saurabh, and V. Surianarayanan. A parameterized approximation scheme for min k-cut. In *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 798–809. IEEE, 2020b. URL https://doi.org/10.1109/FOCS46700.2020.00079. Full version: https://arxiv.org/abs/2005.00134.

K. Majewski, M. Pilipczuk, and M. Sokołowski. Maintaining CMSO_2 properties on dynamic structures with bounded feedback vertex number. In *Proceedings of the 40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *LIPIcs*, pages 46:1–46:13. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2023. URL https://doi.org/10.4230/LIPIcs.STACS.2023.46. Full version: https://arxiv.org/abs/2107.06232.

J. A. Makowsky and J. Mariño. Tree-width and the monadic quantifier hierarchy. *Theoretical Computer Science*, 303(1):157–170, 2003. URL https://doi.org/10.1016/S0304-3975(02)00449-8.

P. Manurangsi and L. Trevisan. Mildly exponential time approximation algorithms for vertex cover, balanced separator and uniform sparsest cut. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2018)*, volume 116 of *LIPIcs*, pages 20:1–20:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2018.20. Full version: https://arxiv.org/abs/1807.09898.

I. L. Markov and Y. Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008. URL https://doi.org/10.1137/050644756.

D. Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351 (3):394–406, 2006. URL https://doi.org/10.1016/j.tcs.2005.10.007.

D. Marx. Approximating fractional hypertree width. *ACM Transactions on Algorithms*, 6(2):29:1–29:17, 2010a. URL https://doi.org/10.1145/1721837.1721845.

D. Marx. Can you beat treewidth? *Theory of Computing*, 6(1):85–112, 2010b. URL https://doi.org/10.4086/toc.2010.v006a005.

D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM*, 60(6):42:1–42:51, 2013. URL https://doi.org/10.1145/2535926.

D. Marx and I. Razgon. Fixed-parameter tractability of multicut parameterized by the size of the cutset. *SIAM Journal on Computing*, 43(2):355–388, 2014. URL https://doi.org/10.1137/110855247.

J. Matoušek and R. Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12(1):1–22, 1991. URL https://doi.org/10.1016/0196-6774(91)90020-Y.

K. Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927. URL http://eudml.org/doc/211191.

C. J. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proceedings of the 25th Canadian Conference on Artificial Intelligence (Canadian AI 2012)*, volume 7310 of *LNCS*, pages 356–361. Springer, 2012. URL https://doi.org/10.1007/978-3-642-30353-1_36.

M. Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2018)*, pages 769–778. ACM, 2018. URL https://doi.org/10.1145/3209108.3209144.

J. Olkowski, M. Pilipczuk, M. Rychlicki, K. Węgrzycki, and A. Zych-Pawlewicz. Dynamic data structures for parameterized string problems. In *Proceedings of the 40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *LIPIcs*, pages 50:1–50:22. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2023. URL https://doi.org/10.4230/LIPIcs.STACS.2023.50.

S. Oum. Rank-width and vertex-minors. *Journal of Combinatorial Theory, Series B*, 95 (1):79–100, 2005. URL https://doi.org/10.1016/j.jctb.2005.03.003.

S. Oum. Rank-width is less than or equal to branch-width. *Journal of Graph Theory*, 57 (3):239–244, 2008a. URL https://doi.org/10.1002/jgt.20280.

S. Oum. Approximating rank-width and clique-width quickly. *ACM Transactions on Algorithms*, 5(1), 2008b. URL https://doi.org/10.1145/1435375.1435385.

S. Oum. Excluding a bipartite circle graph from line graphs. *Journal of Graph Theory*, 60(3):183–203, 2009. URL https://doi.org/10.1002/jgt.20353.

S. Oum. Rank-width: Algorithmic and structural results. *Discrete Applied Mathematics*, 231:15–24, 2017. URL https://doi.org/10.1016/j.dam.2016.08.006.

S. Oum and P. Seymour. Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, 96(4):514–528, 2006. URL https://doi.org/10.1016/j.jctb.2005.10.006.

S. Oum and P. D. Seymour. Testing branch-width. *Journal of Combinatorial Theory, Series B*, 97(3):385–393, 2007. URL https://doi.org/10.1016/j.jctb.2006.06.006.

S. Oum, S. H. Sæther, and M. Vatshelle. Faster algorithms for vertex partitioning problems parameterized by clique-width. *Theoretical Computer Science*, 535:16–24, 2014. URL https://doi.org/10.1016/j.tcs.2014.03.024.

L. Perković and B. A. Reed. An improved algorithm for finding tree decompositions of small width. *International Journal of Foundations of Computer Science*, 11(3):365–371, 2000. URL https://doi.org/10.1142/S0129054100000247.

M. Pilipczuk. Problems parameterized by treewidth tractable in single exponential time: A logical approach. In *Proceedings of the 36th International Symposium on Mathematical*

Foundations of Computer Science (MFCS 2011), volume 6907 of *LNCS*, pages 520–531. Springer, 2011. URL https://doi.org/10.1007/978-3-642-22993-0_47.

M. Pilipczuk. Computing tree decompositions. In F. V. Fomin, S. Kratsch, and E. J. van Leeuwen, editors, *Treewidth, Kernels, and Algorithms — Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*, volume 12160 of *LNCS*, pages 189–213. Springer, 2020. URL https://doi.org/10.1007/978-3-030-42071-0_14.

P. Raghavendra and D. Steurer. Graph expansion and the unique games conjecture. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC 2010)*, pages 755–764. ACM, 2010. URL https://doi.org/10.1145/1806689.1806792.

B. A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC 1992)*, pages 221–228. ACM, 1992. URL https://doi.org/10.1145/129712.129734.

B. A. Reed. Rooted routing in the plane. *Discrete Applied Mathematics*, 57(2-3):213–227, 1995. URL https://doi.org/10.1016/0166-218X(94)00104-L.

B. A. Reed and D. R. Wood. Polynomial treewidth forces a large grid-like-minor. *European Journal of Combinatorics*, 33(3):374–379, 2012. URL https://doi.org/10.1016/j.ejc.2011.09.004.

B. A. Reed, N. Robertson, A. Schrijver, and P. D. Seymour. Finding disjoint trees in planar graphs in linear time. In N. Robertson and P. D. Seymour, editors, *Graph Structure Theory, Proceedings of a AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors held June 22 to July 5, 1991, at the University of Washington, Seattle, USA*, volume 147 of *Contemporary Mathematics*, pages 295–301. American Mathematical Society, 1993.

N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984. URL https://doi.org/10.1016/0095-8956(84)90013-3.

N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986a. URL https://doi.org/10.1016/0196-6774(86)90023-4.

N. Robertson and P. D. Seymour. Graph minors. V. Excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, 1986b. URL https://doi.org/10.1016/0095-8956(86)90030-4.

N. Robertson and P. D. Seymour. Graph minors. IV. Tree-width and well-quasi-ordering. *Journal of Combinatorial Theory, Series B*, 48(2):227–254, 1990. URL https://doi.org/10.1016/0095-8956(90)90120-O.

N. Robertson and P. D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991. URL https://doi.org/10.1016/0095-8956(91)90061-N.

N. Robertson and P. D. Seymour. Graph Minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995. URL https://doi.org/10.1006/jctb.1995.1006.

N. Robertson and P. D. Seymour. Graph minors. XVI. Excluding a non-planar graph. *Journal of Combinatorial Theory, Series B*, 89(1):43–76, 2003. URL https://doi.org/10.1016/S0095-8956(03)00042-X.

N. Robertson and P. D. Seymour. Graph minors. XX. Wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004. URL https://doi.org/10.1016/j.jctb.2004.08.001.

N. Robertson and P. D. Seymour. Graph minors. XXII. Irrelevant vertices in linkage problems. *Journal of Combinatorial Theory, Series B*, 102(2):530–563, 2012. URL https://doi.org/10.1016/j.jctb.2007.12.007.

N. Robertson, P. D. Seymour, and R. Thomas. Quickly excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 62(2):323–348, 1994. URL https://doi.org/10.1006/jctb.1994.1073.

D. Seese. The structure of models of decidable monadic theories of graphs. *Annals of Pure and Applied Logic*, 53(2):169–195, 1991. URL https://doi.org/10.1016/0168-0072(91)90054-P.

P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994. URL https://doi.org/10.1007/BF01215352.

M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012. ISBN 978-1133187790.

D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC 1981)*, pages 114–122. ACM, 1981. URL https://doi.org/10.1145/800076.802464.

R. P. Soares. *Pursuit-evasion, decompositions and convexity on graphs*. PhD thesis, Université Nice Sophia Antipolis, 2013. URL https://theses.hal.science/tel-00908227.

K. Suchan and I. Todinca. On powers of graphs of bounded NLC-width (clique-width). *Discrete Applied Mathematics*, 155(14):1885–1893, 2007. URL https://doi.org/10.1016/j.dam.2007.03.014.

S. Szeider. On fixed-parameter tractable parameterizations of SAT. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 188–202. Springer, 2003. URL https://doi.org/10.1007/978-3-540-24605-3_15.

J. A. Telle and A. Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997. URL https://doi.org/10.1137/S0895480194275825.

D. M. Thilikos, M. J. Serna, and H. L. Bodlaender. Cutwidth I: A linear time fixed parameter algorithm. *Journal of Algorithms*, 56(1):1–24, 2005. URL https://doi.org/10.1016/j.jalgor.2004.12.001.

R. Thomas. A Menger-like property of tree-width: The finite case. *Journal of Combinatorial Theory, Series B*, 48(1):67–76, 1990. URL https://doi.org/10.1016/0095-8956(90)90130-R.

M. Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998. URL https://doi.org/10.1006/inco.1997.2697.

S. Torunczyk. Flip-width: Cops and robber on dense graphs. In *Proceedings of the 64th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2023)*, pages 663–700. IEEE, 2023. URL https://doi.org/10.1109/FOCS57990.2023.00045. Full version: https://arxiv.org/abs/2302.00352.

G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA 2009)*, volume 5757 of *LNCS*, pages 566–577. Springer, 2009. URL https://doi.org/10.1007/978-3-642-04128-0_51.

E. Wanke. k-NLC graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54 (2-3):251–266, 1994. URL https://doi.org/10.1016/0166-218X(94)90026-4.

P. Wollan. The structure of graphs not admitting a fixed immersion. *Journal of Combinatorial Theory, Series B*, 110:47–66, 2015. URL https://doi.org/10.1016/j.jctb.2014.07.003.

Y. Wu, P. Austrin, T. Pitassi, and D. Liu. Inapproximability of treewidth and related problems. *Journal of Artificial Intelligence Research*, 49:569–600, 2014. URL https://doi.org/10.1613/jair.4030.

N. Yolov. Minor-matching hypertree width. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 219–233. SIAM, 2018. URL https://doi.org/10.1137/1.9781611975031.16. Full version: http://arxiv.org/abs/1704.02939.

D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3(1):103–128, 2007. URL https://doi.org/10.4086/toc.2007.v003a006.